**Владимирский государственный университет**

**А. Б. ГРАДУСОВ  А. В. ШУТОВ**

# БАЗЫ ДАННЫХ

## Проектирование баз данных

**Учебно-практическое пособие**

**Владимир 2024**

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»

А. Б. ГРАДУСОВ  А. В. ШУТОВ

# БАЗЫ ДАННЫХ

## Проектирование баз данных

Учебно-практическое пособие

*Электронное издание*

Владимир 2024

ISBN 978-5-9984-1888-4
© ВлГУ, 2024

УДК 004.65
ББК 16.35

Рецензенты:
Кандидат физико-математических наук, доцент
зав. кафедрой физико-математического образования
и информационных технологий
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых
*Ю. Ю. Евсеева*

Кандидат физико-математических наук, доцент
доцент кафедры специальной техники и информационных технологий
Владимирского юридического института
Федеральной службы исполнения наказаний
*А. В. Хорошева*

Издается по решению редакционно-издательского совета ВлГУ

**Градусов, А. Б.** БАЗЫ ДАННЫХ. Проектирование баз данных [Электронный ресурс] : учеб.-практ. пособие / А. Б. Градусов, А. В. Шутов ; Владим. гос. ун-т им. А. Г. и Н. Г. Столетовых. – Владимир : Изд-во ВлГУ, 2024. – 259 с. – ISBN 978-5-9984-1888-4. – Электрон. дан. (3,43 Мб). – 1 электрон. опт. диск (CD-ROM). – Систем. требования: Intel от 1,3 ГГц ; Windows XP/7/8/10 ; Adobe Reader ; дисковод CD-ROM. – Загл. с титул. экрана.

Изложены основы теории проектирования баз данных. Рассмотрены этапы инфологического, логического и физического проектирования баз данных.

Предназначено для студентов вузов всех форм обучения направлений подготовки 09.03.03 – Прикладная информатика и 27.03.04 – Управление в технических системах.

Рекомендовано для формирования профессиональных компетенций в соответствии с ФГОС ВО.

Табл. 36. Ил. 93. Библиогр.: 12 назв.

ISBN 978-5-9984-1888-4 © ВлГУ, 2024

## ОГЛАВЛЕНИЕ

# ПРЕДИСЛОВИЕ

Проектирование базы данных (БД) – одна из наиболее сложных и ответственных задач, связанных с созданием информационных систем.

Цель пособия состоит в формировании концептуальных представлений о принципах проектирования баз данных.

Проектирование базы данных – это процесс, в основе которого лежит методология, подразумевающая последовательное выполнение определенных этапов и шагов. Пособие состоит из глав, каждая из которых соответствует одному из этапов проектирования. В конце каждой темы внутри главы приводятся вопросы для самопроверки и практические задания. В главе 1 описаны требования, предъявляемые к базам данных, а также основные этапы процесса проектирования баз данных. Глава 2 посвящена инфологическому проектированию баз данных. Рассмотрены основные конструктивные элементы модели «сущность – связь»: сущность, атрибут, связь. Приведен пример разработки инфологической модели учебной базы данных. В главе 3 рассмотрены задачи логического проектирования баз данных. Приведены правила преобразования конструктивных элементов концептуальной модели в реляционную модель. Раскрыты понятие избыточного дублирования данных и проблемы при работе с БД, к которым оно приводит. Описан метод нормальных форм. Глава 4 посвящена вопросам физического проектирования баз данных. Рассмотрен перенос логической модели базы данных в среду выбранной СУБД. Описана декларативная поддержка целостности данных. Представлены различные средства процедурной поддержки целостности данных: хранимые процедуры, триггеры и пользовательские функции. Приведено

понятие транзакций и описаны возможности управления ими. Отмечена особая роль транзакций в многопользовательском режиме работы баз данных. Рассмотрены вопросы, связанные с размещением данных и доступом к ним в реляционных базах данных.

Пособие ориентировано на использование в учебном процессе, поэтому теоретический материал по вопросам проектирования баз данных дополнен примерами по каждой теме для грамотного использования полученных знаний.

Материал пособия соответствует содержанию учебной дисциплины «Базы данных» федеральных государственных образовательных стандартов по направлениям подготовки бакалавров 09.03.03 «Прикладная информатика», 27.03.04 «Управление в технических системах».

# Глава 1. МЕТОДОЛОГИЯ ПРОЕКТИРОВАНИЯ БАЗ ДАННЫХ

Методология проектирования баз данных может рассматриваться как совокупность методов и средств, последовательное применение которых обеспечивает разработку проекта базы данных, удовлетворяющего заданным целям. Она предусматривает разбиение всего процесса на несколько этапов, каждый из которых, в свою очередь, состоит из нескольких шагов.

На каждом шаге разработчику предлагается набор технических приемов, позволяющих решать задачи, стоящие перед ним на данном шаге разработки.

Проект базы данных должен быть точен и выверен, потому что он является фундаментом информационной системы, которая будет использоваться достаточно долго и многими пользователями. И как в любом здании, можно достраивать этажи, переделывать окна, двери, крышу, но заменить фундамент, не разрушив всего здания, невозможно.

## 1.1. Требования, предъявляемые к базам данных

Проектирование базы данных представляет собой трудоемкий процесс, требующий совместных усилий аналитиков, проектировщиков и пользователей. При проектировании базы данных необходимо учитывать тот факт, что правильно спроектированная база данных должна удовлетворять следующим требованиям.

1. *База данных должна удовлетворять актуальным информационным потребностям пользователей.*

Организация данных в базе данных должна по структуре и содержанию соответствовать решаемым задачам. База данных должна быть адекватной моделью предметной области, т. е. каждому объекту предметной области должны соответствовать данные в памяти ЭВМ, а каждому процессу – адекватные процедуры обработки данных.

2. *База данных должна иметь минимальную избыточность данных.*

Избыточность означает дублирование данных. Избыточность данных, если она существует, влечет две опасности:

- неоправданно большой расход памяти;

• появление проблем при попытке удаления, добавления или редактирования данных. Например, при внесении изменений в запись может случиться так, что отдельные экземпляры этой записи, хранящиеся в различных местах машинной памяти, окажутся нескорректированными.

Современные методы проектирования баз данных позволяют свести к минимуму дублирование данных.

3. *В базе данных должна поддерживаться целостность данных.*

В процессе работы с базой данных возможно искажение или разрушение данных в результате неосторожных действий пользователей, ошибок в программах и сбоев оборудования.

Целостность данных означает то, что в базе данных должны храниться только правильные данные. Для обеспечения целостности на данные, хранящиеся в БД, накладывают ограничения, т. е. логические условия, в соответствии с которыми данные считаются правильными. Например, количество хранящегося на складе товара не может быть отрицательным числом; оплата заказов должна осуществляться не позднее 25-го числа каждого месяца. Первое ограничение можно реализовать с помощью языка SQL при создании таблицы. Для реализации второго ограничения потребуется использование программных средств, например триггеров.

Выполнимость ограничений проверяется СУБД.

4. *База данных должна обеспечивать получение требуемых данных за приемлемое время, т. е. обеспечивать заданную производительность.*

База данных должна быть спроектирована таким образом, чтобы при её эксплуатации соблюдались ограничения на время реакции системы на запросы и модификацию данных. Структура базы данных должна позволять включать новые и удалять устаревшие данные, корректировать хранимые данные без разрушения логических связей, установленных в схеме базы данных. Для этого схема базы данных должна быть правильно разработана, а операции ведения БД не должны нарушать схему базы данных.

5. *В базе данных должна быть обеспечена защита данных.*

Доступ к данным, размещаемым в базе данных, должны иметь только лица с соответствующими полномочиями. Проект базы данных должен включать в себя описание защиты данных от несанкцио-

нированного доступа. В базе данных должны быть предусмотрены средства восстановления данных после программных сбоев и сбоев оборудования. Защита от сбоев – внутренняя функция СУБД, но требования к настройке механизмов защиты также выдвигаются на этапе проектирования БД.

## 1.2. Этапы проектирования базы данных

Проектирование базы данных представляет собой процесс последовательного отображения объектов реального мира в данные, хранящиеся в памяти ЭВМ. Проектирование БД базируется на информационном моделировании предметной области. В основе данного подхода лежит положение об определяющей роли и независимости данных [1 – 3]. Технологии, ориентированные на информационное моделирование, сначала определяют структуры данных, а затем описывают процессы, использующие эти данные.

Процесс проектирования базы данных представляет собой последовательность простых, обычно итеративных, шагов проектирования, в ходе которых строятся информационные модели различных уровней представления данных.

В рассматриваемой методологии весь процесс проектирования базы данных подразделяется на следующие этапы проектирования:

1) инфологическое (концептуальное) проектирование базы данных;
2) логическое (даталогическое) проектирование базы данных;
3) физическое проектирование базы данных.

***Инфологическое проектирование базы данных.*** Цель этапа – создание концептуальной модели предметной области на основе требований пользователей.

Концептуальная модель данных – это семантическая (смысловая) модель предметной области, основанная на представлениях пользователей о предметной области. Такая модель создаётся без ориентации на какую-либо модель данных и конкретную систему управления базами данных (СУБД). В такой модели отсутствуют какие-либо понятия, связанные с ЭВМ, памятью ЭВМ, со способами размещения данных в памяти ЭВМ. По сути, концептуальная модель представляет собой информационное описание предметной области с учетом логи-

ческих взаимосвязей, поэтому её еще называют инфологической (информационно-логической) моделью. Термин «инфологическая модель» в данном случае означает модель, ориентированную на человека. Речь идет о средстве для выражения и передачи понимания того, что собой представляет предметная область базы данных.

Чаще всего концептуальная модель базы данных включает в себя описание:

- информационных объектов и связей между ними;
- ограничений целостности, т. е. требований к допустимым значениям данных и связям между ними.

Модель предметной области может быть описана любым удобным для разработчика способом (словесное описание, набор формул, диаграмма потоков данных и т. п.).

Более информативны и полезны при разработке баз данных описания предметной области, выполненные при помощи специализированных графических нотаций. Имеется большое количество методик описания предметной области. Из наиболее известных можно назвать методику структурного анализа SADT [4] и основанную на ней методологию функционального моделирования IDEF0, методику объектно-ориентированного анализа UML [5], методику, основанную на семантической объектной модели [6], и др. При проектировании баз данных наиболее часто используется метод «сущность – связь» [7], который представляет собой концептуальную модель в виде диаграммы «сущность – связь» (Entity-Relation Diagram, ER-диаграммы).

***Логическое проектирование базы данных.*** Цель этапа – преобразование концептуальной модели в даталогическую модель с учетом принимаемой модели базы данных (иерархической, сетевой, реляционной и т. д.), но независимо от конкретной СУБД и других деталей физической реализации.

Таким образом, логическое проектирование предполагает преобразование концептуальной модели в модель данных, ориентированную на среду хранения и обработки данных.

На этапе логического проектирования учитывается специфика даталогической модели данных, но не учитывается специфика конкретной СУБД.

Для реляционной модели данных даталогическая модель – это набор схем таблиц обычно с указанием первичных ключей, а также «связей» между таблицами, представляющих собой внешние ключи.

Преобразование концептуальной модели в логическую модель в большинстве случаев осуществляется по формальным правилам. Этот этап может быть в значительной степени автоматизирован.

Одна из важных задач этого этапа проектирования реляционных баз данных заключается в нормализации реляционной модели, которая выполняется с использованием специальных правил нормализации. Цель нормализации – минимизировать проблемы при работе с базой данных, связанные со вставкой, с удалением и обновлением данных. На этапе даталогического проектирования решается еще одна очень важная задача – обеспечение целостности данных. Под целостностью базы данных понимается ее свойство, означающее, что она содержит полную, непротиворечивую информацию, адекватно отражающую предметную область.

***Физическое проектирование базы данных.*** При логическом проектировании отвечают на вопрос, что надо сделать, а при физическом – выбирают способ, как это сделать.

Цель этапа физического проектирования – создание описания конкретной реализации базы данных, размещаемой во внешней памяти в среде выбранной СУБД.

Будем считать, что физическая модель данных реализована средствами реляционной СУБД, хотя это необязательно. В этом случае физическая модель данных описывает данные на языке DDL (Data Definition Language) – языке определения данных, который поддерживается выбранной СУБД.

Как уже отмечалось выше, для обеспечения целостности на данные, хранящиеся в БД, накладывают ограничения. Ограничения, имеющиеся в даталогической модели данных, реализуются различными средствами СУБД, например при помощи декларативных ограничений целостности, триггеров, хранимых процедур. Способ их реализации зависит от выбранной СУБД. Одни системы предоставляют больше возможностей, другие – меньше.

На этом шаге проектирования выявляются транзакции, которые будут выполняться в проектируемой базе данных. *Транзакцией* называется некоторая неделимая последовательность операций над дан-

ными, выполняемая как единое целое. Если по какой-то причине какая-либо из операций не выполнилась, то транзакция отменяется полностью. При этом происходит «откат» путем отмены всех уже выполненных изменений. Современные СУБД позволяют отслеживать ход выполнения транзакции от начала до ее завершения. Однако при проектировании базы данных необходимо предусмотреть обработку ошибок, которые могут произойти при выполнении транзакции.

Также на этапе физического проектирования необходимо определить способы хранения данных (кластеризация, хеширование) и доступа к данным (индексирование) и создать соответствующие индексы и кластеры (если нужно).

Если пользователей базы данных можно разделить на группы по характеру решаемых задач, то для каждой группы создаётся свой набор прав доступа к объектам БД.

Централизованное хранение данных – причина высокой вероятности того, что двум или более пользователям одновременно понадобятся одни и те же данные. Если один из пользователей обращается к данным, а другой в то же время вносит в них изменения, то без принятия специальных мер будут получены противоречивые данные. Объясняется это тем, что процесс обновления данных требует определенного времени, в течение которого одни и те же данные оказываются на разных стадиях обновления. При обращении к таким данным параллельно работающих программ будут получены противоречивые сведения. В системе управления базами данных существуют сложные механизмы блокирования обновляемых данных от доступа к ним других пользователей.

Из вышеизложенного следует, что проектирование базы данных представляет собой процесс последовательного отображения объектов реального мира в данные, хранящиеся в памяти ЭВМ.

При разработке базы данных можно выделить несколько уровней моделирования, при помощи которых происходит переход от предметной области к конкретной реализации базы данных средствами конкретной СУБД. Можно выделить следующие уровни:

1) сама предметная область;
2) концептуальная модель предметной области;
3) логическая модель данных;
4) физическая модель данных.

### 1.3. Последовательность проектирования базы данных

Процесс проектирования базы данных включает в себя следующие шаги.

1. Создание концептуальной модели данных.

1.1. Составление описания предметной области.

1.2. Определение сущностей.

1.3. Определение связей между сущностями.

1.4. Определение атрибутов сущностей и связей.

1.5. Определение значений (доменов) атрибутов.

1.6. Определение ключей сущностей.

1.7. Обсуждение концептуальной модели с пользователями.

2. Создание логической модели данных.

2.1. Определение набора таблиц исходя из концептуальной модели данных.

2.2. Нормализация таблиц.

2.3. Определение ограничений целостности данных.

2.4. Обсуждение разработанной логической модели данных с пользователями.

3. Создание физической модели данных.

3.1. Выбор СУБД.

3.2. Написание скриптов для создания основных объектов базы данных на языке SQL в синтаксисе выбранной СУБД.

3.3. Написание скриптов для создания вспомогательных объектов базы данных (представления, хранимые процедуры, триггеры, роли и т. д.).

3.4. Организация файлов и индексов.

3.5. Определение прав доступа пользователей к объектам базы данных.

3.6. Разработка стратегии защиты базы данных.

## Вопросы для самопроверки

1. Какие требования к базе данных вы знаете?
2. Каковы основные этапы проектирования баз данных?
3. Какова цель этапа концептуального проектирования?
4. Какова цель этапа логического проектирования?
5. Какова цель этапа физического проектирования?
6. В чем отличие физического проектирования от логического?
7. Кратко расскажите, что происходит на каждом этапе проектирования базы данных.
8. Какие шаги выполняются на стадии концептуального проектирования?
9. Какие шаги выполняются на стадии логического проектирования?
10. Какие шаги выполняются на стадии физического проектирования?

# Глава 2. ИНФОЛОГИЧЕСКОЕ ПРОЕКТИРОВАНИЕ БАЗ ДАННЫХ

Инфологическое проектирование заключается в построении концептуальной модели, отражающей предметную область и информационные потребности пользователей.

Инфологическое проектирование прежде всего связано с попыткой представления семантики предметной области в модели БД, т. е. это моделирование структуры данных с опорой на смысл этих данных.

Зачем нужна концептуальная модель и какую пользу она дает проектировщикам? Следует помнить, что процесс проектирования длительный, он требует обсуждений с заказчиком, со специалистами в предметной области. Следовательно, концептуальная модель должна включать в себя такое формализованное описание предметной области, которое легко «читается» не только специалистами по базам данных. И это описание должно быть настолько емким, чтобы можно было оценить глубину и корректность проработки проекта БД, и, конечно, оно не должно быть привязано к конкретной СУБД.

Прежде чем приступать к созданию базы данных, проектировщик должен сформировать понятия о предметах, фактах и событиях, которыми будет оперировать разрабатываемая база данных. Для того чтобы привести эти понятия к той или иной модели данных, необходимо заменить их информационными представлениями. Один из наиболее удобных инструментов унифицированного представления данных, независимого от реализующего его программного обеспечения, – модель «сущность – связь», или ER-модель (ER – аббревиатура от слов Entity (сущность) и Relation (связь)). Эта модель была разработана Питером Ченом в 1976 г. с целью упрощения задачи проектирования БД [7].

Модель «сущность – связь» основывается на некоторой важной семантической информации о реальном мире и предназначена для логического представления данных. Она определяет значения данных в контексте их взаимосвязи с другими данными. На основе модели «сущность – связь» могут быть порождены все существующие даталогические модели данных (иерархическая, сетевая, реляционная, объектная), поэтому она наиболее общая.

Построение концептуальной модели может выполняться как «вручную», так и с использованием автоматизированных средств проектирования. Большинство современных CASE-средств содержат инструментальные средства для описания данных в формализме этой модели. Кроме того, разработаны методы автоматического преобразования проекта базы данных из модели «сущность – связь» в даталогическую или физическую модель, соответствующую конкретной СУБД.

## 2.1. Конструктивные элементы модели «сущность – связь»

Модель «сущность – связь» – это абстрактная модель предметной области, которая используется на этапе инфологического проектирования базы данных.

Важное свойство модели «сущность – связь» состоит в том, что она может быть представлена в виде графической схемы – диаграммы «сущность – связь». Это значительно облегчает анализ предметной области. Используются разные графические нотации отображения модели «сущность – связь» (классическая нотация Питера Чена, нотация Ричарда Баркера, нотация Джеймса Мартина, нотации IDEF и др.). В дальнейшем будем использовать некий гибрид нотаций Чена (обозначение сущностей, связей и атрибутов) и Мартина (обозначение мощности и класса принадлежности связей).

Общее для всех подходов – использование трех основных конструктивных элементов для представления составляющих предметной области: сущности, связи, атрибута.

### *2.1.1. Сущности предметной области*

Различают понятия «сущность» и «экземпляр сущности».

*Сущность* – это класс однотипных объектов, информация о которых должна быть учтена в модели. Примерами сущностей могут быть такие классы объектов, как «Сотрудник», «Автомобиль», «Товар», «Книга». На диаграмме «сущность – связь» каждая сущность изображается в виде прямоугольника, внутри которого записано имя сущности. Название сущности – имя существительное в единственном числе, например: ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА, КАФЕД-

РА, ГРУППА. Имя сущности должно быть уникально в рамках модели данных.

Каждая сущность состоит из экземпляров.

*Экземпляр сущности* – это конкретный представитель данной сущности. Например, экземплярами сущности ГОРОД могут быть города Москва, Владимир, Тула и т. д.

Модель «сущность – связь» строится на уровне сущностей, а не на уровне отдельных экземпляров сущностей.

Экземпляры сущности отличаются друг от друга и однозначно идентифицируются, т. е. сущности должны иметь некоторые свойства, уникальные для каждого экземпляра этой сущности.

### *2.1.2. Описание атрибутов сущностей*

Именованная характеристика, являющаяся некоторым свойством сущности, называется *атрибутом сущности*. Атрибуты содержат значения, которые описывают каждый экземпляр сущности и составляют основную часть информации, сохраняемую в базе данных.

Наименование атрибута должно быть выражено существительным в единственном числе (возможно, с характеризующими прилагательными). Например, у сущности СТУДЕНТ могут быть такие атрибуты: «Номер зачетной книжки», «Фамилия», «Имя», «Отчество», «Номер группы», «Дата рождения», «Домашний адрес».

Атрибуты могут быть:

- простыми и составными;
- обязательными и необязательными;
- однозначными и многозначными;
- основными и производными;
- ключевыми и неключевыми.

*Простой атрибут* состоит из одного компонента, его значение неделимо. *Составной атрибут* является комбинацией нескольких компонентов. Например, такой составной атрибут, как адрес, может включать в себя набор значений: индекс, страна, город, улица, дом. При построении концептуальной модели такие атрибуты допустимы. При переходе к даталогической модели все зависит от выбранной модели данных и возможностей СУБД. Например, в реляционной модели составных атрибутов нет, их надо представлять как множество простых атрибутов.

Значение *обязательного атрибута* всегда устанавливается при помещении данных в базу данных. Значение *необязательного атрибута* может быть пропущено, т. е. ему может быть присвоено неопределенное значение (NULL-значение).

*Однозначные атрибуты* могут иметь только одно значение для каждого экземпляра сущности, а *многозначные атрибуты* – много значений. Например, в компании может быть только один директор и несколько номеров телефонов. При построении модели «сущность – связь» допустимо указывать многозначные атрибуты, но при логическом проектировании от подобных атрибутов придется избавляться. В частности, реляционная модель не поддерживает многозначные атрибуты. Чтобы решить эту проблему, обычно в модель «сущность – связь» вводят дополнительную сущность. В нашем примере с множеством телефонов это может быть сущность ТЕЛЕФОН_ФИРМЫ.

Значение *основного атрибута* не зависит от других атрибутов. Значение *производного атрибута* вычисляется на основе значений других атрибутов. Например, значение атрибута «Стоимость товара на складе» определяется как произведение значений атрибутов «Количество товара» и «Цена товара».

Набор атрибутов, однозначно идентифицирующий конкретный экземпляр сущности, называют *ключом.* Неизбыточность заключается в том, что удаление любого атрибута из ключа нарушает его уникальность. Как видно из определения, понятие ключа сущности аналогично понятию ключа реляционной таблицы.

Сущность может иметь несколько различных ключей.

На диаграммах «сущность – связь» атрибуты изображаются овалами (рис. 2.1), которые соединяются с соответствующими сущностями ненаправленными ребрами. Ключевые атрибуты подчеркиваются.

Важно понимать, что сущности нужно в концептуальном плане отделять от атрибутов, которые их описывают, так как значения атрибутов могут меняться, в то время как описываемый ими объект остается прежним. Например, у человека может измениться рост, вес, семейное положение, но это будет тот же самый человек.

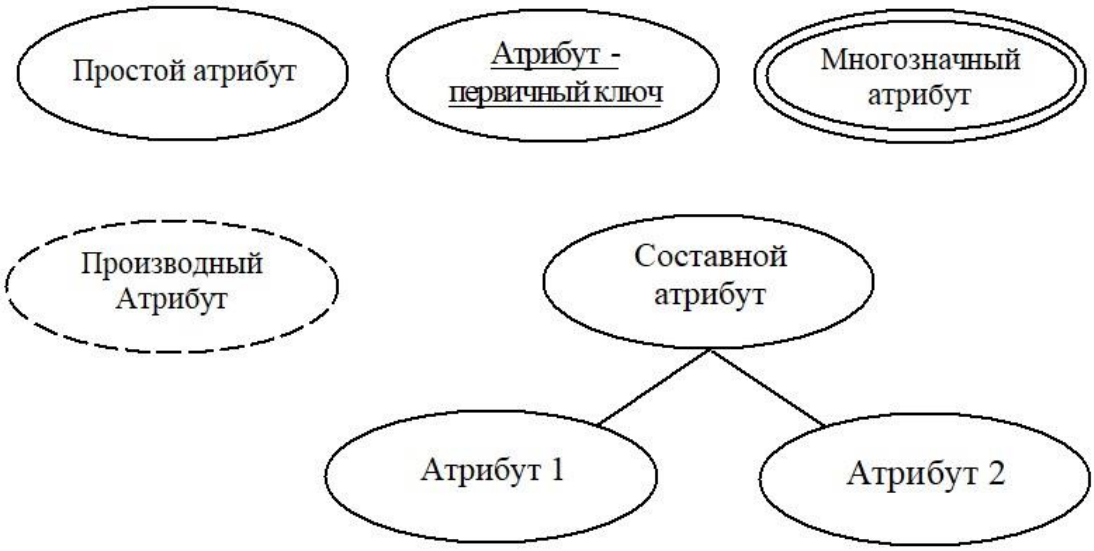



Рис. 2.1. Представление атрибутов на диаграммах модели «сущность – связь»

Абсолютное различие между сущностями и атрибутами отсутствует. Атрибут является таковым только в связи с сущностью. В другом контексте атрибут может выступать как самостоятельная сущность. Например, в базе данных должна храниться информация о том, какое учебное заведение закончили специалисты предприятия. Если больше никакой специальной обработки по этому признаку не производится, то «Учебное заведение» следует считать атрибутом соответствующей сущности. Если же в предметной области отражается дополнительная информация об учебных заведениях, например их адрес, тип и тому подобное, то УЧЕБНОЕ_ЗАВЕДЕНИЕ следует рассматривать как самостоятельную сущность.

С понятием атрибута связано понятие «домен атрибута».

***2.1.2.1. Домен атрибута.*** *Домен атрибута* – набор допустимых значений одного или нескольких атрибутов. Например, количество квартир в многоквартирном доме находится в пределах от 1 до 500. Следовательно, набор допустимых значений для атрибута «Номер квартиры» можно определить как набор целых чисел от 1 до 500. Фамилия студента представляет собой строку, содержащую, как правило, до 20 символов. Поэтому для атрибута «Фамилия студента» набор допустимых значений может представлять собой набор строк, максимальная длина которых составляет 20 символов. Атрибут «Пол сотрудника» может принимать одно из двух значений – «Женский» или «Мужской».

Домен – это семантическое понятие. Его можно рассматривать как подмножество значений некоторого типа данных, имеющих определенный смысл. Домен характеризуется следующими свойствами [8]:

- имеет уникальное имя;
- определен на некотором простом (скалярном) типе данных или на другом домене;
- может иметь некоторое логическое условие, позволяющее описать подмножество данных, допустимых для данного домена;
- может быть задан перечислением множества допустимых элементов данных;
- несет определенную смысловую нагрузку.

Примеры доменов:

- домен «зарплата» – множество вещественных чисел, превышающих МРОТ (минимальный размер оплаты труда);
- домен «образование» – множество строк символов, заданное перечислением (начальное, среднее общее, среднее специальное, высшее);
- домен «фамилия» – множество строк символов не длиннее 20;
- домен «оценка» – множество целых чисел в интервале от 2 до 5;
- домен «номер» – множество положительных целых чисел.

Понятие домена помогает правильно моделировать предметную область, исключая некорректные сравнения семантически разнородных данных. Некорректно с логической точки зрения сравнивать значения из различных доменов, даже если они имеют одинаковый тип. В этом проявляется смысловое ограничение доменов. Домены могут представлять собой комбинации других доменов, образованные по принципу агрегации. Например, домен «дата» может состоять из подчиненных доменов «день», «месяц», «год», а домен «адрес» – из доменов «город», «улица», «номер дома». На одном и том же домене могут быть определены несколько атрибутов. Например, на домене «выплаты» могут быть определены атрибуты «зарплата», «премия», «пособие», «ссуда» и т. д. Однако любой атрибут может быть определен только на одном домене. Итак, область определения атрибутов – это специфические множества элементов данных, или домены.

На диаграммах «сущность – связь» домены обычно не указываются. Эта информация помещается в отдельный документ, называемый словарем данных.

***2.1.2.2. Выбор ключа сущности.*** Для того чтобы один экземпляр сущности можно было отличить от другого экземпляра этой же сущности, выбирается один или несколько атрибутов с уникальным значением (ключ).

Выбор и задание ключа сущности должны подчиняться следующим правилам.

1. По возможности ключ должен быть несоставным или составленным из минимального числа атрибутов, т. е. наиболее компактным из всех потенциальных ключей.

2. Предпочтительный тип данных для ключа – целочисленный. Нецелесообразно использовать ключи с длинными текстовыми значениями.

3. Значения ключа не должны подвергаться частым модификациям (в идеальном случае вообще не должны меняться).

4. Правила модификации ключа должны контролироваться внутренней функциональностью предметной области, а не решениями, которые принимаются за ее пределами. Например, в базе данных, разрабатываемой для нужд деканата, для сущности СТУДЕНТ не следует назначать первичным ключом такие обладающие уникальностью атрибуты, как серия и номер паспорта, так как их изменение может быть инициировано самим студентом, а не администрацией факультета.

5. Если среди информации, собранной о сущности, не удается выделить данные, претендующие на роль первичного ключа, то рекомендуется рассмотреть возможность создания *суррогатного ключа.* Это искусственно сформированные внутренние номера без смысла вне базы данных, которые однозначно определяют элемент сущности. Использование суррогатных ключей предпочтительнее по сравнению с использованием составных ключей, так как уменьшается вероятность нарушения целостности данных из-за ошибок в одном из атрибутов.

На рис. 2.2 приведен пример сущности СТУДЕНТ со своими атрибутами.

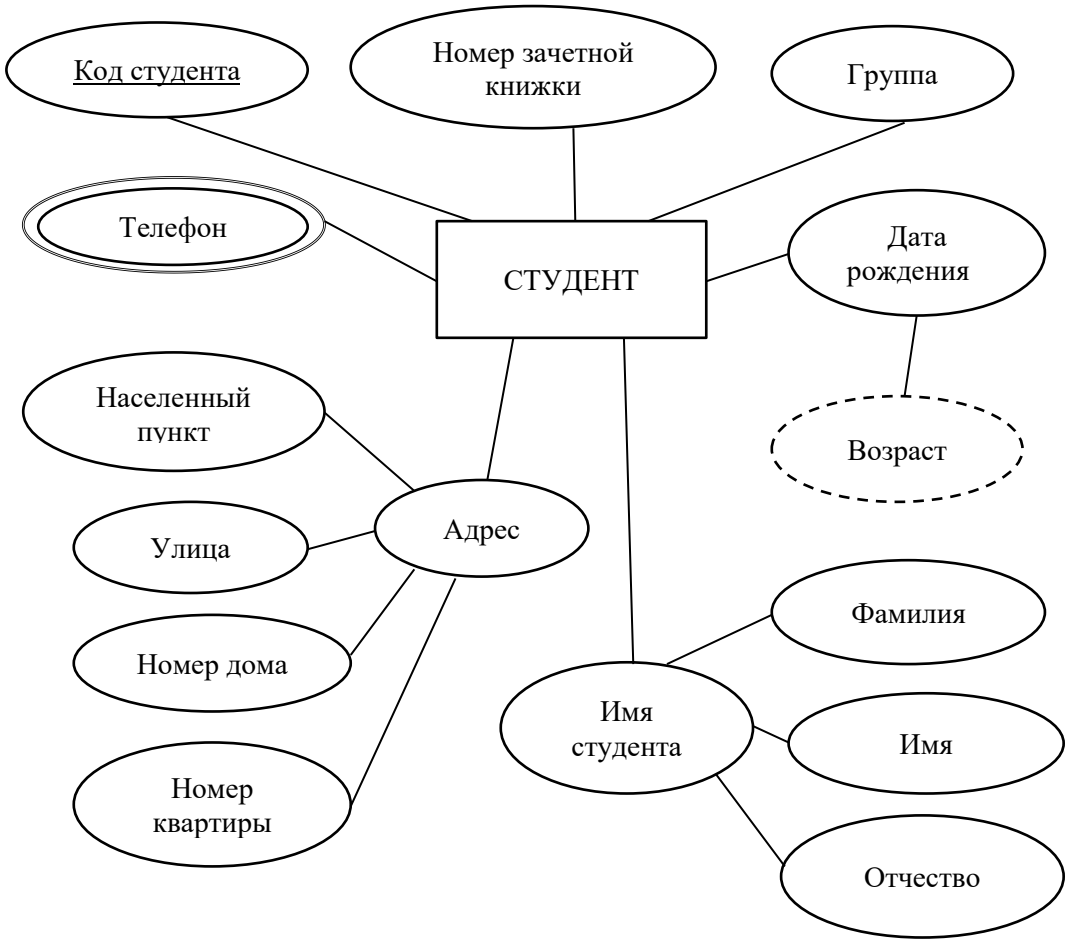



Рис. 2.2. Диаграмма сущности СТУДЕНТ

Сущность СТУДЕНТ имеет следующие атрибуты:

- атрибут «Код студента» – суррогатный ключ, имеющий уникальные целочисленные значения;
- атрибут «Номер зачетной книжки» – простой атрибут, который можно реализовать как строку символов, поскольку номер зачетной книжки, кроме цифр, может содержать и буквы;
- атрибут «Имя студента» – составной атрибут, включающий в себя простые атрибуты «Фамилия», «Имя», «Отчество», значения которых – строки длиной 15 символов;
- атрибут «Адрес» – составной атрибут, включающий с себя простые атрибуты «Населенный пункт», «Улица», «Номер дома», «Номер квартиры». При этом значения атрибутов «Населенный пункт», «Улица» – строки длиной 20 символов, а значение атрибута

«Номер дома» – строка длиной 5 символов. Значение атрибута «Номер квартиры – целое число;

- атрибут «День рождения» – простой атрибут типа «Дата»;
- атрибут «Возраст студента» – вычисляемый атрибут, значение которого определяется как разность текущей даты и значения атрибута «День рождения»;
- атрибут «Номер группы» – простой атрибут, который можно реализовать как строку символов, поскольку номер группы, кроме цифр, может содержать и буквы;
- атрибут «Номер телефона» – многозначный атрибут, который может быть реализован как массив или коллекция и т. п.

В табл. 2.1 приведен словарь данных с описанием атрибутов сущности СТУДЕНТ.

Таблица 2.1

Словарь данных с описаниями атрибутов сущности СТУДЕНТ

| Имя сущности | Атрибуты | Описание | Ключ | Тип данных и длина | Неопределенные значения |
|---|---|---|---|---|---|
| СТУДЕНТ | Код студента | Однозначно определяет студента | Да | Целое число | Нет |
| | Имя студента | Составной атрибут | Нет | | Нет |
| | Фамилия | Фамилия студента | Нет | Строка длиной до 15 символов | Нет |
| | Имя | Имя студента | Нет | Строка длиной до 15 символов | Нет |
| | Отчество | Отчество студента | Нет | Строка длиной до 15 символов | Нет |
| | | | | | |

*Окончание табл. 2.1*

| Имя сущ-ности | Атрибуты | Описание | Ключ | Тип данных и длина | Неопре-деленные значения |
|---|---|---|---|---|---|
| | Номер зачетной книжки | Номер зачетной книжки | Нет | Строка длиной до 10 символов | Нет |
| | Группа | Название группы, в которой учится студент | Нет | Строка длиной до 6 символов | Нет |
| | Дата рождения | Число, месяц и год рождения студента | Нет | Дата | Нет |
| | Возраст | Вычисляемый атрибут, значение которого определя-ется как разность текущей даты и значения атрибута «День рождения» | Нет | Целое число | Нет |
| | Адрес | Составной атрибут | Нет | | Да |
| | Населен-ный пункт | Название населенного пункта | Нет | Строка длиной до 20 символов | Да |
| | Улица | Название улицы | Нет | Строка длиной до 20 символов | Да |
| | Номер дома | Номер дома | Нет | Строка длиной до 5 символов | Да |
| | Номер квартиры | Номер квартиры | Нет | Целое число | Да |
| | Номер телефона | Многозначный атрибут, так как студент может иметь несколько телефонов | Нет | Массив целых чисел | Да |

### *2.1.3. Описание связей*

Сущности не существуют отдельно друг от друга. Между ними имеются реальные связи, и они должны быть отражены в информационной модели предметной области. Связь представляет собой соединение двух или более сущностей. Каждая связь реализуется через значения атрибутов сущностей.

Количество сущностей, которые охвачены данной связью, называют *степенью связи.* Степень связи бывает унарной, бинарной, *n*-арной.

*Унарная связь* существует тогда, когда связь поддерживается внутри единственной сущности. Такую связь еще называют *рекурсивной*.

Если в связи участвует две сущности, то связь называется *бинарной.*

Для описания связей со степенью больше двух принято применять термин *сложная*, или *n-арная, связь.*

Связь между сущностями на диаграмме «сущность – связь» представляют в виде соединяющего их отрезка, дополненного ромбом, внутри которого указывается имя связи. Имя связи выражается глаголом: «Имеет», «Принадлежит» и т. п.

Связь устанавливается между экземплярами сущностей. Например, для сущностей КЛИЕНТ и ЗАКАЗ существует связь «Оформляет» (рис. 2.3), которая устанавливается между конкретным клиентом и его заказами.


Рис. 2.3. Пример связи между сущностями КЛИЕНТ и ЗАКАЗ

В практике проектирования баз данных наиболее часто используются бинарные связи.

Бинарные связи имеют следующие характеристики:

1) мощность связи;

2) класс принадлежности сущностей.

*Мощность связи* – максимальное количество экземпляров одной сущности, связанных с одним экземпляром другой сущности.

Связь, максимальная мощность которой в обоих направлениях равна одному, называется *один-к-одному (1:1)*. Связь один-к-одному означает, что экземпляр одной сущности связан только с одним экземпляром другой сущности. Связь один-к-одному чаще всего свидетельствует о том, что на самом деле мы имеем всего одну сущность, разделенную на две.

Связь, максимальная мощность которой равна одному в одном направлении и многим в другом, называется *один-ко-многим (1:M)*. Связь 1:M означает, что один экземпляр сущности, расположенный слева по связи, может быть связан с несколькими экземплярами сущности, расположенными справа по связи. Это наиболее часто используемый тип связи.

Связь, максимальная мощность которой равна многим в обоих направлениях, называется *многие-ко-многим (M:M)*. Связь многие-ко-многим означает, что один экземпляр первой сущности может быть связан с несколькими экземплярами второй сущности, и наоборот: один экземпляр второй сущности может быть связан с несколькими экземплярами первой сущности.

В некоторых случаях в обозначении связи один-ко-многим и многие-ко-многим вместо символа M указывают конкретное число. Например, связь между сущностями ФУТБОЛЬНАЯ_КОМАНДА и ИГРОК имеет тип один-ко-многим (1:11), что говорит о том, что в одной футбольной команде на поле может находиться не более 11 игроков.

*Класс принадлежности* – признак, определяющий обязательность участия экземпляров сущности в некоторой связи.

Класс принадлежности *обязательный*, если каждый экземпляр сущности обязан (должен) быть связанным не менее чем с одним экземпляром другой сущности.

Класс принадлежности *необязательный*, если только некоторые экземпляры сущности могут быть связаны с одним или несколькими экземплярами другой сущности, т. е. могут быть экземпляры, которые не связаны ни с одним экземпляром другой сущности.

Например, имеется связь между сущностями СОТРУДНИК и ИНОСТРАННЫЙ ЯЗЫК. Некоторые сотрудники знают иностранный язык. Есть сотрудники, которые не владеют ни одним иностранным языком. Естественно, что имеется много иностранных языков, кото-

рыми не владеет ни один из сотрудников. В данном случае класс принадлежности обеих сущностей необязательный.

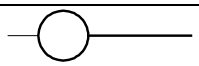
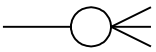
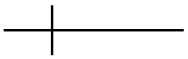
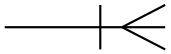
Сущности, между которыми имеется связь, могут иметь разный класс принадлежности.

В табл. 2.2 приведены графические обозначения связей с различными характеристиками.

Класс принадлежности может также обозначаться как указание интервала числа возможных вхождений сущности в связь. Необязательный класс принадлежности в зависимости от мощности связи обозначается следующим образом: 0,1 или 0,M. Для обозначения обязательного класса принадлежности используется обозначение 1,1 или 1,M.

Таблица 2.2

Графические обозначения бинарных связей
с различными характеристиками

| Класс принадлежности | Мощность связи | |
|---|---|---|
| | 1 | M |
| Необязательный |  0,1 |  0,M |
| Обязательный |  1,1 |  1,M |

Примеры бинарных связей с различными мощностями и классами принадлежности приведены на рис. 2.4.

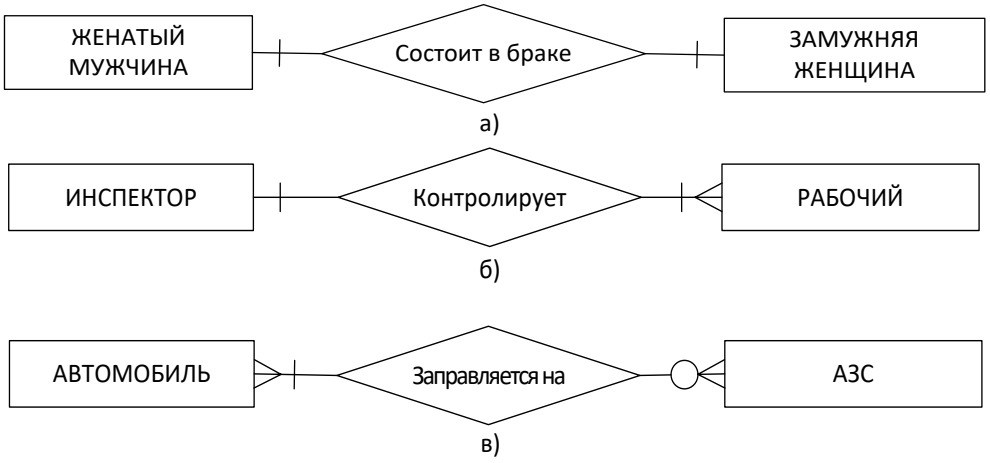



Рис. 2.4. Связи различной мощности

Каждый экземпляр сущности ЖЕНАТЫЙ МУЖЧИНА (см. рис. 2.4, *а*) должен быть обязательно связан с одним экземпляром сущности ЗАМУЖНЯЯ ЖЕНЩИНА, так же как и наоборот, т. е. класс принадлежности обеих сущностей обязательный, а мощность связи – один-к-одному (1:1).

Предполагается, что один ИНСПЕКТОР может контролировать несколько РАБОЧИХ (см. рис. 2.4, *б*), а каждого РАБОЧЕГО контролирует только один ИНСПЕКТОР. Такая связь будет иметь мощность один-ко-многим (1:M). Класс принадлежности с обеих сторон обязательный, так как все ИНСПЕКТОРА должны заниматься контролем, а все РАБОЧИЕ должны контролироваться.

В случае сущностей АВТОМОБИЛЬ и АЗС (автозаправочная станция) (см. рис. 2.4, *в*) мощность связи многие-ко-многим (M:M). Каждый АВТОМОБИЛЬ может заправляться на разных АЗС, а на одной АЗС могут заправляться разные АВТОМОБИЛИ. Класс принадлежности со стороны сущности АЗС необязательный, так как АЗС может быть закрыта на ремонт, следовательно, на этой АЗС не может заправиться ни один автомобиль, т. е. с этой АЗС не будет связан ни один экземпляр сущности АВТОМОБИЛЬ.

Между двумя сущностями может быть задано сколько угодно связей с разными смысловыми нагрузками. Например, между двумя сущностями СТУДЕНТ и ПРЕПОДАВАТЕЛЬ можно установить две связи, одна называется «Руководит ВКР», а вторая – «Читает лекции» (рис. 2.5).

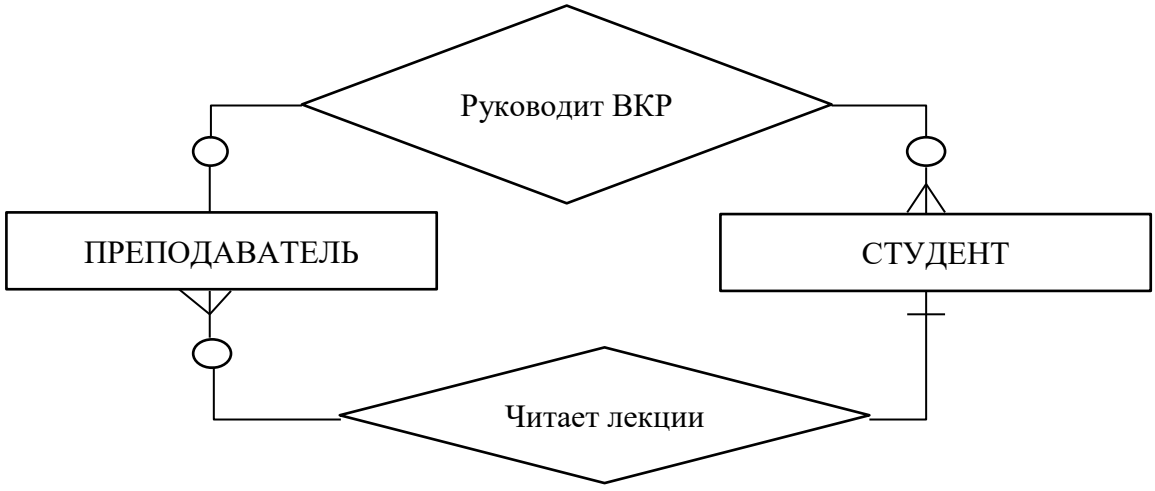



Рис. 2.5. Связи с разными смысловыми нагрузками

Каждый студент имеет только одного руководителя, но один и тот же преподаватель может руководить ВКР нескольких студентов. Поэтому связь «Руководит ВКР» будет типа один-ко-многим (1:M), один – со стороны ПРЕПОДАВАТЕЛЬ и многие – со стороны СТУДЕНТ. Класс принадлежности обеих сущностей необязательный, так как не каждый преподаватель руководит ВКР, а ВКР выполняют только студенты выпускного курса. Связь «Читает лекции» определяет, лекции каких преподавателей слушает данный студент и каким студентам данный преподаватель читает лекции. Ясно, что это связь типа многие-ко-многим. Класс принадлежности сущности СТУДЕНТ обязательный, так как все студенты слушают лекции разных преподавателей по различным предметам. Но не каждый преподаватель может читать лекции.

В табл. 2.3 приведен словарь данных с описаниями типов связей.

Таблица 2.3

Словарь данных с описаниями типов связей

| Связь | Сущности | Мощность связи | Класс принадлежности |
|---|---|---|---|
| Руководит ВКР | ПРЕПОДАВАТЕЛЬ<br>СТУДЕНТ | 1:M | Необязательный<br>Необязательный |
| Читает лекции | ПРЕПОДАВАТЕЛЬ<br>СТУДЕНТ | 1:M | Необязательный<br>Обязательный |

## 2.2. Категориальная сущность и иерархия наследования

Каждая сущность имеет набор уникальных атрибутов. Однако атрибуты разных сущностей могут повторяться. Повторение атрибутов несет избыточность в базу данных, и размер базы данных становится необоснованно большим. Поэтому нужно так разработать диаграмму «сущность – связь», чтобы количество повторяемых атрибутов в разных сущностях было минимальным или сведено к нулю. Для решения данной проблемы используется иерархия наследования.

Идея использования иерархии наследования состоит в том, что для всего разнообразного набора сущностей выделяется обобщающая сущность (родовой предок), которая содержит общую для всех сущностей информацию. Детали каждой сущности выносятся по отдельности в несколько категориальных сущностей.

Рассмотрим пример из ресурса [9]. Пусть нужно разработать базу данных сотрудников учебного заведения. В учебном заведении выделяют три сущности, каждая из которых представляет профессиональную группу сотрудников:

- сущность АДМИНИСТРАЦИЯ;
- сущность ПРЕПОДАВАТЕЛЬ;
- сущность ВСПОМОГАТЕЛЬНЫЙ_ПЕРСОНАЛ.

Если для каждой сущности описать собственный набор атрибутов, то можно заметить, что некоторые атрибуты в разных сущностях будут повторяться.

Нижеследующие атрибуты общие для всех сущностей: «Код сотрудника», «Фамилия», «Имя», «Отчество», «Название должности».

Также можно выделить некоторые уникальные атрибуты:

- сущность АДМИНИСТРАЦИЯ имеет атрибут «Административная ставка»;
- сущность ПРЕПОДАВАТЕЛЬ имеет атрибуты «Ученая степень», «Ученое звание»;
- сущность ВСПОМОГАТЕЛЬНЫЙ_ПЕРСОНАЛ имеет атрибут «Коэффициент выходного дня» (если сотрудник работал по выходным дням).

Чтобы решить проблему дублирования данных, в модель «сущность – связь» вносят изменения, как показано на рис. 2.6, а именно:

- вводят обобщающую сущность СОТРУДНИК, которая содержит общие атрибуты для всех категориальных сущностей;
- вводят категориальные сущности АДМИНИСТРАЦИЯ, ПРЕПОДАВАТЕЛЬ, ВСПОМОГАТЕЛЬНЫЙ_ПЕРСОНАЛ. Каждая из этих сущностей имеет свои уникальные атрибуты.

Между обобщающей и категориальными сущностями существует связь 1:1.

Категориальные сущности наследуют не только атрибуты, но и все связи обобщающей сущности.

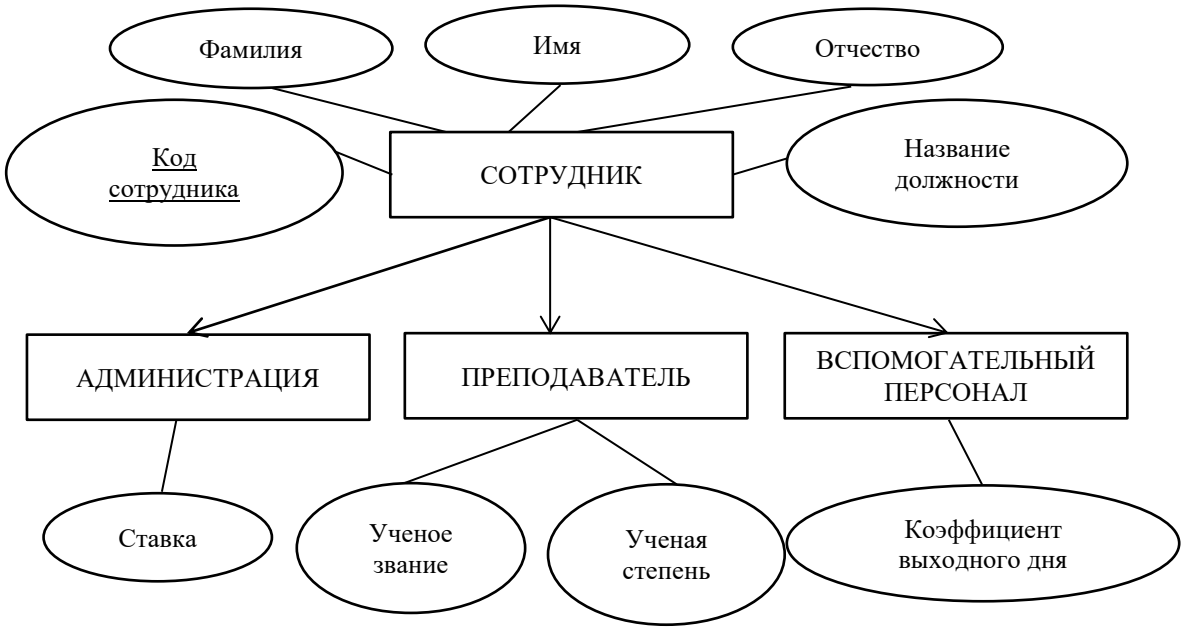



Рис. 2.6. Иерархия наследования на диаграмме «сущность – связь»

## 2.3. Атрибуты связи

Связь, соединяющую сущности, можно понимать как сущность особого типа [1]. Поэтому связь также может иметь атрибуты, аналогичные атрибутам сущности. И если с атрибутами сущностей все очевидно и понятно: они отражают свойства информационных объектов, то с атрибутами связей такой ясности нет. Непонятно, в каких случаях имеет смысл их использовать и как такой атрибут представить в словаре данных.

Попытаемся разобраться с атрибутами связи на конкретном примере.

Рассмотрим диаграмму «сущность – связь» фрагмента базы данных некоторого предприятия, в котором рассчитывается оклад каждого сотрудника. Оклад зависит от занимаемой им должности (инженер, ведущий инженер, бухгалтер, уборщик и т. д.). На предприятии допускается совместительство должностей, т. е. каждый сотрудник может иметь более чем одну должность, причем может занимать неполную ставку. В то же время одну и ту же должность могут занимать одновременно несколько сотрудников.

Очевидно, что в рассматриваемом примере диаграмма «сущность – связь» должна содержать сущности СОТРУДНИК, ДОЛЖНОСТЬ и связь «Занимает» между ними (рис. 2.7).

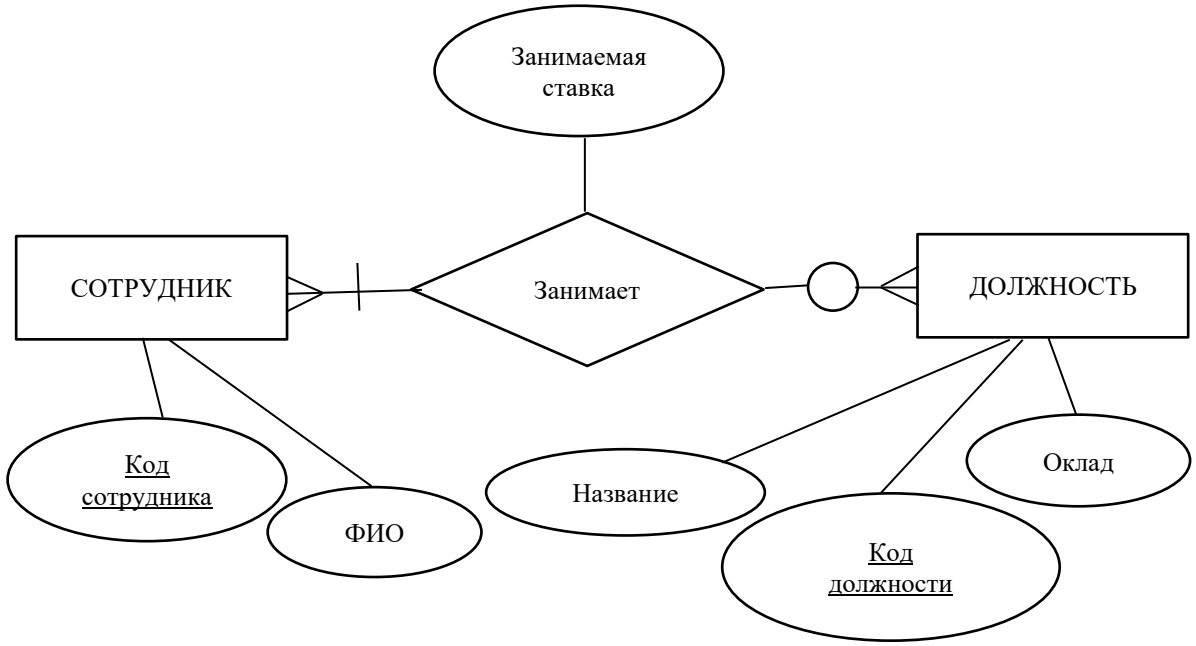



Рис. 2.7. Пример диаграммы «сущность – связь» с атрибутом связи

Сотрудник может занимать несколько должностей, а на одной должности могут работать несколько сотрудников. Поэтому связь между сущностями СОТРУДНИК и ДОЛЖНОСТЬ относится к типу M:M. Каждый сотрудник должен занимать какую-либо должность. Следовательно, класс принадлежности сущности СОТРУДНИК обязательный. Необязательный класс принадлежности сущности ДОЛЖНОСТЬ означает, что какие-то должности могут быть в данный момент времени не заняты, т. е. вакантны. Так как сотрудник может работать на разных должностях, то ставки, которые он занимает по разным должностям, могут быть разными. Ставка может принимать значения из интервала больше нуля, но меньше или равного единице; она определяет, какую часть должностного оклада получает данный сотрудник.

Нетрудно заметить, что значение атрибута «Занимаемая ставка» зависит и от сотрудника, и от должности, которую он занимает. То есть нельзя определить значение атрибута «Занимаемая ставка» только по сотруднику или только по должности. Поэтому атрибут «Занимаемая ставка» – это атрибут связи между сущностями СОТРУДНИК и ДОЛЖНОСТЬ.

Изменим в рассматриваемом примере условие задачи: запретим сотрудникам совместительство, т. е. сотрудник может работать толь-

ко на одной должности. При этом, как и в предыдущем примере, он может занимать неполную ставку. Диаграмма «сущность – связь» для этого случая представлена на рис. 2.8.

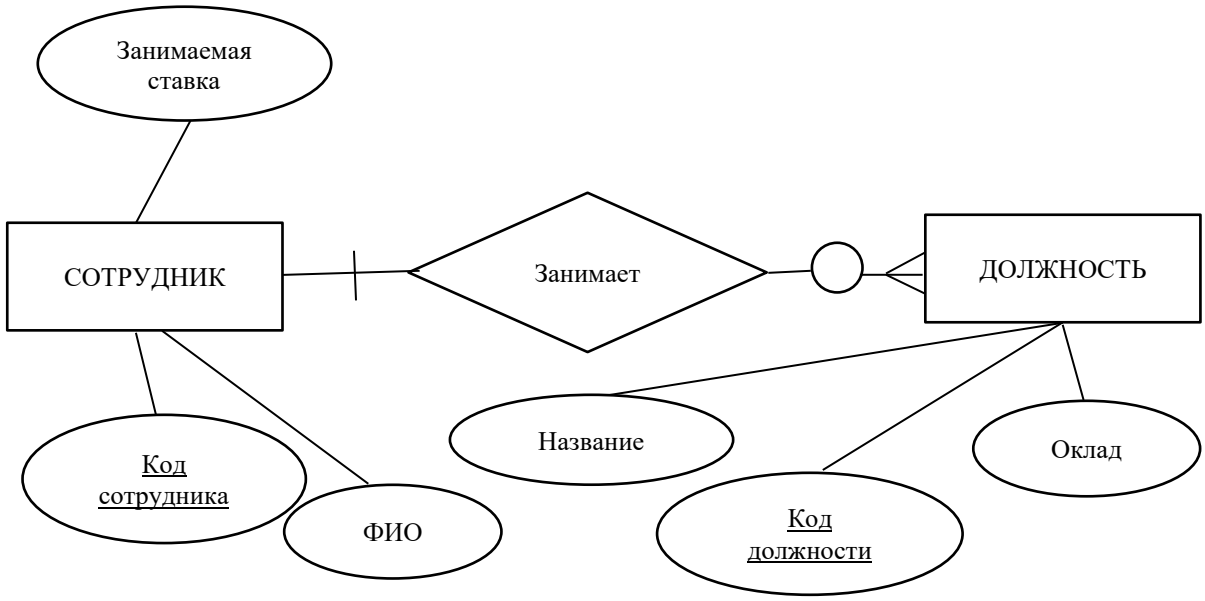



Рис. 2.8. Диаграмма «сущность – связь» без атрибута связи

Как видно по рис. 2.8, мощность связи теперь стала типа 1:M (сотрудник занимает только одну должность) и отсутствует атрибут, принадлежащий связи «Занимает». Последнее объясняется тем, что значение атрибута «Занимаемая ставка» не зависит от должности и этот атрибут характеризует сущность СОТРУДНИК.

Таким образом, можно сделать вывод, что придавать связям свойства, т. е. вводить атрибуты, имеет смысл только в том случае, когда значение этих свойств зависит от обеих сущностей, участвующих в связи. В противном случае этот атрибут можно отнести к одной из сущностей.

Разберемся теперь, к какому объекту относится атрибут связи в словаре данных.

Известно, что любая связь типа многие-ко-многим может быть преобразована в две связи типа один-ко-многим (рис. 2.9). На рис. 2.9 появилась сущность СТАВКА, которая служит для связи между сущностями СОТРУДНИК и ДОЛЖНОСТЬ, и атрибут «Занимаемая ставка» является атрибутом этой сущности.

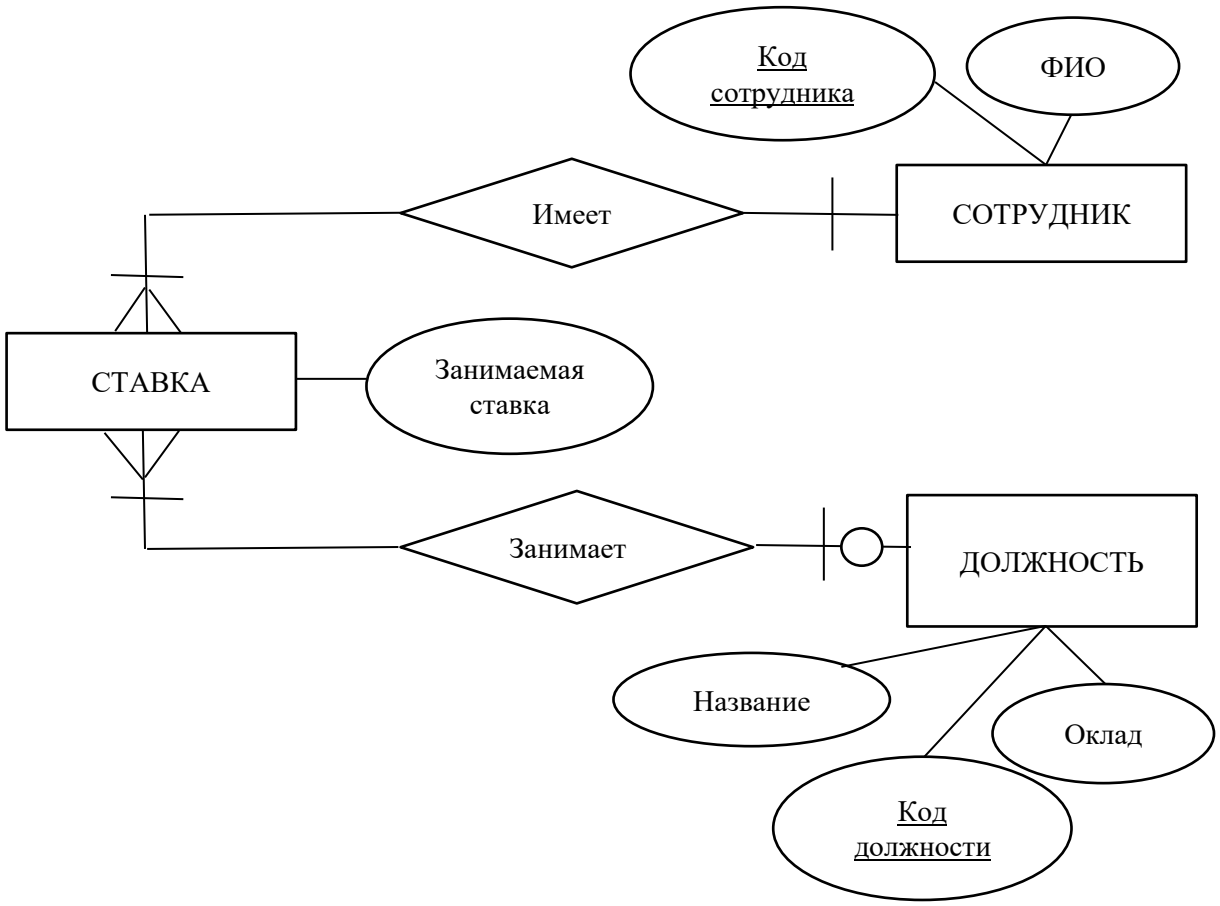



Рис. 2.9. Вариант модели «сущность – связь» с двумя связями 1:М

Словарь данных с описаниями атрибутов сущностей СОТРУДНИК, ДОЛЖНОСТЬ и СТАВКА представлен в табл. 2.4.

Таблица 2.4

Словарь данных с описаниями атрибутов сущностей СОТРУДНИК, ДОЛЖНОСТЬ и СТАВКА

| Имя сущности | Атрибуты | Описание | Ключ | Тип данных и длина | Пустые значения |
|---|---|---|---|---|---|
| СОТРУДНИК | Код сотрудника | Однозначно определяет сотрудника | Да | Целое число | Нет |
| | ФИО | Фамилия, имя и отчество сотрудника | Нет | Строка длиной до 50 символов | Нет |
| | | | | | |

*Окончание табл. 2.4*

| Имя сущности | Атрибуты | Описание | Ключ | Тип данных и длина | Пустые значения |
|---|---|---|---|---|---|
| ДОЛЖНОСТЬ | Код должности | Однозначно определяет должность | Да | Целое число | Нет |
| | Название | Название должности | Нет | Строка длиной до 15 символов | Нет |
| | Оклад | Оклад должности | Нет | Целое число | Нет |
| СТАВКА | Занимаемая ставка | Размер ставки, которую занимает сотрудник по должности | Нет | Вещественное число | Нет |

В табл. 2.5 приведен словарь данных с описаниями типов связей модели на рис. 2.9.

Таблица 2.5

Словарь данных с описаниями типов связей

| Связь | Сущности | Мощность связи | Класс принадлежности |
|---|---|---|---|
| Имеет | СОТРУДНИК<br>СТАВКА | 1:M | Обязательный<br>Обязательный |
| Занимает | ДОЛЖНОСТЬ<br>СТАВКА | 1:M | Необязательный<br>Обязательный |

## 2.4. Рекурсивная связь

Рекурсивная связь связывает сущность саму с собой. Такая связь относится к унарной связи. На рис. 2.10 представлен пример рекурсивной связи, которая показывает, как сотрудник компании контролирует других сотрудников этой компании.

В этом примере связь мощностью один-ко-многим означает, что один сотрудник (например, бригадир) контролирует несколько других сотрудников.

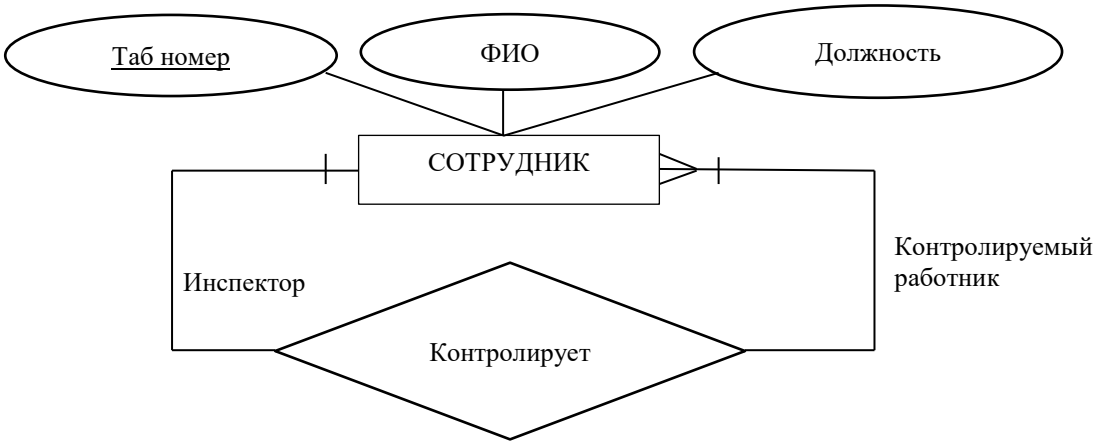



Рис. 2.10. Рекурсивная связь

Связям могут присваиваться *ролевые имена* для указания назначения каждой сущности, участвующей в данной связи.

Первый участник связи «Контролирует» с сущностью СОТРУДНИК получил имя роли «Инспектор», а второй получил имя роли «Контролируемый сотрудник».

## 2.5. Сложные связи

Связь со степенью больше двух называют сложной.

В рассмотренных выше примерах использовались унарная и бинарные связи. Однако модель «сущность – связь» допускает применение связей более высокого порядка, чем бинарные: тернарных, кватернарных и т. д. В этом есть определенная логика, ведь в реальном мире объединение одной связью, скажем, трех сущностей далеко не редкость.

В качестве примера рассмотрим базу данных, в которой фиксируются результаты экзаменационной сессии. Во время экзаменационной сессии студент сдает экзамен по предмету, преподаватель принимает экзамен по этому предмету и выставляет студенту оценку. Следовательно, для моделирования сдачи экзамена потребуется три сущности: СТУДЕНТ, ПРЕДМЕТ и ПРЕПОДАВАТЕЛЬ. Все эти сущности связаны между собой, как показано на рис. 2.11. Такая связь называется тернарной.

При определении мощности связи и класса принадлежности сущности учитывалось, что в течение сессии:

- все студенты сдают несколько экзаменов;
- экзамен по каждому предмету сдают все студенты;
- преподаватель принимает экзамен у многих студентов;
- преподаватель может принимать экзамен по нескольким предметам;
- не все преподаватели принимают экзамены;
- не по всем предметам учебного плана проводятся экзамены.

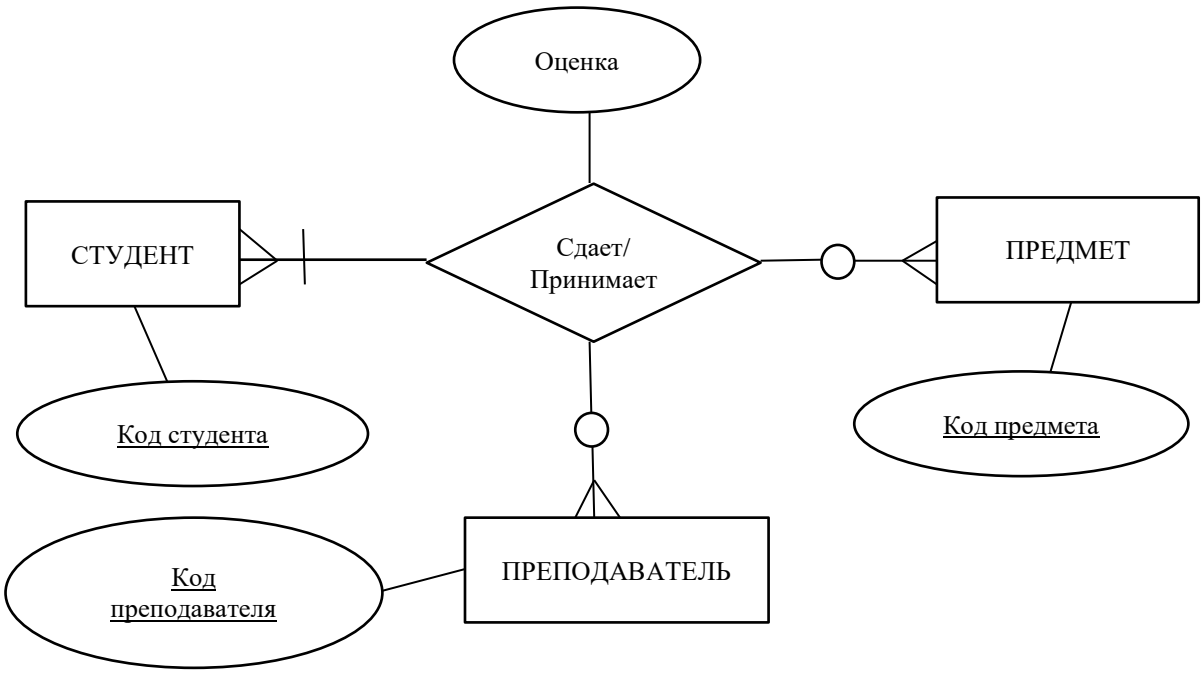



Рис. 2.11. Тернарная связь

Сложная связь «Сдает/Принимает» содержит атрибут «Оценка», так как его значение зависит от сущностей СТУДЕНТ, ПРЕДМЕТ и ПРЕПОДАВАТЕЛЬ.

Следует отметить, что любую сложную связь можно преобразовать в набор связей типа один-ко-многим [11]. Для этого в модель «сущность – связь» вводится новая сущность – соединяющая сущность, атрибутами которой являются ключи соединяемых сущностей. Далее эта новая сущность соединяется со всеми сущностями, участвующими в сложной связи, связью типа один-ко-многим. Обращаем внимание на то, что соединяющая сущность, кроме ключей сущностей, может иметь и другие атрибуты. Например, тернарную связь, представленную на рис. 2.11, можно изобразить следующим образом (рис. 2.12)

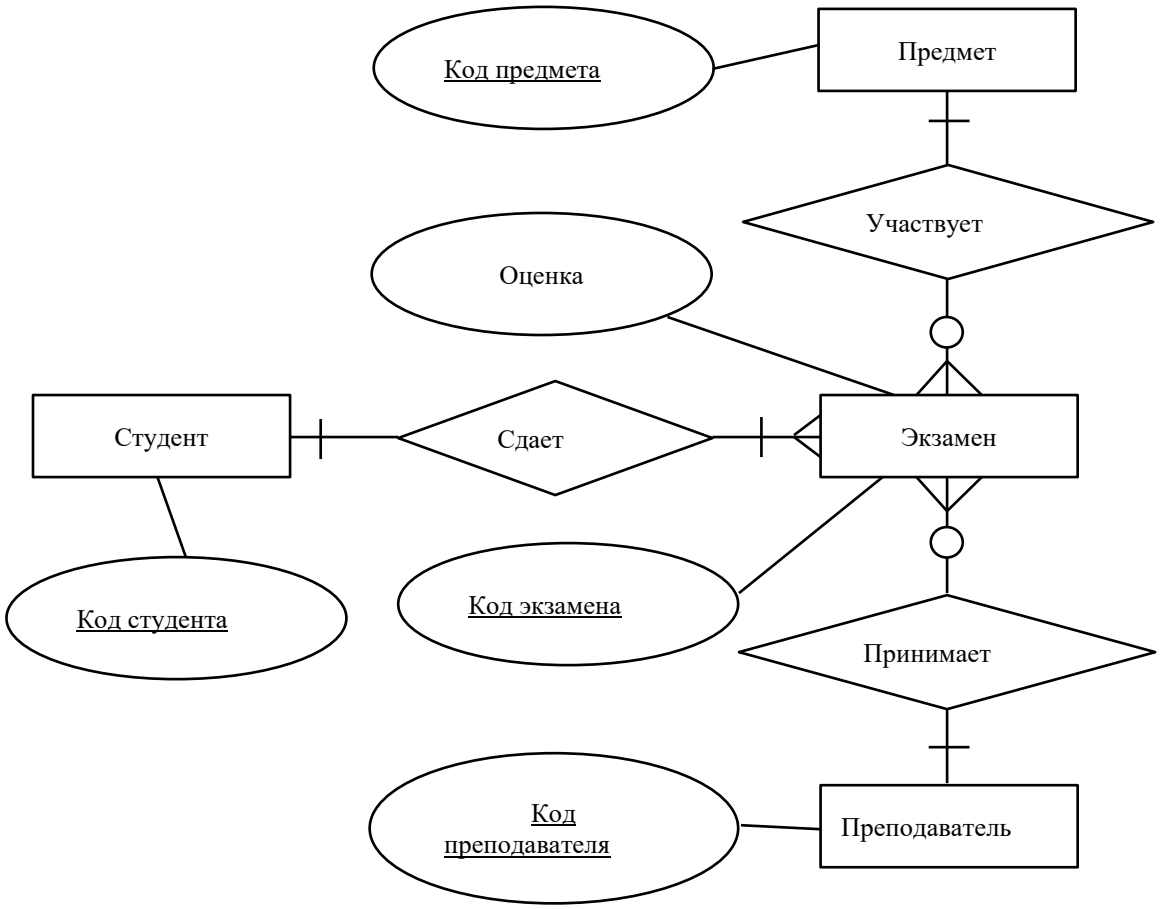



Рис. 2.12. Преобразованная модель тернарной связи на рис. 2.11

## 2.6. Проблемы инфологического моделирования

В процессе создания диаграмм «сущность – связь» могут возникать нежелательные ситуации, которые принято называть *ловушками соединения.* Эти проблемы возникают из-за неправильного толкования смысла некоторых связей между сущностями. Очень важно своевременно выявлять в модели данные ловушки соединения, иначе они могут привести к неадекватному описанию предметной области и необходимости перестройки всей концептуальной модели.

Наиболее распространены два вида ловушек соединения:

- ловушки типа «разветвление»;
- ловушки типа «разрыв».

Ловушка типа «разветвление» имеет место в том случае, когда модель отображает связь между сущностями, но путь между отдельными сущностями этого типа определен неоднозначно. Это происхо-

дит, когда две или несколько связей типа один-ко-многим (1:M) исходят (разветвляются) из одной сущности.

Потенциальная ловушка разветвления показана на рис. 2.13, где две связи типа 1:M выходят из одной сущности ИНСТИТУТ. Проанализировав модель, можно сделать вывод, что в одном институте ведется обучение по нескольким направлениям подготовки и в институте учится много студентов.

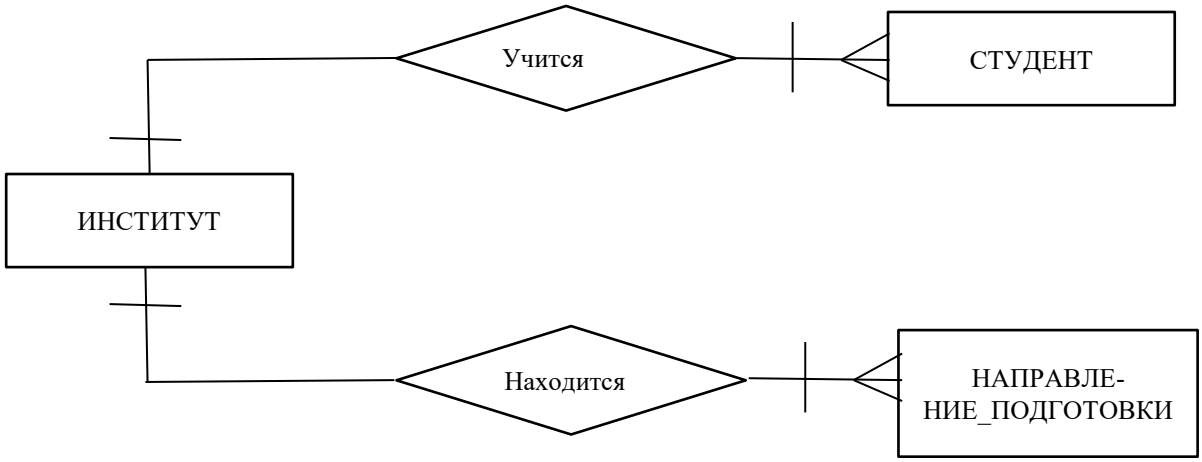



Рис. 2.13. Пример ловушки типа «разветвление»

Проблема может возникнуть при попытке выяснить, по какому направлению подготовки обучается каждый из студентов института.

На рис. 2.14 приведен пример, того, как связаны экземпляры сущностей ИНСТИТУТ, СТУДЕНТ и НАПРАВЛЕНИЕ_ПОДГОТОВКИ.

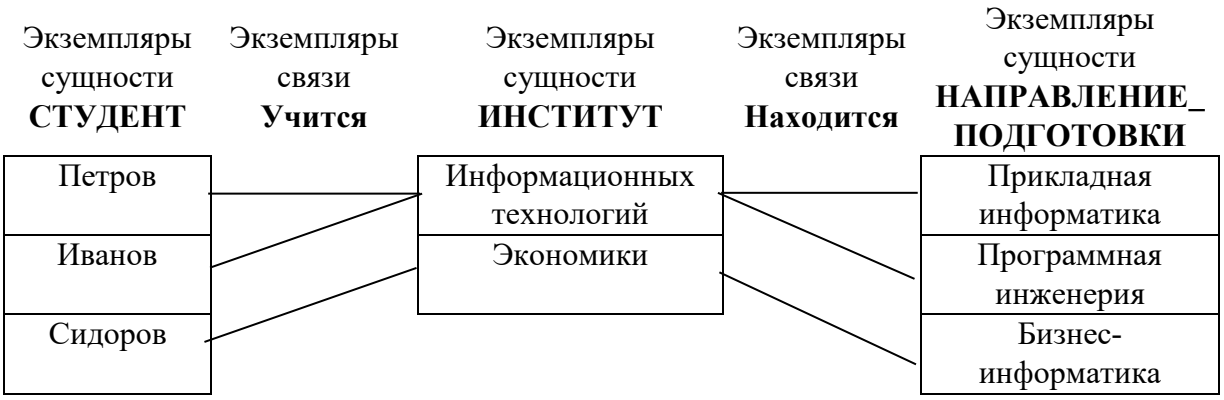



Рис. 2.14. Связь экземпляров сущностей

С помощью рис. 2.14 на конкретном примере невозможно дать однозначный ответ на вопрос: «По какому направлению подготовки обучается студент Петров?». Эта неприятность произошла из-за неправильной трактовки связей между сущностями ИНСТИТУТ, СТУДЕНТ и НАПРАВЛЕНИЕ_ПОДГОТОВКИ. Устранить эту проблему можно путем перестройки модели «сущность – связь» для представления правильного взаимодействия этих сущностей (рис. 2.15).

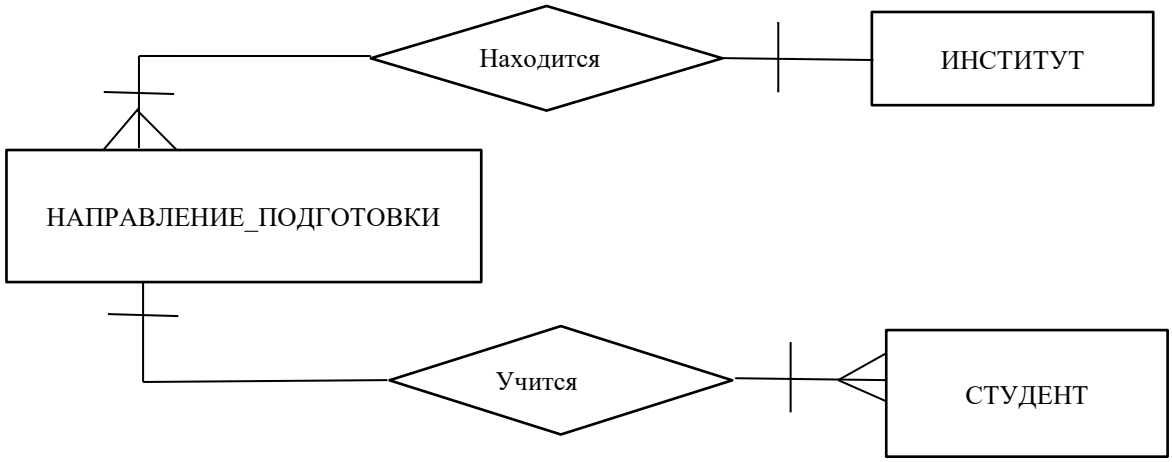



Рис. 2.15. Перестроенная модель «сущность – связь»

Перестройка модели устраняет ловушку типа «разветвление» (рис. 2.16).

Как видно по рис. 2.16, можно однозначно определить, на каком направлении подготовки обучается каждый студент.

| Экземпляры сущности **ИНСТИТУТ** | Экземпляры связи **Находится** | Экземпляры сущности **НАПРАВЛЕНИЕ_ ПОДГОТОВКИ** | Экземпляры связи **Учится** | Экземпляры сущности **СТУДЕНТ** |
|---|---|---|---|---|
| Информационных технологий | | Прикладная информатика | | Петров |
| Экономики | | Программная инженерия | | Иванов |
| | | Бизнес-информатика | | Сидоров |

Рис. 2.16. Перестроенная связь экземпляров сущностей

Ловушка типа «разрыв» появляется в том случае, если в модели предполагается наличие связи между сущностями, но не существует пути между отдельными экземплярами этих сущностей. Это может

иметь место в том случае, если среди сущностей, участвующих в ассоциации, есть сущность с необязательным классом принадлежности. В результате может оказаться, что некоторые экземпляры этой сущности не участвуют в связи и эти связи составляют часть пути между взаимосвязанными сущностями.

Потенциальную ловушку типа «разрыв» покажем на примере диаграммы «сущность – связь» некой компании, продающей объекты недвижимости и имеющей несколько отделений (рис. 2.17).

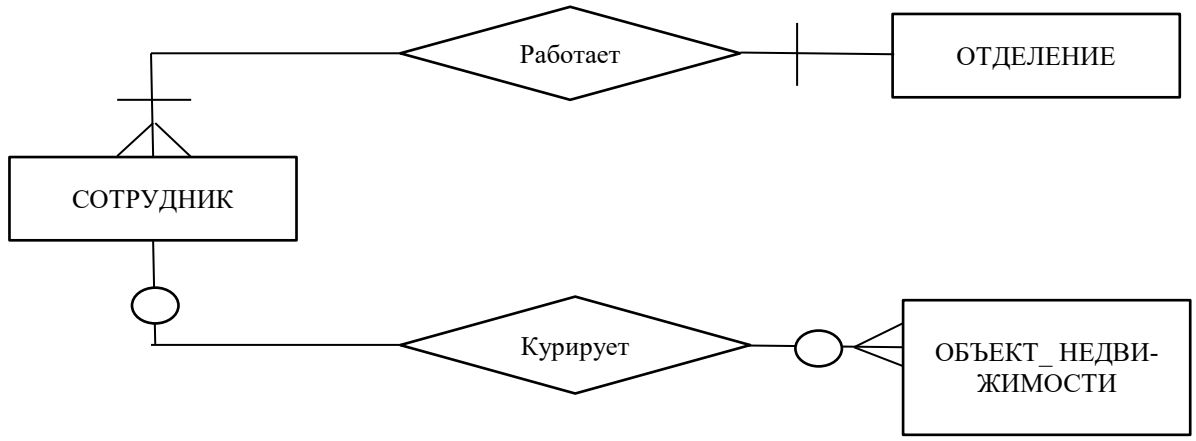



Рис. 2.17. Пример ловушки типа «разрыв»

Класс принадлежности сущностей СОТРУДНИК и ОБЪЕКТ_НЕДВИЖИМОСТИ необязательный, а это означает, что некоторые объекты недвижимости не могут быть связаны с отделением компании с помощью информации о сотрудниках.

Проблемы начинаются при попытках выяснить, какое отделение компании отвечает за работу с объектом под конкретным номером. На рис. 2.18 представлены экземпляры сущностей и связей модели на рис. 2.17.


Рис. 2.18. Связь экземпляров сущностей ОТДЕЛЕНИЕ, СОТРУДНИК и ОБЪЕКТ_НЕДВИЖИМОСТИ

Из рис. 2.18 следует, что за объект с номером А255 никто не отвечает.

Для разрешения этой проблемы следует ввести между сущностями ОТДЕЛЕНИЕ и ОБЪЕКТ_НЕДВИЖИМОСТИ связь «Предлагает» (рис. 2.19).

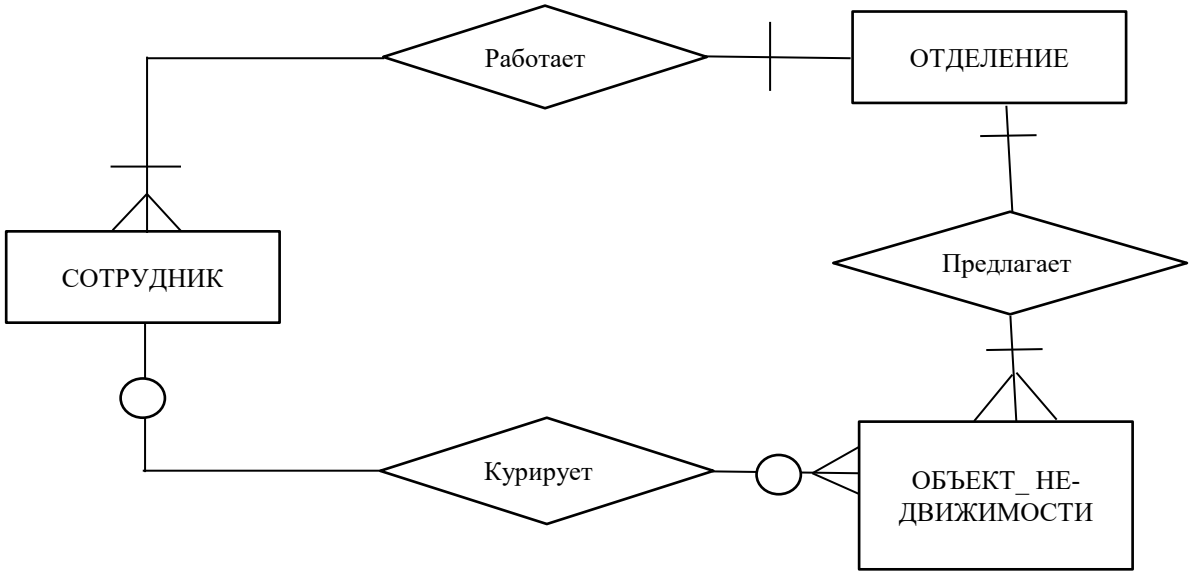



Рис. 2.19. Модель, в которой отсутствует ловушка типа «разрыв»

Добавление связи «Предлагает» устраняет ловушку типа «разрыв» (рис. 2.20).

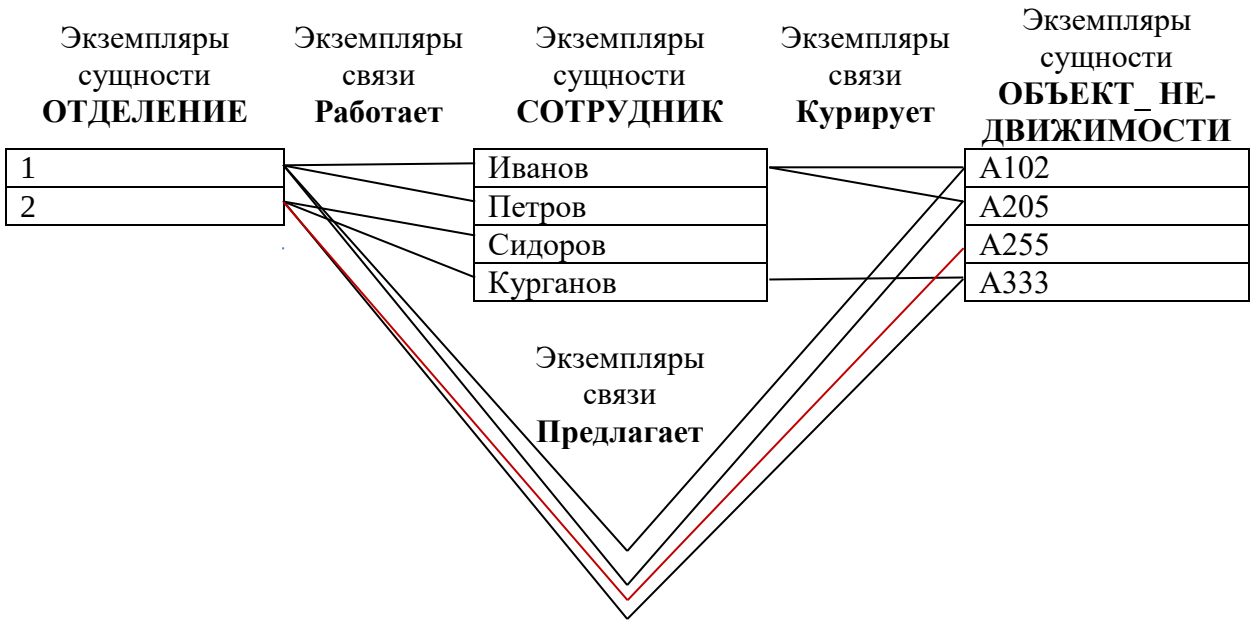



Рис. 2.20. Перестроенная связь экземпляров сущностей ОТДЕЛЕНИЕ, СОТРУДНИК, ОБЪЕКТ_НЕДВИЖИМОСТИ

## 2.7. Построение диаграммы «сущность – связь»

Модель «сущность – связь» – результат инфологического проектирования базы данных. Создается она на основе требований к данным предприятия или организации и состоит из сущностей, связей и атрибутов.

Концептуальное моделирование поддерживается документацией, включая диаграмму «сущность – связь» и словарь данных.

Разработка диаграммы «сущность – связь» представляет собой нисходящий подход к проектированию базы данных, в котором можно выделить несколько очевидных шагов:

1) идентификация представляющих интерес сущностей;
2) идентификация связей;
3) определение атрибутов и связывание их с сущностями и связями;
4) проверка модели на существование ловушек соединения;
5) проверка концептуальной модели относительно реализуемости пользовательских транзакций;
6) обсуждение концептуальной модели с пользователями.

Диаграммы «сущность – связь» удобны тем, что процесс выделения сущностей, атрибутов и связей итерационный. Разработав первый приближенный вариант диаграмм, мы уточняем их, опрашивая экспертов предметной области.

Одна из проблем разработки диаграммы состоит в том, что трудно решить, считать ли конкретный объект сущностью, связью или атрибутом. Например, факт, что конкретная деталь используется при сборке конкретного изделия, можно выразить либо как связь «Входит в состав» между сущностями ДЕТАЛЬ и ИЗДЕЛИЕ, либо как атрибут «Имеет в составе деталь» для сущности ИЗДЕЛИЕ, либо как сущность СХЕМА_СБОРОЧНОГО_СОСТАВА.

В этих случаях рекомендуется проработать несколько вариантов моделей и отобрать более гибкий с точки зрения представления информации. Такой подход повышает возможности совместного использования данных в БД различными пользователями и закладывает основы для обеспечения долгосрочной гибкости системы при удовлетворении информационных потребностей пользователей.

В качестве примера построим диаграмму «сущность – связь» учебной базы данных с именем «Контроль_товара», предназначенной для контроля поступления материалов и товаров на склад промышленного предприятия. Обращаем внимание на то, что в данном примере предприятие имеет только один склад.

Разработку диаграммы начнем с изучения предметной области и процессов, происходящих в ней. Для этого следует опросить сотрудников предприятия, прочитать документацию, изучить формы накладных, счетов-фактур и т. п.

Например, в ходе беседы со специалистами предприятия выяснилось, что проектируемая подсистема должна выполнять следующие действия:

- хранить информацию о поставщиках;
- хранить данные о поступлении на предприятие всех материалов и товаров, причем один и тот же товар может поступать несколько раз;
- хранить информацию о документах на поступившие товары и материалы;
- следить за наличием товаров и материалов на складе.

Поставщики поставляют товары по накладным, в которые внесены данные о количестве и цене поставленного товара. Каждый поставщик может много раз поставлять товары по разным накладным. В каждой накладной указывается только один поставщик. Каждая накладная может содержать несколько товаров (не бывает пустых накладных). Каждый товар, в свою очередь, может быть поставлен несколькими поставщиками по нескольким накладным.

Кроме того, выяснилось, что каждый товар имеет некоторую текущую цену, по которой товар используется предприятием в данный момент. Естественно, что эта цена может меняться со временем. Цена одного и того же товара в разных накладных, выписанных в разное время, может быть различной. Таким образом, имеется две цены: цена товара в приходной накладной и текущая цена товара.

Выделим шаги в разработке диаграммы.

**Шаг 1.** Идентификация представляющих интерес сущностей.

Так как необходимо хранить в базе данных информацию о поставщиках, то потребуется сущность ПОСТАВЩИК.

Материалы и товары поступают на предприятие по приходным документам (накладная, счет-фактура). Поэтому, чтобы обеспечить контроль за документами о поступлении на предприятие всех материалов и товаров, введем сущность ДОКУМЕНТ. Наиболее часто в качестве приходного документа используется накладная. По этой причине сущность ДОКУМЕНТ имеет псевдоним НАКЛАДНАЯ.

В приходных документах указываются товары и материалы, поступающие на предприятие, а также их цены. Причем, как выяснилось при общении с сотрудниками предприятия, цена товара в накладной и цена этого же товара, хранящегося на складе, может быть разной. Чтобы учесть этот факт введем две сущности – ТОВАР_В_НАКЛАДНОЙ и ТОВАР_НА_СКЛАДЕ. В дальнейшем для краткости будем называть эти сущности ПРИХОД и СКЛАД соответственно.

Рассмотрим теперь более внимательно информационный объект «поставщик». На практике очень часто возникает необходимость различать национальную принадлежность юридических лиц, с которыми предприятие вступает в договорные отношения. Это связано с тем, что для зарубежных фирм необходимо хранить, например, сведения о валюте, в которой осуществляются расчеты, языке, на котором подписан контракт и т. д. В свою очередь, для отечественных компаний необходимо иметь сведения об их форме собственности (частная или государственная), поскольку от этого может зависеть порядок налогообложения средств, полученных за выполнение работ по контракту. Таким образом, приходим к выводу, что необходимо ввести в рассмотрение еще две категориальные сущности: ЗАРУБЕЖНЫЙ_ПОСТАВЩИК и ОТЕЧЕСТВЕННЫЙ_ПОСТАВЩИК, объединение которых составляет обобщающую сущность ПОСТАВЩИК, содержащую общую для всех сущностей информацию.

Таким образом, концептуальная модель проектируемой базы данных должна содержать следующие сущности: ДОКУМЕНТ, ПРИХОД, СКЛАД, ПОСТАВЩИК, ЗАРУБЕЖНЫЙ_ПОСТАВЩИК и ОТЕЧЕСТВЕННЫЙ_ПОСТАВЩИК.

Выбранное имя и описание сущности помещаются в словарь данных (табл. 2.6). Если это возможно, следует установить и внести в документацию данные об ожидаемом количестве экземпляров каждой сущности. Если сущность известна пользователям под разными именами, все дополнительные имена рекомендуется определить как синонимы или псевдонимы и также занести в словарь данных.

Таблица 2.6

Словарь данных с описаниями сущностей БД «Контроль_товара»

| Имя сущности | Описание | Псевдонимы | Комментарий |
|---|---|---|---|
| ПРИХОД | Общее обозначение для всех поступивших на предприятие товаров и материалов, указанных в накладных | ТОВАР_В_НАКЛАД-НОЙ | Один и тот же товар или материал может много раз поставляться. Причем в разных количествах и по разным ценам |
| ДОКУМЕНТ | Общее обозначение для всех документов, по которым поступают на предприятие товары и материалы | НАКЛАД-НАЯ | Каждый документ обязан содержать несколько товаров (не бывает пустых накладных). В документе указывается только один поставщик. Один и тот же товар может поставляться много раз |
| СКЛАД | Общее обозначение для всех хранящихся на предприятии товаров и материалов | ТОВАР_НА_СКЛАДЕ | Каждый товар хранится на складе в единственном экземпляре |
| ПОСТАВЩИК | Общее обозначение для всех поставщиков товаров и материалов промышленного предприятия | | Каждый поставщик может много раз поставлять товар. Обобщающая сущность |
| ЗАРУБЕЖ-НЫЙ_ПОСТАВ-ЩИК | Общее обозначение для всех зарубежных поставщиков товаров и материалов промышленного предприятия | | Категориальная сущность |
| ОТЕЧЕСТВЕН-НЫЙ_ПОСТАВ-ЩИК | Общее обозначение для всех отечественных поставщиков товаров и материалов промышленного предприятия | | Категориальная сущность |

**Шаг 2.** Идентификация связей.

Идентификация связей предполагает установление ассоциаций между сущностями, а также определение мощности связи и класса принадлежности.

Для того чтобы определить, какие связи установлены между сущностями, следует еще раз проанализировать процесс поступления товаров и материалов на склад предприятия. Как мы уже знаем, товары поступают на предприятие от поставщика вместе с приходными документами. Следовательно, между сущностями ПОСТАВЩИК и ДОКУМЕНТ установлена бинарная связь «Оформляет». Мощность этой связи один-ко-многим (1:M), так как приходный документ может содержать только одного поставщика, а поставщик может много раз поставлять товары. Класс принадлежности обеих сущностей обязательный.

Каждый документ содержит список товаров, следовательно, между сущностями ДОКУМЕНТ и ПРИХОД существует бинарная связь «Содержит» мощностью один-ко-многим.

Между сущностями ПРИХОД и СКЛАД существует бинарная связь «Поступает». Так как на складе товар представлен одной записью, а поступать на склад он может много раз, то мощность связи один-ко-многим (1:M).

Связь между сущностью ПОСТАВЩИК и сущностями ЗАРУБЕЖНЫЙ_ПОСТАВЩИК и ОТЕЧЕСТВЕННЫЙ_ПОСТАВЩИК называют иерархией наследования (см. п. 2.2), так как сущности ЗАРУБЕЖНЫЙ_ПОСТАВЩИК и ОТЕЧЕСТВЕНЫЙ_ПОСТАВЩИК наследуют атрибуты сущности ПОСТАВЩИК. Мощность такой связи один-к-одному.

В табл. 2.7 приведен словарь данных с описаниями типов связей.

Таблица 2.7

Словарь данных с описаниями типов связей БД «Контроль_товара»

| Связь | Сущности | Мощность связи | Класс принадлежности |
|---|---|---|---|
| Оформляет | ПОСТАВЩИК<br>ДОКУМЕНТ | 1:M | Обязательный<br>Обязательный |
| Содержит | ДОКУМЕНТ<br>ПРИХОД | 1:M | Обязательный<br>Обязательный |
| | | | |

*Окончание табл. 2.7*

| Связь | Сущности | Мощность связи | Класс принадлежности |
|---|---|---|---|
| Поступает | ПРИХОД<br>СКЛАД | 1:M | Обязательный<br>Обязательный |
| Иерархическая | ПОСТАВЩИК<br>ЗАРУБЕЖ-НЫЙ_ПОСТАВЩИК | 1:1 | Обязательный<br>Обязательный |
| Иерархическая | ПОСТАВЩИК<br>ОТЕЧЕСТВЕН-НЫЙ_ПОСТАВЩИК | 1:1 | Обязательный<br>Обязательный |

**Шаг 3.** Определение атрибутов и связывание их с сущностями и связями.

Для каждого атрибута нужно записать:

- имя и описание;
- тип данных и длину;
- простой/составной (если он составной, то указать все входящие в него атрибуты);
- является ли многозначным;
- является ли вычисляемым (и как это делается);
- значение по умолчанию (если есть).

Для каждой сущности необходимо определить атрибуты ее ключа. Рекомендации по выбору ключей приведены в п. 2.1.2.2.

Словарь данных с описаниями атрибутов сущностей представлен в табл. 2.8.

Таблица 2.8

Словарь данных с описаниями атрибутов сущностей
БД «Контроль_товара»

| Имя сущности | Атрибуты | Описание | Ключ | Тип данных и длина | Пустые значения | Значение по умолчанию |
|---|---|---|---|---|---|---|
| ПОСТАВЩИК | Код_поставщика | Однозначно определяет поставщика | Да | Целое число | Нет | Нет |
| | Название | Название поставщика | Нет | Строка длиной до 20 символов | Нет | Нет |
| | Регион | Наименование региона | Нет | Строка длиной до 15 символов | Да | Нет |
| | Рейтинг | Рейтинг поставщика. Зависит от количества поставок | Нет | Целое число в диапазоне от 0 до 100 | Нет | 0 |
| ДОКУМЕНТ | Номер_документа | Номер документа. Однозначно определяет документ | Да | Целое число | Нет | Нет |
| | Дата | Дата оформления документа | Нет | Дата | Нет | Текущая дата |
| ПРИХОД | Цена | Цена товара в документе | Нет | Денежный формат | Нет | Нет |
| | Количество | Количество товара в документе | Нет | Целое число | Нет | Нет |
| | Тип | Тип товара | Нет | Строка длиной до 10 символов | Нет | Нет |
| | | | | | | |

*Окончание табл. 2.8*

| Имя сущности | Атрибуты | Описание | Ключ | Тип данных и длина | Пустые значения | Значение по умолчанию |
|---|---|---|---|---|---|---|
| СКЛАД | Номенклатурный_номер | Номенклатурный номер товара | Да | Целое число | Нет | Нет |
| | Наименование | Наименование товара | Нет | Строка длиной до 20 символов | Нет | Нет |
| | Цена | Цена товара на складе | Нет | Денежный формат | Нет | Нет |
| | Количество | Количество товара на складе | Нет | Целое число | Нет | Нет |
| ЗАРУБЕЖНЫЙ_ПОСТАВЩИК | Валюта | Валюта, в которой осуществляются расчеты | Нет | Строка длиной до 10 символов | Нет | Нет |
| | Язык | Иностранный язык, на котором подписан контракт | Нет | Строка длиной до 15 символов | Нет | Нет |
| ОТЕЧЕСТВЕННЫЙ_ПОСТАВЩИК | Форма_собственности | Форма собственности поставщика | Нет | Строка длиной до 20 символов | Нет | Нет |

Обобщая все проведенные выше рассуждения, получим диаграмму «сущность – связь», показанную на рис. 2.21.

Таким образом, концептуальное моделирование представляет собой нисходящий подход к проектированию базы данных, который начинается с выявления наиболее важных данных, называемых сущностями, и связей между данными, которые должны быть представлены в модели. Затем в модель вносятся дополнительные сведения,

например указывается информация о сущностях и связях, называемая атрибутами, а также все ограничения, относящиеся к сущностям, связям и атрибутам.

**Шаг 4.** Проверка модели на существование ловушек соединения. По рис. 2.21 видно, что в разработанной модели «сущность – связь» отсутствуют ловушки соединения.

**Шаг 5.** Проверка концептуальной модели относительно реализуемости пользовательских транзакций.

Цель такой проверки заключается в ответе на вопрос: поддерживает ли модель все операции, необходимые для конкретной системы?

Для этого должна быть предпринята попытка выполнить все необходимые операции вручную с помощью данной модели.

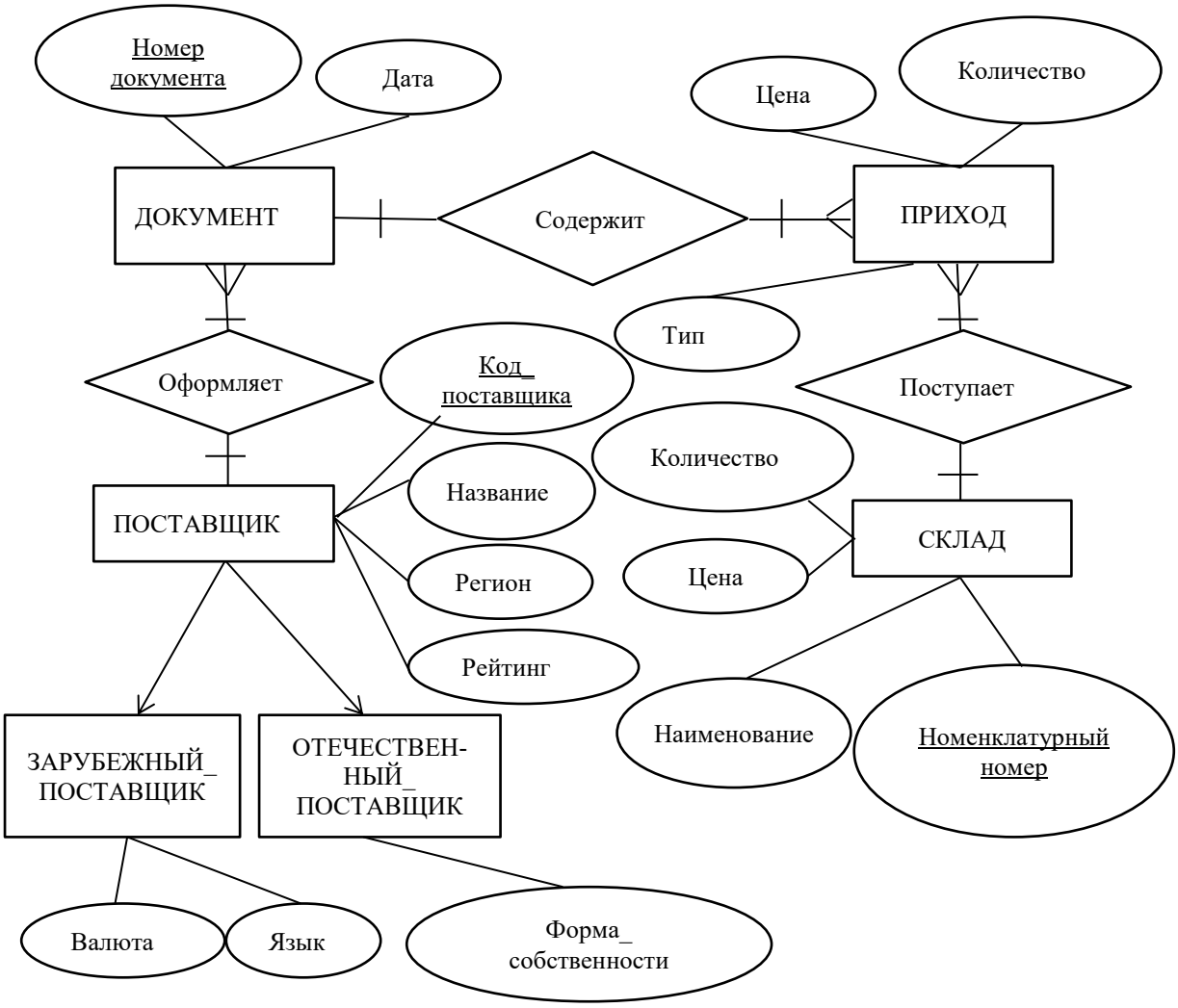



Рис. 2.21. Диаграмма «сущность – связь» БД «Контроль_товара»

Если все операции удалось выполнить, то проверка считается успешной.

Но если невозможно провести вручную все операции, это означает, что модель данных содержит дефекты, которые должны быть устранены. В таком случае, вероятно, в модели данных не учтены какие-либо сущности, связи или атрибуты.

Самый простой способ проверки состоит в том, чтобы определить: предоставляет ли модель всю необходимую информацию для выполнения операции.

В качестве примера рассмотрим запрос: вывести наименование материалов и товаров вместе с датой их поступления на склад. Наименование товара хранится в сущности СКЛАД, а дата поступления товара – в сущности ДОКУМЕНТ (см. рис. 2.21). Причем эти сущности напрямую не связаны друг с другом. Однако сущности СКЛАД и ДОКУМЕНТ соединяются с помощью сущности ПРИХОД. Следовательно, разработанная модель предоставляет всю необходимую информацию для выполнения сформулированного запроса.

Подобным образом проверяются все операции в системе.

**Шаг 6.** Обсуждение концептуальных моделей данных с конечными пользователями.

Этот шаг позволяет выявить несоответствия концептуальной модели спецификациям требований пользовательского представления и в случае необходимости внести в модель соответствующие изменения.

## Вопросы для самопроверки

1. Какова цель инфологического проектирования?
2. Для чего используются диаграммы «сущность – связь»?
3. Какие основные конструктивные элементы используются при построении модели «сущность – связь»?
4. Что означает понятие «сущность»?
5. Какие разновидности сущностей вы знаете?
6. Дайте определение атрибута. Приведите примеры.
7. Какие виды атрибутов вы знаете?
8. Что понимается под понятием «домен атрибута»?
9. Какие правила выбора ключа вы знаете?

10. Что представляет собой словарь данных?
11. Как определяется понятие связи?
12. Что такое степень связи?
13. Что такое мощность связи?
14. Что такое класс принадлежности сущности?
15. Что представляет собой иерархия наследования?
16. Каково назначение рекурсивной связи?
17. Какие связи называются сложными?
18. В каких случаях концептуальная модель может иметь ловушку типа «разветвление»?
19. В каких случаях концептуальная модель может иметь ловушку типа «разрыв»?
20. Какие шаги выделяют при разработке концептуальной модели?

## Практические задания

1. Создайте концептуальную модель базы данных, которая должна содержать сведения об автомобилях (марка, год выпуска, цвет), автосалонах (адрес, телефон) и директоре автосалона (ФИО). Известно, что автосалон продает много автомобилей и автомобиль продается только в одном салоне. У автосалона есть директор, который может быть директором нескольких автосалонов.

2. Шахматисты играют партии в рамках турниров, проводимых организаторами. Создайте концептуальную модель базы данных, которая должна содержать сведения о шахматистах (ФИО, пол, возраст), партиях (игравший белыми, игравший черными, результат игры), турнирах (название, сроки), об организаторах (название, адрес). Известно, что в турнире участвуют два или более шахматиста. Шахматист может участвовать в нескольких турнирах. У турнира может быть много организаторов. Организатор может проводить много турниров.

3. Создайте концептуальную модель данных, которая давала бы ответы на следующие вопросы.

а) Какие товары имеют продажную цену более 200 рублей? Какие из них имеют закупочную цену менее 150 рублей? Какие товары произведены в восточных регионах России? Какие фирмы производят эти товары?

б) Сколько преподавателей работает на факультете информатики и прикладной математики? Их фамилии? Какие курсы они читают?

в) Кто из продавцов оформил наибольшее количество продаж? Даты этих продаж? Каков оклад этих продавцов?

г) Какой процент обладателей текущих счетов банка составляют его служащие? Сколько кассиров и менеджеров имеют в банке сберегательные счета? Сколько кассиров не имеют таких счетов?

4. Нарисуйте концептуальную модель данных.

а) Студенты изучают предметы и получают по ним оценки.

б) Лекции по данному предмету начинаются в определенное время в определенной аудитории.

в) Каждый день служащие работают определенное количество часов.

г) Люди подписываются на журналы. Каждая подписка имеет дату начала и окончания.

д) Летчики имеют определенное число налетов на каждом типе самолета.

# Глава 3. ЛОГИЧЕСКОЕ ПРОЕКТИРОВАНИЕ БАЗ ДАННЫХ

Логическое (даталогическое) проектирование – это отображение концептуальной модели в даталогическую модель данных.

Так как подавляющее большинство современных СУБД реляционные, то в дальнейшем при проектировании баз данных в качестве даталогической модели будем использовать реляционную модель.

Как известно, реляционная модель представляет собой набор таблиц (обычно с указанием ключевых полей) и связей между таблицами. Если концептуальная модель построена в виде диаграммы «сущность – связь», то логическое проектирование состоит в построении таблиц по определённым формализованным правилам, а также в нормализации этих таблиц.

Обращаем внимание, что разрабатываемая логическая модель независима от особенностей реально используемой СУБД и других физических условий.

Логическое проектирование реляционной базы данных состоит из следующих шагов.

Шаг 1. Создание таблиц для логической модели данных.

Шаг 2. Проверка таблиц с помощью правил нормализации.

Шаг 3. Определение ограничений целостности данных.

Шаг 4. Обсуждение разработанной логической модели данных с пользователями.

При построении логической модели необходимо обеспечить:

- возможность хранения в базе данных всех необходимых данных;
- исключение избыточности данных;
- нормализацию таблиц для упрощения решения проблем, связанных с обновлением и удалением данных.

## 3.1. Преобразование концептуальной модели в реляционную

Концептуальная модель «сущность – связь» состоит из сущностей, связей, атрибутов, категориальных сущностей, рекурсивных сущностей и т. д. Рассмотрим методы преобразования каждой из этих конструкций в реляционные таблицы.

### *3.1.1. Преобразование сущностей*

Суть процесса преобразования заключается в следующем: каждой сущности ставится в соответствие реляционная таблица. При этом имена сущности и таблицы не обязательно должны совпадать.

Атрибутами таблицы становятся атрибуты сущности. Переименование атрибутов должно происходить в соответствии с теми же правилами, что и переименование сущностей.

Составные атрибуты сущности разбиваются на простые атрибуты. На рис. 2.2 приведена сущность СТУДЕНТ, которая имеет два составных атрибута: «Имя_студента» и «Адрес», разбитые на простые атрибуты.

Для каждого многозначного атрибута, имеющегося в какой-то сущности, создается новая таблица, представляющая этот атрибут. Новая таблица должна содержать столько атрибутов, сколько значений имеет многозначный атрибут, а также ключ исходной сущности. Он будет служить в качестве внешнего ключа. Например, сущность СТУДЕНТ (см. рис. 2.2) имеет многозначный атрибут «Телефон». Предположим, каждый студент может иметь три номера телефона (домашний, сотовый, рабочий). Тогда реляционная модель сущности СТУДЕНТ будет выглядеть следующим образом:

СТУДЕНТ (Код_студента, Номер_зачетной_книжки, Фамилия, Имя, Отчество, Населенный_пункт, Улица, Номер_дома, Номер_квартиры, День_рождения, Возраст_студента, Номер_группы)

ТЕЛЕФОН (Домашний, Сотовый, Рабочий, Код_студента)

Ключ сущности становится первичным ключом соответствующей таблицы.

Если сущность не имеет ключа, то к таблице добавляется атрибут, значения которого будут однозначно определять строки этой таблицы. Рассмотрим пример модели, приведенный на рис. 3.1.

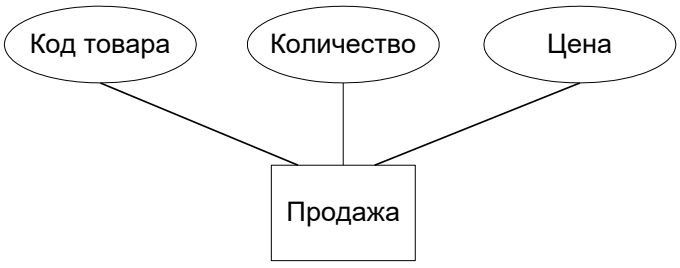



Рис. 3.1. Концептуальная модель продажи

Полученная из этой модели таблица

ПРОДАЖИ (Код_товара, Количество, Цена)

не имеет ключа, так как может быть несколько продаж с одними и теми же значениями кода товара, количества и цены. Следовательно, необходимо добавить новый атрибут, который и будет первичным ключом таблицы. Назовем его «Номер продажи». В результате получим таблицу

ПРОДАЖИ (Номер_продажи, Код_товара, Количество, Цена)

Обращаем внимание, что вне базы данных атрибут «Номер_продажи» не существует. Это искусственно созданный атрибут. Такие ключевые атрибуты называются суррогатными ключами [6]. Суррогатные ключи целесообразно использовать в следующих случаях:

- естественный атрибут должен играть роль ключа, но не обладает необходимым свойством уникальности;
- выбранный в качестве первичного ключа атрибут имеет большую длину (например, длинная строка символов);
- первичный ключ составной и состоит из большого числа атрибутов.

### *3.1.2. Преобразование бинарных связей*

Общие правила преобразования бинарных связей модели «сущность – связь» в реляционные таблицы зависят от мощности связи и класса принадлежности сущностей.

Рассмотрим отдельно бинарные связи один-к-одному, один-ко-многим и многие-ко-многим.

В качестве примера будем рассматривать фрагмент концептуальной модели базы данных для книжного издательства. Этот фрагмент имеет две сущности – АВТОР и КНИГА. Экземплярами сущности АВТОР являются авторы, которые работают с данным издательством. А экземпляры сущности КНИГА представляют собой книги, которые издательство планирует издать.

***3.1.2.1. Связь один-к-одному.*** *Случай 1. Класс принадлежности обеих сущностей обязательный.*

Рассмотрим модель, приведенную на рис. 3.2.

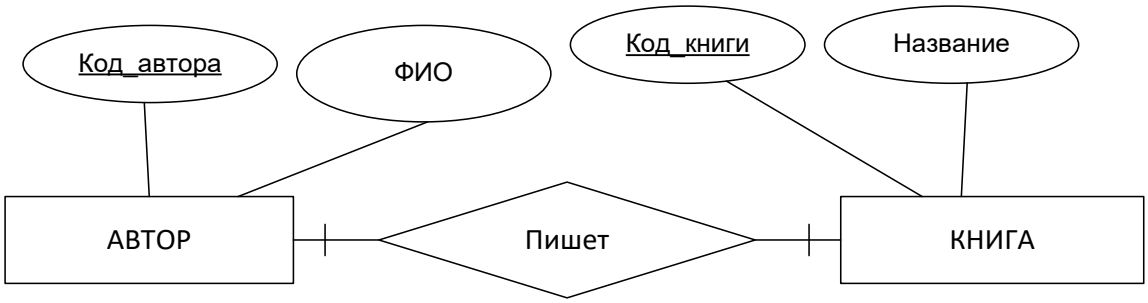



Рис. 3.2. Концептуальная модель со связью один-к-одному с обязательным классом принадлежности обеих сущностей

Из рис. 3.2 следует, что книгу пишет только один автор, т. е. связь имеет мощность один-к-одному. При этом класс принадлежности обеих сущностей обязательный, а ключами сущностей являются атрибуты «Код_автора» и «Код_книги». В этом случае для преобразования следует использовать следующее правило.

**Правило 1.** Если мощность связи один-к-одному и класс принадлежности обеих сущностей обязательный, требуется только одна таблица, первичным ключом которой может быть ключ любой из двух сущностей.

В нашем примере это может быть таблица

АВТОР_КНИГА (Код_автора, ФИО, Код_книги, Название)

В качестве первичного ключа таблицы АВТОР_КНИГА выбран атрибут «Код_автора». Атрибут «Код_книги» – альтернативный ключ.

На рис. 3.3 приведены экземпляры сущностей АВТОР и КНИГА, а также экземпляры связи «Пишет».

Экземпляры сущности АВТОР

| **Код_автора** | **ФИО** |
|---|---|
| 1 | Иванов |
| 2 | Петров |
| 3 | Сидоров |

Экземпляры связи «Пишет»

Экземпляры сущности КНИГА

| **Код_книги** | **Название** |
|---|---|
| 25 | Язык Python |
| 30 | Язык SQL |
| 40 | Язык C++ |

Рис. 3.3. Экземпляры сущностей АВТОР и КНИГА

На рис. 3.4 представлена реляционная таблица АВТОР_КНИГА, полученная в результате преобразования концептуальной модели на рис. 3.2.

| **Код_автора** | **ФИО** | **Код_книги** | **Название** |
|---|---|---|---|
| 1 | Иванов | 30 | Язык SQL |
| 2 | Петров | 40 | Язык C++ |
| 3 | Сидоров | 25 | Язык Python |

Рис. 3.4. Реляционная таблица АВТОР, полученная в результате преобразования концептуальной модели на рис. 3.2

*Случай 2. Класс принадлежности одной сущности – обязательный, а другой – необязательный.*

На рис. 3.5 приведена концептуальная модель, соответствующая этому случаю.

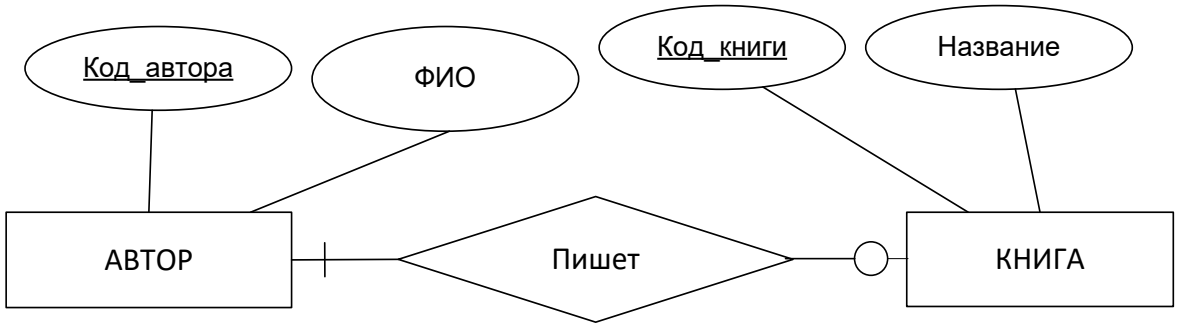



Рис. 3.5. Концептуальная модель со связью один-к-одному с обязательным классом принадлежности одной сущности и необязательным – другой сущности

Необязательный класс принадлежности сущности КНИГА означает, что в настоящий момент в списке издаваемых книг есть такие, которые не имеют пока автора.

Пусть экземпляры сущностей АВТОР и КНИГА и экземпляры связи «Пишет» выглядят, как показано на рис. 3.6.

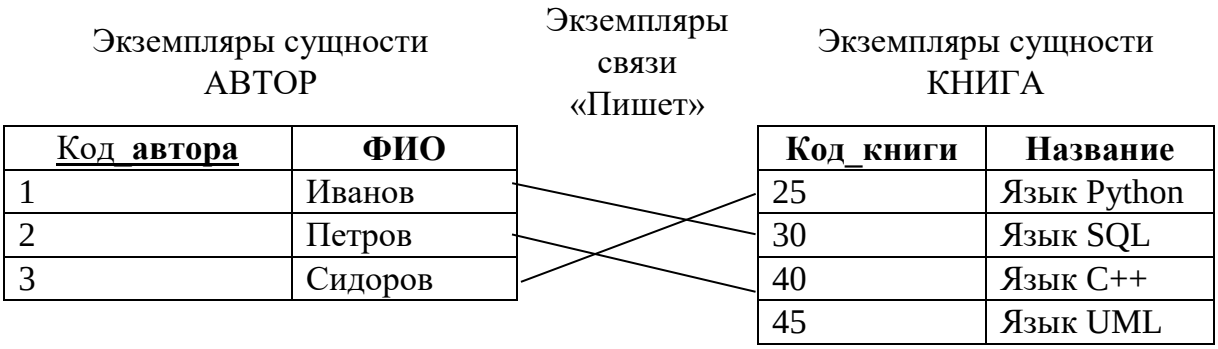



Рис. 3.6. Экземпляры сущностей АВТОР и КНИГА

По рис. 3.6 видно, что книга с кодом 45 не связана ни с каким автором.

В данном случае одной таблицы будет недостаточно, так как в реляционной таблице появляются значения NULL во всех строках, содержащих информацию об авторах, которые не пишут книги (рис. 3.7). Как известно, наличие в реляционных таблицах неопределенных значений NULL может приводить к проблемам при обработке данных.

| **Код_автора** | **ФИО** | **Код_книги** | **Название** |
|---|---|---|---|
| 1 | Иванов | 30 | Язык SQL |
| 2 | Петров | 40 | Язык C++ |
| 3 | Сидоров | 25 | Язык Python |
| NULL | NULL | 45 | Язык UML |

Рис. 3.7. Реляционная таблица, содержащая значения NULL

Способ исключения неопределенных значений состоит в использовании двух таблиц.

**Правило 2.** Если мощность связи один-к-одному и класс принадлежности одной сущности – обязательный, а другой – необязательный, то необходимо построение двух таблиц. Под каждую сущность необходимо выделить одну таблицу. При этом ключ сущности должен служить первичным ключом для соответствующей таблицы. Кроме того, ключ сущности, для которой класс принадлежности необязательный, добавляется в таблицу, выделенную для сущности с обязательным классом принадлежности. В этом случае этот добавленный атрибут будет выполнять роль внешнего ключа.

Согласно правилу 2 реляционная модель для модели, приведенной на рис. 3.5, будет выглядеть следующим образом:

АВТОР (Код_автора, ФИО, Код_книги)
КНИГА (Код_книги, Название)

На рис. 3.8 приведены реляционные таблицы, полученные в результате преобразования концептуальной модели на рис. 3.5.

АВТОР

| **Код_автора** | **ФИО** | **Код_книги** |
|---|---|---|
| 1 | Иванов | 30 |
| 2 | Петров | 40 |
| 3 | Сидоров | 25 |

КНИГА

| **Код_книги** | **Название** |
|---|---|
| 25 | Язык Python |
| 30 | Язык SQL |
| 40 | Язык C++ |
| 45 | Язык UML |

Рис. 3.8. Реляционные таблицы, полученные в результате преобразования концептуальной модели на рис. 3.5

*Случай 3. Класс принадлежности обеих сущностей необязательный.*

На рис. 3.9 представлена концептуальная модель, соответствующая случаю 3.

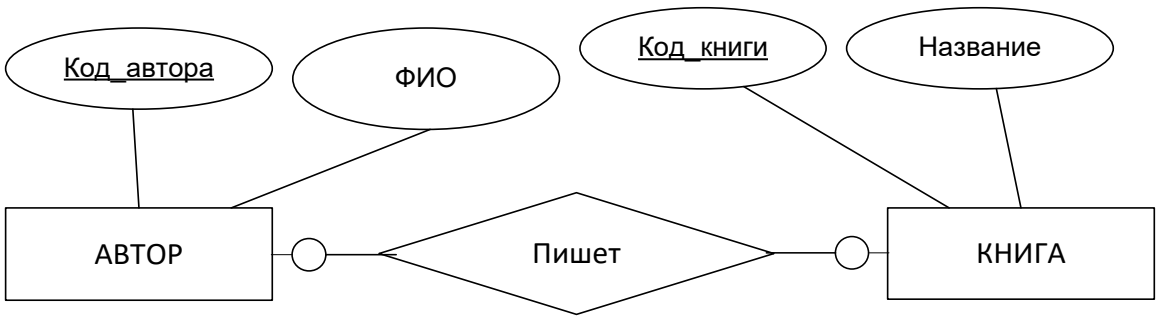
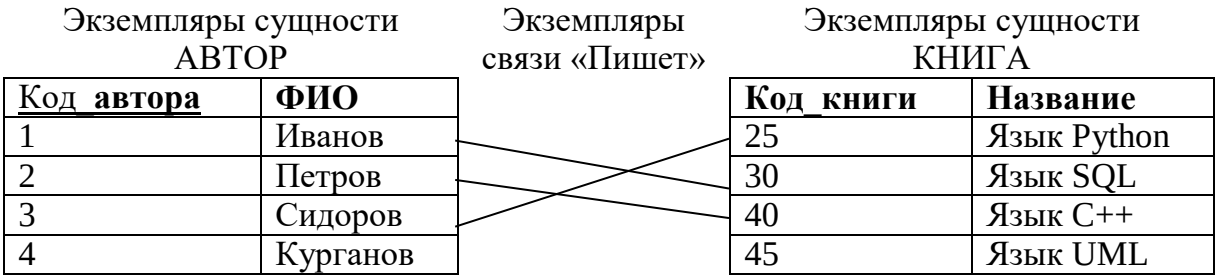



Рис. 3.9. Концептуальная модель со связью один-к-одному и необязательным классом принадлежности обеих сущностей

На рис. 3.10 приведены экземпляры сущностей АВТОР и КНИГА, а также экземпляры связи «Пишет».




| Код_**автора** | **ФИО** |
|---|---|
| 1 | Иванов |
| 2 | Петров |
| 3 | Сидоров |
| 4 | Курганов |

| **Код_книги** | **Название** |
|---|---|
| 25 | Язык Python |
| 30 | Язык SQL |
| 40 | Язык C++ |
| 45 | Язык UML |

Рис. 3.10. Экземпляры сущностей АВТОР, КНИГА и экземпляры связи «Пишет»

В данном случае одной таблицы недостаточно, поскольку возможны два пути возникновения неопределенных значений NULL. Также недостаточно и использования двух таблиц из-за возникновения проблемы с внесением ключа одной сущности в таблицу, выделенную под другую сущность, так как значение ключа может принимать неопределенное значение NULL.

Сформулируем третье правило формирования таблиц.

**Правило 3.** Если мощность связи один-к-одному и класс принадлежности обеих сущностей необязательный, то необходимо использовать три таблицы: по одной для каждой сущности и одну таблицу для связи. Среди своих атрибутов таблица связи будет иметь по одному ключу от каждой сущности. Первичным ключом таблицы связи может быть ключ любой из двух сущностей.

В соответствии с правилом 3 реляционная модель для модели, приведенной на рис. 3.9, будет выглядеть следующим образом:

АВТОР (Код_автора, ФИО)
КНИГА (Код_книги, Название)
АВТОР_КНИГА (Код_автора, Код_книги)

На рис. 3.11 приведены таблицы после преобразования концептуальной модели на рис. 3.9.

АВТОР

| **Код_автора** | **ФИО** |
|---|---|
| 1 | Иванов |
| 2 | Петров |
| 3 | Сидоров |
| 4 | Курганов |

КНИГА

| **Код_книги** | **Название** |
|---|---|
| 25 | Язык Python |
| 30 | Язык SQL |
| 40 | Язык C++ |
| 45 | Язык UML |

АВТОР_КНИГА

| **Код_автора** | **Код_книги** |
|---|---|
| 1 | 30 |
| 2 | 40 |
| 3 | 25 |

Рис. 3.11. Реляционные таблицы, полученные в результате преобразования концептуальной модели на рис. 3.9

Таблица АВТОР содержит информацию обо всех авторах издательства, таблица КНИГА – обо всех издающихся книгах. В таблице АВТОР_КНИГА любые значения, как Код_автора, так и Код_книги, могут появиться только однажды, так как мощность связи один-к-одному. Кроме того, таблица АВТОР_КНИГА содержит код только тех книг, которые пишутся, и код только тех авторов, которые пишут книги в данный момент. Первичный ключ таблицы АВТОР_КНИГА составной, состоит из двух атрибутов – «Код_автора» и «Код_книги». В качестве первичного ключа таблицы АВТОР_КНИГА выбран атрибут «Код_автора».

*Примечание*. В обсуждаемом случае таблица АВТОР_КНИГА не имеет других атрибутов, кроме требуемых ключей. Такая ситуация наблюдается далеко не всегда. Например, если бы каждой книге на самой ранней стадии ее написания назначался редактор, то фамилия последнего была бы значением атрибута «Редактор» таблицы АВТОР_КНИГА.

***3.1.2.2. Связь один-ко-многим.*** Для преобразования связи один-ко-многим требуется два правила. Фактором, определяющим выбор одного из двух правил, является класс принадлежности *n*-связанной сущности. Класс принадлежности 1-связанной сущности в обоих случаях не влияет на выбор правила.

*Случай 1. Класс принадлежности n-связанной сущности обязательный.*

В качестве примера рассмотрим ситуацию, когда автор может писать несколько книг. Модель представлена на рис. 3.12.

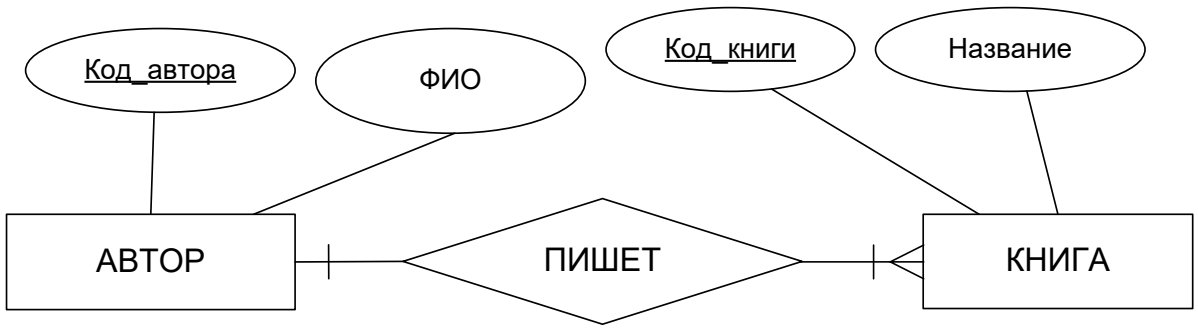
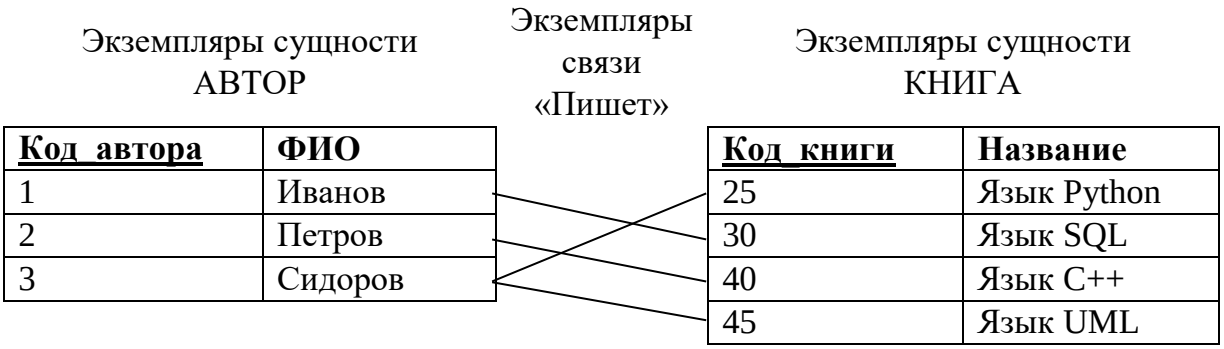



Рис. 3.12. Концептуальная модель со связью один-ко-многим
и обязательным классом принадлежности *n*-связанной сущности

На рис. 3.13 приведены экземпляры сущностей АВТОР и КНИГА, а также экземпляры связи «Пишет».




| **Код_автора** | **ФИО** |
|---|---|
| 1 | Иванов |
| 2 | Петров |
| 3 | Сидоров |

| **Код_книги** | **Название** |
|---|---|
| 25 | Язык Python |
| 30 | Язык SQL |
| 40 | Язык C++ |
| 45 | Язык UML |

Рис. 3.13. Экземпляры сущностей АВТОР, КНИГА и экземпляры связи «Пишет» концептуальной модели на рис. 3.12

**Правило 4.** Если мощность связи один-ко-многим и класс принадлежности *n*-связанной сущности обязательный, то достаточно двух таблиц – по одной на каждую сущность. Помимо этого, ключ 1-связанной сущности должен быть добавлен как атрибут в таблицу, отводимую *n*-связанной сущности.

Если воспользоваться этим правилом, то реляционная модель будет выглядеть следующим образом:

АВТОР (Код_автора, ФИО)
КНИГА (Код_книги, Название, Код_автора)

Таблица АВТОР – родительская таблица, а таблица КНИГА – дочерняя. Атрибут «Код_автора» в таблице КНИГА выполняет роль внешнего ключа.

На рис. 3.14 приведены таблицы после преобразования концептуальной модели (см. рис. 3.12).

АВТОР

| **Код_автора** | **ФИО** |
|---|---|
| 1 | Иванов |
| 2 | Петров |
| 3 | Сидоров |

КНИГА

| **Код_книги** | **Название** | **Код_автора** |
|---|---|---|
| 25 | Язык Python | 3 |
| 30 | Язык SQL | 1 |
| 40 | Язык C++ | 2 |
| 45 | Язык UML | 3 |

Рис. 3.14. Реляционные таблицы, полученные в результате преобразования концептуальной модели, изображенной на рис. 3.12

*Случай 2. Класс принадлежности n-связанной сущности необязательный.*

Концептуальная модель, соответствующая этому случаю, приведена на рис. 3.15.

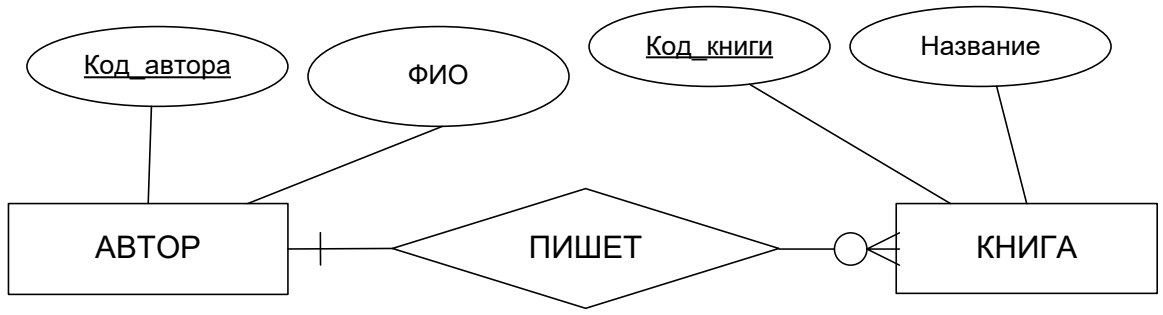



Рис. 3.15. Концептуальная модель со связью один-ко-многим и необязательным классом принадлежности *n*-связанной сущности

В этом случае двух таблиц будет недостаточно, так как в таблице КНИГА столбец «Код_автора» будет содержать неопределенные значения NULL. Поэтому требуется использовать другое правило.

**Правило 5.** Если мощность связи один-ко-многим и класс принадлежности *n*-связанной сущности необязательный, то необходимо формирование трех таблиц: по одной на каждую сущность и одной таблицы связи. Среди своих атрибутов таблица связи должна иметь по одному ключу от каждой сущности. Первичный ключ таблицы связи может быть простым. Им служит атрибут, который является ключом *n*-связанной сущности.

В соответствии с этим правилом реляционная модель будет выглядеть следующим образом:

АВТОР (Код_автора, ФИО)

КНИГА (Код_книги, Название)

АВТОР_КНИГА (Код_автора, Код_книги)

***3.1.2.3. Связь многие-ко-многим.*** Для преобразования связи многие-ко-многим требуется одно правило вне зависимости от класса принадлежности как одной, так и другой сущности.

Концептуальная модель связи многие-ко-многим представлена на рис. 3.16.


Рис. 3.16. Концептуальная модель связи многие-ко-многим

**Правило 6.** Если мощность связи многие-ко-многим, то необходимо формирование трех таблиц: по одной для каждой сущности и одной таблицы связи. Среди своих атрибутов таблица связи должна иметь по одному ключу от каждой сущности. Ключ таблицы связи составной.

Используя это правило для концептуальной модели на рис. 3.16, получим реляционную модель, состоящую из трех таблиц:

АВТОР (Код_автора, ФИО)
КНИГА (Код_книги, Название)
АВТОР_КНИГА (Код_автора, Код_книги)

Таблица АВТОР_КНИГА – таблица связи. Нетрудно заметить, что в результате преобразования одной связи типа многие-ко-многим концептуальной модели было получено две связи типа один-ко-многим реляционной модели. Атрибуты «Код_автора» и «Код_книги» дочерней таблицы АВТОР_КНИГА – внешние ключи родительских таблиц АВТОР и КНИГА. Первичный ключ таблицы связи составной: «Код_автора» + «Код_книги».

*Примечание.* Составной ключ таблицы связи можно заменить простым суррогатным ключом. Например, добавим в таблицу АВТОР_КНИГА дополнительный атрибут с именем «Номер», значениями которого являются целые числа, и объявим его первичным ключом таблицы связи. Реляционная модель в этом случае будет выглядеть таким образом:

АВТОР (Код_автора, ФИО)
КНИГА (Код_книги, Название)
АВТОР_КНИГА (Номер, Код_автора, Код_книги)

Таблица связи, кроме атрибутов, образующих первичный ключ, может содержать и другие атрибуты. Например, схема реляционной базы данных, соответствующая модели «сущность – связь» на рис. 2.7 п. 2.3, будет выглядеть следующим образом:

СОТРУДНИК (Код_сотрудника, ФИО)
ДОЛЖНОСТЬ (Код_должности, Название, Оклад)
СТАВКА (Код_сотрудника, Код_должности, Ставка)

В таблице СТАВКА атрибут «Код_сотрудника» ссылается на таблицу СОТРУДНИК, а атрибут «Код_должности» – на таблицу ДОЛЖНОСТЬ, т. е. эти атрибуты – внешние ключи таблицы СТАВКА.

Обращаем внимание на то, что таблица СТАВКА, помимо атрибутов, предназначенных для связывания таблиц, содержит атрибут «Ставка», который является атрибутом таблицы связи модели (см. рис. 2.7).

Таким образом, наличие достаточно большого количества правил преобразования бинарных связей концептуальной модели в реляционную объясняется желанием проектировщиков исключить из реляционных таблиц неопределенное значение NULL. Присутствие такого значения в таблицах БД приводит к проблемам при обработке данных в них. Если эти проблемы проектировщиков не интересуют, то достаточно будет использовать три правила: 1, 4 и 6. В этом случае в атрибутах внешнего ключа дочерней сущности, у которой класс принадлежности обязательный, допускаются неопределенные значения NULL.

### *3.1.3. Преобразование связи «иерархия наследования»*

Преобразование связи «иерархия наследования» рассмотрим на примере концептуальной модели (см. рис. 2.6), которая содержит обобщающую сущность СОТРУДНИК и три категориальные сущности АДМИНИСТРАЦИЯ, ПРЕПОДАВАТЕЛЬ и ВСПОМОГАТЕЛЬНЫЙ_ПЕРСОНАЛ. При преобразовании будем учитывать, что между сущностью СОТРУДНИК и каждой категориальной сущностью существует связь типа один-к-одному, а класс принадлежности всех сущностей модели на рис. 2.6 обязательный.

Сущность СОТРУДНИК легко преобразуется в одноименную таблицу:

СОТРУДНИК (Код_сотрудника, Фамилия, Имя, Отчество, Название_должности)

Категориальные сущности АДМИНИСТРАЦИЯ, ПРЕПОДАВАТЕЛЬ и ВСПОМОГАТЕЛЬНЫЙ_ПЕРСОНАЛ наследуют атрибуты сущности СОТРУДНИК. Кроме того, у каждой категориальной сущности есть собственные атрибуты. Следовательно, получим три реляционные таблицы:

АДМИНИСТРАЦИЯ (Код_сотрудника, Административная_ставка)

ПРЕПОДАВАТЕЛЬ (Код_сотрудника, Ученая_степень, Ученое_звание)

ВСПОМОГАТЕЛЬНЫЙ_ПЕРСОНАЛ (Код_сотрудника, Коэффициент_выходного_дня)

Внешний ключ «Код_сотрудника» ссылается на таблицу обобщения СОТРУДНИК.

### *3.1.4. Преобразование рекурсивной связи*

Как мы уже знаем, рекурсивная связь связывает сущность саму с собой, а точнее, экземпляр сущности с другими экземплярами той же самой сущности.

Выбор способа преобразования рекурсивной связи зависит от класса принадлежности сущности с обеих сторон связи.

*Случай 1. Обязательный класс принадлежности на обеих сторонах связи.*

**Правило 1.** Требуется одна таблица, содержащая все атрибуты сущности. Первичным ключом таблицы становится ключ сущности. Кроме того, в таблицу добавляется атрибут, который служит для организации связи между экземплярами сущности.

На рис. 2.10 приведена концептуальная модель с обязательным классом принадлежности на обеих сторонах рекурсивной связи.

В соответствии с правилом 1 получим следующую реляционную таблицу:

СОТРУДНИК (Таб_номер, ФИО, Должность, Таб_номер_инспектора)

Атрибут «Таб_номер_инспектора» – это рекурсивный внешний ключ, поскольку он ссылается на атрибут «Таб_номер», т. е. на ключ своей собственной таблицы.

*Случай 2. Необязательный класс принадлежности на одной или обеих сторонах связи.*

**Правило 2.** Потребуется две таблицы: одна таблица соответствует сущности, а другая таблица – это таблица связи, которая содержит две копии ключа сущности, являющиеся внешними ключами. Одну из копий нужно переименовать.

Пусть имеется концептуальная модель, представленная на рис. 3.17.

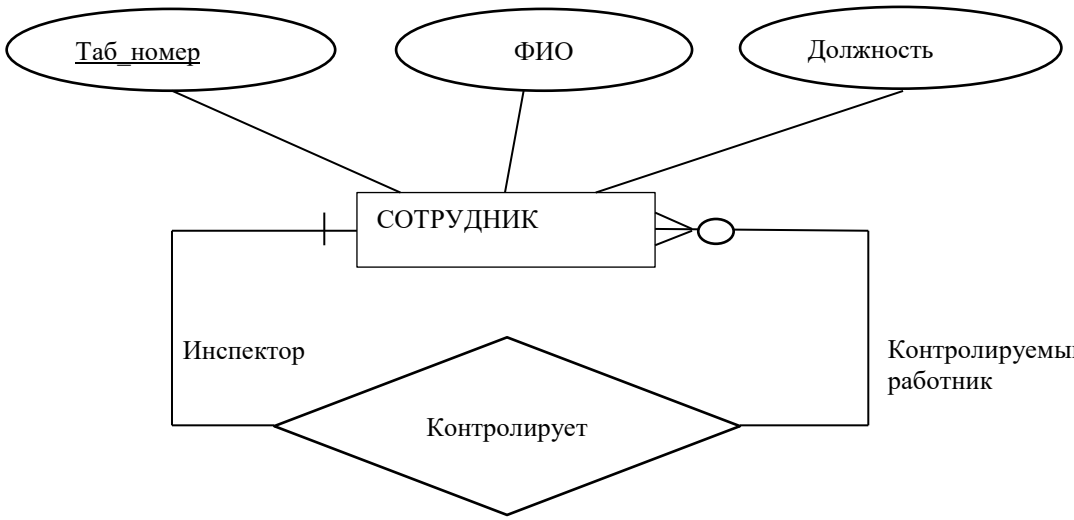



Рис. 3.17. Рекурсивная связь с необязательным классом принадлежности на одной стороне связи

Реляционная модель такой концептуальной модели выглядит следующим образом:

СОТРУДНИК (Таб_номер, ФИО, Должность)
КОНТРОЛЬ (Таб_номер, Таб_номер_инспектора)

### *3.1.5. Преобразование связей, имеющих атрибуты*

Для такого преобразования никаких новых правил не требуется. Достаточно тех, которые уже были приведены выше.

Рассмотрим рис. 2.7, на котором приведена модель «сущность – связь» фрагмента базы данных для расчета оклада каждого сотрудни-

ка некоторого предприятия. Связь «Занимает» в этой модели имеет атрибут «Занимаемая_ставка».

Преобразуем инфологическую модель (см. рис. 2.7) в реляционную в соответствии с рассмотренными выше правилами. Так как связь имеет мощность многие-ко-многим, создадим три таблицы:

СОТРУДНИК (Код_сотрудника, ФИО)

ДОЛЖНОСТЬ (Код_должности, Название, Оклад)

СТАВКА (Код_сотрудника, Код_должности, Занимаемая_ставка).

Таблица СТАВКА – таблица связи. Обращаем внимание на то, что эта таблица содержит не только первичные ключи обеих таблиц, участвующих в связи, но и атрибут «Занимаемая_ставка», который относился к связи «Занимает».

К сказанному можно добавить, что в полученной реляционной модели внешними ключами таблицы СТАВКА являются атрибут «Код_сотрудника», который ссылается на таблицу СОТРУДНИК, и атрибут «Код_должности» для ссылки на таблицу ДОЛЖНОСТЬ.

Кстати, если преобразовать другой вариант инфологической модели рассматриваемого фрагмента базы данных, приведенный на рис. 2.9, то получим тот же самый набор реляционных таблиц.

### *3.1.6. Преобразование сложных типов связи*

Сложные типы связи описаны в п. 2.5. Реляционная модель данных без проблем поддерживает унарные и бинарные связи. При превышении размерности связи любую сложную связь можно разбить на несколько бинарных связей с помощью следующего алгоритма:

- каждой сущности, входящей в сложную связь, ставится в соответствие таблица;
- создается новая таблица – таблица связи, которая содержит в качестве внешних ключей атрибуты ключей сущностей, участвующих в сложной связи. Кроме того, в таблицу связи включаются все атрибуты, входящие в состав рассматриваемой связи;
- все внешние ключи, соответствующие стороне связи «многие», образуют первичный ключ таблицы связи.

На рис. 2.11 приведен пример тернарной связи. Согласно выше приведенному правилу получим следующую реляционную модель:

СТУДЕНТ (Код_студента)
ПРЕПОДАВАТЕЛЬ (Код_преподавателя)
ПРЕДМЕТ (Код_предмета)
ЭКЗАМЕН (Код_студента, Код_преподавателя, Код_предмета, Оценка)

Точно такую же реляционную модель будет иметь концептуальная модель, приведенная на рис. 2.12, полученная путем преобразования тернарной связи в три бинарных связи.

## Вопросы для самопроверки

1. Какие задачи решаются на этапе логического проектирования?
2. Что такое логическая модель базы данных?
3. Как преобразуются сущности концептуальной модели в реляционную?
4. Как преобразуются атрибуты концептуальной модели в реляционную?
5. От каких факторов зависит выбор правила преобразования связи?
6. Как преобразуются связи типа один-ко-многим, изображенные на концептуальной модели, в реляционную модель?
7. Как преобразуются связи типа многие-ко-многим, изображенные на концептуальной модели, в реляционную модель?
8. Как преобразуется связь «иерархия наследования» в реляционную модель?
9. Какие правила преобразования рекурсивных связей вы знаете?
10. Как преобразуются сложные типы связей в реляционную модель?

## Практические задания

Преобразуйте концептуальные модели, созданные в соответствии с практическими заданиями 1 – 3 гл. 2, в реляционные.

## 3.2. Нормализация таблиц реляционной базы данных

После получения логической модели базы данных необходимо убедиться, что полученный набор таблиц адекватно поддерживает требования к данным с точки зрения пользователя. Дело в том, что при неправильно спроектированной схеме БД могут возникнуть проблемы при модификации данных в таблицах. Для устранения таких проблем необходимо провести нормализацию таблиц базы данных.

Общее назначение процесса нормализации заключается в следующем:

- исключение избыточного дублирования данных;
- устранение аномалий ввода, удаления и обновления.

Охарактеризуем основные проблемы, имеющие место при определении структур данных в таблицах реляционной модели.

### *3.2.1. Избыточное дублирование данных*

Различают два вида дублирования данных: простое (неизбыточное) и избыточное. Наличие первого из них допускается в базе данных, а избыточное дублирование данных может приводить к проблемам при обработке данных.

Какое дублирование данных является простым, а какое – избыточным? В качестве примера рассмотрим таблицу ДИСЦИПЛИНА_ПРЕПОДАВАТЕЛЬ (Дисциплина, Преподаватель). В табл. 3.1 содержатся данные об учебных дисциплинах и преподавателях, ведущих эти дисциплины.

Таблица 3.1

Данные таблицы ДИСЦИПЛИНА_ПРЕПОДАВАТЕЛЬ

| Дисциплина | Преподаватель |
|---|---|
| Информационное обеспечение систем управления | Иванов |
| Базы данных и знаний | Иванов |
| Программирование и основы алгоритмизации | Петров |
| Автоматизация проектирования систем и средств управления | Петров |

В этой таблице фамилии преподавателей появляются неоднократно, поскольку один преподаватель может читать лекции по нескольким предметам. Однако повторение фамилий в данном случае

нельзя считать избыточным. Это объясняется тем, что при удалении повторяющихся фамилий теряется информация о том, кто ведет данный предмет.

Добавим в таблицу ДИСЦИПЛИНА_ПРЕПОДАВАТЕЛЬ еще один атрибут – «Должность». Такое небольшое изменение таблицы приводит к появлению избыточного дублирования данных (табл. 3.2). Естественно, доцент Иванов, ведущий разные дисциплины, имеет одну и ту же должность. Это же можно сказать и о профессоре Петрове.

Таблица 3.2

Пример избыточного дублирования данных

| Дисциплина | Преподаватель | Должность |
|---|---|---|
| Информационное обеспечение систем управления | Иванов | Доцент |
| Базы данных и знаний | Иванов | Доцент |
| Программирование и основы алгоритмизации | Петров | Профессор |
| Автоматизация проектирования систем и средств управления | Петров | Профессор |

Исключения избыточности в данном случае можно добиться путем разбиения табл. 3.2 на две таблицы:

ДИСЦИПЛИНА_ПРЕПОДАВАТЕЛЬ (Дисциплина, Преподаватель) (см. табл. 3.1);

ПРЕПОДАВАТЕЛЬ_ДОЛЖНОСТЬ (Преподаватель, Должность) (табл. 3.3).

Таблицы связаны друг с другом через атрибут «Преподаватель».

Таблица 3.3

Преподаватели и должности

| Преподаватель | Должность |
|---|---|
| Иванов | Доцент |
| Петров | Профессор |

### *3.2.2. Аномалии ввода, удаления и обновления*

Избыточное дублирование данных приводит к появлению проблем при вводе, удалении и обновлении данных, которые принято называть *аномалиями.*

Аномалиями будем называть такую ситуацию в таблицах БД, которая приводит к нарушениям целостности БД либо существенно усложняет обработку данных.

Выделяют *три вида аномалий*: аномалии ввода, аномалии удаления и аномалии обновления.

В качестве примера рассмотрим таблицу РАБОТНИК (Табельный_номер, Фамилия, Специальность, Номер_объекта). Данные этой таблицы представлены в табл. 3.4 и отражают сведения о работниках строительной фирмы.

Таблица 3.4

Таблица РАБОТНИК

| Табельный номер | Фамилия | Специальность | Номер объекта |
|---|---|---|---|
| 1234 | Иванов | Электрик | 1 |
| 1234 | Иванов | Электрик | 2 |
| 1235 | Петров | Плотник | 1 |
| 1235 | Петров | Плотник | 4 |
| 1235 | Петров | Плотник | 8 |
| 1236 | Сидоров | Маляр | 2 |

Приведенная табл. 3.4 содержит избыточные данные. Проблема возникает из-за того, что один и тот же работник может работать на разных объектах. Например, в трех записях, соответствующих рабочему с табельным номером 1235, повторяется одно и то же имя и информация о специальности. Эта избыточность данных не только приводит к увеличению объема требуемой памяти компьютера, но может вызвать и нарушение целостности данных в базе данных.

*Аномалии обновления* проявляются в том, что изменение значения одного данного может повлечь за собой просмотр всей таблицы и соответствующее изменение некоторых других записей.

Предположим, что специальность Петрова была указана неверно, ее изменение потребует просмотра всей таблицы. И если исправление было внесено только в первую запись, то между записями, содержащими информацию о Петрове, возникает несоответствие. База данных становится противоречивой, так как имеет место проблема нарушения целостности данных.

*Аномалия удаления* состоит в том, что при удалении какого-либо данного из таблицы может пропасть другая информация, не связанная напрямую с удаляемыми данными.

Предположим, что Петров длительное время отсутствовал на рабочем месте по причине болезни и за это время все объекты, в строительстве которых он принимал участие, были завершены. Принято решение удалить все записи таблицы, связанные с завершенными объектами. В этом случае информация о Петрове будет потеряна полностью. В такой ситуации система выдаст неверный ответ на любой запрос о Петрове, например о его специальности.

*Аномалия ввода* возникает в случаях, когда информацию в таблицу нельзя поместить до тех пор, пока она неполная, либо вставка новой записи требует дополнительного просмотра таблицы.

Рассмотрим такой случай. Принят новый работник. Этот работник еще не получил назначения ни на один из объектов. Если неопределенные значения NULL не допускаются в базе данных, то нельзя ввести информацию о новом работнике. Можно, конечно, поместить неопределенные значения (например, пробелы) в поля, но это чревато трудностями и не всегда возможно. Если подобные атрибуты входят в состав ключа, то невозможно организовать в таблице поиск записи с неопределенным значением ключа.

### *3.2.3. Функциональные зависимости*

Метод нормальных форм основан на фундаментальном в теории реляционных БД понятии – функциональной зависимости, которая описывает связь между атрибутами таблицы.

Если в таблице *R*, содержащей атрибуты *A* и *B*, каждому значению атрибута *A* соответствует в точности одно значение атрибута *B*, то говорят, что атрибут *B функционально зависит* от атрибута *A*.

Следовательно, если нам известно значение атрибута *A*, то при рассмотрении таблицы с такой функциональной зависимостью в любой момент времени во всех строках этой таблицы, содержащих указанное значение атрибута *A*, мы найдем одно и то же значение атрибута *B*. Таким образом, если две строки имеют одно и то же значение атрибута *A*, то они обязательно имеют одно и то же значение атрибута *B*.

Таким образом, функциональные зависимости (ФЗ) позволяют накладывать дополнительные ограничения на реляционную схему.

Функциональная зависимость *B* от *A* обозначается *A→B*. Читается это выражение следующим образом: «*A* функционально определяет *B*» или «*B* функционально зависит от *A*».

Например, в таблице ДОЛЖНОСТЬ (Код_должности, Название, Оклад) атрибут «Код_должности» однозначно определяет значения атрибутов «Название» и «Оклад». Это можно записать следующим образом:

Код_должности → Название,
Код_должности → Оклад.

Атрибут в левой части ФЗ называется *детерминантом*, так как его значение определяет значение атрибута в правой части. Первичный ключ таблицы является детерминантом, так как его значение однозначно определяет значение каждого атрибута таблицы.

Если существует функциональная зависимость вида *A→B* и *B→A*, то между атрибутами *A* и *B* имеется взаимно однозначное соответствие, или *функциональная взаимозависимость*. Такая взаимозависимость между атрибутами *A* и *B* обозначается *A↔B*, или *B↔A*. Например, имеется таблица

СЛУЖАЩИЙ (Номер_паспорта, ИНН, ФИО)

Между атрибутами «Номер_паспорта» и «ИНН» существует функциональная взаимозависимость, так как значение атрибута «Номер_паспорта» однозначно определяет значение атрибута «ИНН» и в тоже время имеется обратная функциональная зависимость: каждому значению атрибута «ИНН» соответствует одно значение атрибута «Номер_паспорта».

Отметим, что атрибуты *A* и *B* могут быть составными.

*Полной функциональной зависимостью* называется зависимость неключевого атрибута от всего составного ключа.

*Частичной функциональной зависимостью* называется зависимость неключевого атрибута от части составного ключа.

Например, в таблице

РАБОТНИК (Табельный_номер, Фамилия, Специальность, Номер_объекта)

первичный ключ состоит из двух атрибутов – «Табельный_номер» и «Номер_объекта», т. е. является составным.

Значение атрибута «Табельный_номер» однозначно определяет значения атрибутов «Фамилия» и «Специальность», т. е. в таблице РАБОТНИК имеет место частичная функциональная зависимость:

Табельный_номер → Фамилия,
Табельный_номер → Специальность.

Атрибут *C* зависит от атрибута *A транзитивно,* если для атрибутов *A*, *B*, *C* выполняются условия *A*→*B* и *B*→*C*, но обратная зависимость отсутствует.

Например, в таблице

ГОС_СЛУЖАЩИЙ (Код_Сл, ФИО, Должность, Оклад)

существует транзитивная зависимость между атрибутами

Код_Сл→Должность→Оклад.

Наличие транзитивной зависимости объясняется тем, что оклад государственного служащего зависит от занимаемой им должности.

Выявление зависимостей между атрибутами необходимо для выполнения нормализации базы данных. Основной способ определения наличия функциональных зависимостей – внимательный анализ семантики атрибутов.

При выявлении функциональных зависимостей между атрибутами таблицы необходимо прежде всего проводить четкое различие между значениями, хранящимися в атрибуте в определенный момент времени, и множеством всех возможных значений, которые могут храниться в атрибуте в тот или иной момент времени.

Иными словами, функциональная зависимость – свойство реляционной схемы (т. е. абстрактной структуры), а не свойство конкретного экземпляра схемы (т. е. ее реализации).

### *3.2.4. Метод нормальных форм*

Метод нормальных форм позволяет избежать избыточного дублирования данных БД, а также проблем при выполнении операций вставки, удаления и обновления данных.

Метод нормальных форм заключается в последовательном преобразовании таблиц базы данных таким образом, чтобы представить

логическую модель БД в виде совокупности таблиц, находящихся в *нормальной форме* (НФ).

Каждая следующая нормальная форма ограничивает определенный тип функциональных зависимостей, устраняет соответствующие аномалии при выполнении операций над таблицами БД и сохраняет свойства предшествующих нормальных форм.

В теории проектирования баз данных известны следующие нормальные формы: 1НФ, 2НФ, 3НФ, нормальная форма Бойса – Кодда (НФБК), 4НФ, 5НФ, 6НФ.

Нормальные формы подчиняются правилу вложенности по возрастанию номеров: если БД находится в 4НФ, то она будет соответствовать НФБК, 3НФ, 2НФ и 1НФ. И наоборот: если БД находится в 1НФ, но не в 2НФ, то она не будет соответствовать ни 3НФ, ни 4НФ.

Процесс нормализации использует метод декомпозиции, который заключается в разбиении исходной таблицы на несколько взаимосвязанных таблиц, составленных таким образом, чтобы обратное соединение этих таблиц позволяло вновь получить исходную таблицу. Возможность выполнения обратного преобразования – весьма важная характеристика, поскольку она означает, что декомпозиция всегда выполняется без потерь информации. Это предполагает отсутствие потери строк и появление ранее не существовавших строк.

На практике очень часто достаточно привести каждую таблицу базы данных хотя бы к 3НФ.

***Первая нормальная форма*** (1НФ) – это основа реляционной модели.

Таблица представлена в *первой нормальной форме* тогда и только тогда, когда все ее столбцы содержат только неделимые значения (в том смысле, что их дальнейшее разложение невозможно), данные в столбце одного типа и в ней отсутствуют повторяющиеся строки. Другими словами, таблица, соответствующая 1НФ, не должна содержать ячеек, включающих в себя несколько значений. Примером таблицы, которая не приведена к первой нормальной форме, служит табл. 3.5.

Таблица 3.5

Пример ненормализованной таблицы

| Дисциплина | Преподаватель |
|---|---|
| Информационное обеспечение систем управления<br>Базы данных и знаний | Иванов |
| Программирование и основы алгоритмизации<br>Автоматизация проектирования систем управления | Петров |

Ячейки столбца «Дисциплина» этой таблицы содержат множественные значения. Приведение такой таблицы к 1НФ осуществляется путем разбиения строк таким образом, чтобы в ее ячейках не содержались множественные значения. В результате этого получим табл. 3.6 – таблицу в первой нормальной форме.

Таблица 3.6

Пример таблицы в 1НФ

| Дисциплина | Преподаватель |
|---|---|
| Информационное обеспечение систем управления | Иванов |
| Базы данных и знаний | Иванов |
| Программирование и основы алгоритмизации | Петров |
| Автоматизация проектирования систем управления | Петров |

***Вторая нормальная форма*** (2НФ): если таблица находится в 1НФ, то отображение ее данных и построение к ней запросов не составляют никакой сложности. Однако в этом случае не исключено избыточное дублирование данных в строках.

Реляционная таблица находится во *второй нормальной форме* тогда и только тогда, когда она находится в первой нормальной форме и каждый неключевой атрибут функционально полно зависит от первичного ключа, т. е. никакие неключевые атрибуты не являются функционально зависимыми лишь от части ключа.

Следовательно, 2НФ может быть нарушена только в том случае, если ключ составной, т. е. ключом является набор из нескольких атрибутов.

Если все возможные ключи таблицы содержат по одному атрибуту, то эта таблица находится во второй нормальной форме, так как

в этом случае все неключевые атрибуты полностью зависят от возможных ключей. Если ключи состоят более чем из одного атрибута, таблица, заданная в первой нормальной форме, может не быть таблицей во второй нормальной форме. Приведение таблицы ко второй нормальной форме заключается в обеспечении полной функциональной зависимости всех атрибутов от ключа за счет разбиения таблицы на несколько таблиц, в которых все имеющиеся атрибуты будут иметь полную функциональную зависимость от ключа этой таблицы. В процессе приведения модели ко второй нормальной форме в основном исключаются аномалии дублирования данных.

В таблице РАБОТНИК (Табельный_номер, Фамилия, Специальность, Номер_объекта) (см. табл. 3.4) первичный ключ состоит из атрибутов «Табельный_номер» и «Номер_объекта». При этом фамилия определяется атрибутом «Табельный_номер», т. е. функционально зависит от части ключа. Эта таблица не удовлетворяет 2НФ. Если таблицу оставить в таком виде, то могут возникнуть следующие проблемы:

- имя работника повторяется в каждой строке, относящейся к назначению этого работника. Если имя работника изменяется, то требуется обновить все строки, содержащие информацию о назначениях этого работника;
- из-за избыточности может возникнуть несоответствие данных, когда в разных строках содержатся разные фамилии для одного и того же работника;
- если в какой-то момент времени работник не имеет назначений, то может отсутствовать строка, в которой хранится фамилия работника.

Для решения этих проблем проведем декомпозицию таблицы путем разбиения ее на две взаимосвязанные реляционные таблицы.

Процесс разбиения на две таблицы состоит из следующих шагов.

1. В исходной таблице выявляются атрибуты, зависящие от части ключа. Создается новая таблица без атрибутов исходной таблицы, находящихся в частичной функциональной зависимости от первичного ключа.

В рассматриваемом примере это будет таблица

НАЗНАЧЕНИЕ (Табельный_номер, Номер_объекта),

данные которой представлены в табл. 3.7.

Таблица 3.7

Данные таблицы НАЗНАЧЕНИЕ

| Табельный номер | Номер объекта |
|---|---|
| 1234 | 1 |
| 1234 | 2 |
| 1235 | 1 |
| 1235 | 4 |
| 1235 | 8 |
| 1236 | 2 |

2. Создается таблица, атрибутами которой являются части составного ключа и атрибуты, зависящие от этих частей.

В результате выполнения этого шага исходная таблица преобразуется в таблицу

РАБОТНИК (Табельный_номер, Фамилия, Специальность),

данные которой приведены в табл. 3.8.

Таблица 3.8

Данные таблицы РАБОТНИК

| Табельный номер | Фамилия | Специальность |
|---|---|---|
| 1234 | Иванов | Электрик |
| 1235 | Петров | Плотник |
| 1236 | Сидоров | Маляр |

Таблицы НАЗНАЧЕНИЕ и РАБОТНИК не содержат частичных функциональных зависимостей и поэтому находятся во второй нормальной форме.

Внешний ключ – атрибут «Табельный_номер».

***Третья нормальная форма*** (3НФ): в общем случае 1НФ и 2НФ рассматриваются как промежуточные ступени в процессе нормализации базы данных. Большая часть СУБД ориентирована на достижение третьей нормальной формы.

Таблица находится в *третьей нормальной форме* тогда и только тогда, когда она находится во второй нормальной форме и не содержит транзитивных зависимостей.

Рассмотрим таблицу ВЫПЛАТА (Табельный_номер, Специальность, Премия). Полагаем, что размер премиальных зависит от специальности. В этом случае имеют место следующие ФЗ:

Табельный_номер → Специальность,

Табельный_номер → Премия,

Специальность → Премия.

Первые две ФЗ удовлетворяют критерию 3НФ. В третьей функциональной зависимости требование 3НФ нарушено. В то же время таблица удовлетворяет 2НФ, так как ключ состоит из одного атрибута. Рассмотрим недостатки, присущие таблице, не удовлетворяющей 3НФ:

- размер премиальных для вида специальности повторяется в каждой строке, относящейся к работнику этой специальности. Это избыточные данные;
- если размер премиальных для вида специальности изменяется, то нужно обновить все строки, содержащие эту специальность;
- если строка удаляется, то можно потерять информацию о размере премиальных для данной специальности. Следовательно, таблица подвержена аномалиям обновления и удаления;
- если в какой-то момент времени отсутствуют работники данной специальности, то может не оказаться строки, в которой можно хранить размер премиальных. Это аномалия ввода.

Нормализация таблиц 2НФ с образованием таблиц 3НФ предусматривает устранение транзитивных зависимостей. Если в таблице существует транзитивная зависимость между атрибутами, то транзитивно зависимые атрибуты удаляются из нее и помещаются в новую таблицу. Если хотя бы одна из полученных таблиц нарушает 3НФ, то процесс разбиения продолжается.

Для приведения таблицы к 3НФ воспользуемся методом декомпозиции и разобьем таблицу ВЫПЛАТА на две таблицы:

РАБОТНИК (Табельный_номер, Специальность)

ПРЕМИЯ (Специальность, Премия)

Обе таблицы имеют простой первичный ключ, следовательно, они находятся во 2НФ. В обеих таблицах отсутствуют транзитивные зависимости. Таким образом, таблицы РАБОТНИК и ПРЕМИЯ находятся в 3НФ.

Дочерняя таблица РАБОТНИК связана с родительской таблицей ПРЕМИЯ внешним ключом, в качестве которого используется атрибут «Специальность».

На практике построения таблиц в 3НФ в большинстве случаев достаточно и приведением к ним процесс проектирования реляционной БД заканчивается.

Если в таблице имеется зависимость атрибутов составного ключа от неключевых атрибутов, то необходимо перейти к нормальной форме Бойса – Кодда (НФБК).

***Нормальная форма Бойса – Кодда:*** 3НФ упрощает решение проблем, связанных с обработкой данных, только если в таблицах отсутствуют какие-либо другие функциональные зависимости. Оказывается, если таблица имеет:

1) два (или более) потенциальных ключа, которые
2) являются составными
3) и имеют по крайней мере один общий атрибут,

определение 3НФ оказывается неадекватным. Поэтому впоследствии исходное определение 3НФ было заменено более строгим определением Бойса – Кодда.

Определение *нормальной формы Бойса – Кодда* звучит следующим образом [1]: таблица находится в НФБК, если она находится в 3НФ и детерминант любой функциональной зависимости является потенциальным ключом.

Иногда НФБК еще называют *усиленной третьей нормальной формой*, поскольку каждая таблица в НФБК является также таблицей 3НФ, но не всякая таблица в 3НФ является таблицей в НФБК.

Следует заметить, что на практике таблицы с условиями 1 – 3 встречаются достаточно редко. Для любой таблицы, в которой не выполняются все эти условия, 3НФ и НФБК полностью эквивалентны. Следовательно, таблицы с простым первичным ключом, которые находятся в третьей нормальной форме, автоматически находятся и в нормальной форме Бойса – Кодда.

Рассмотрим пример, который приведен в источнике [10].

Представим, что у нас есть организация, которая реализует множество различных проектов. При этом в каждом проекте работа ведется по нескольким функциональным направлениям, в каждом из которых есть свой куратор. Сотрудник может быть куратором только

того направления, на котором он специализируется, т. е. если сотрудник – программист, он не может курировать в проекте направление, связанное с бухгалтерией. И еще одно допущение: в организации фамилии и инициалы сотрудников не совпадают.

Допустим, что нам нужно хранить информацию о кураторах всех проектов по каждому направлению.

В итоге получим следующую реляционную таблицу:

ПРОЕКТ_КУРАТОР (Проект, Направление, Куратор),

содержание которой приведено в табл. 3.9.

Таблица 3.9

Таблица ПРОЕКТ_КУРАТОР

| Проект | Направление | Куратор |
|---|---|---|
| 1 | Разработка | Иванов И.И. |
| 1 | Бухгалтерия | Сергеев С.С. |
| 2 | Разработка | Иванов И.И. |
| 2 | Бухгалтерия | Петров П.П. |
| 2 | Реализация | Тимофеев Т.Т. |
| 3 | Разработка | Андреев А.А. |

Набор атрибутов «Проект» и «Направление» определяет атрибут «Куратор», а набор атрибутов «Проект» и «Куратор» определяет атрибут «Направление». Следовательно, каждый из этих наборов является потенциальным ключом. Любой из них может быть первичным ключом. В нашем примере первичным ключом выбран набор атрибутов «Проект» и «Направление».

Таблица ПРОЕКТ_КУРАТОР находится в третьей нормальной форме, так как неключевой атрибут зависит от всего ключа, а не от какой-то его части, и в ней отсутствуют транзитивные зависимости.

При манипулировании данными этой таблицы возникают следующие проблемы. Предположим, что проект с номером 1 закончен и принято решение удалить о нем информацию из таблицы. В результате будет потерян тот факт, что Сергеев С.С. является куратором направления «Бухгалтерия». Это аномалия удаления. Существует еще одна проблема при обработке данных в этой таблице. Невозможно записать в эту таблицу тот факт, что Мишин М.М. курирует

IT-направление до тех пор, пока не появится проект, в котором присутствуют вопросы, связанные с IT-технологиями. А это аномалия ввода.

Для устранения указанных выше аномалий следует перевести таблицу ПРОЕКТ_КУРАТОР в нормальную форму Бойса – Кодда.

Таблица не находится в нормальной форме Бойса – Кодда, потому что в ней при наличии двух потенциальных составных ключей есть функциональная зависимость

Куратор → Направление,

детерминант которой не является потенциальным ключом. Таким образом, в таблице ПРОЕКТ_КУРАТОР присутствуют все три условия (1 – 3).

Чтобы привести таблицу ПРОЕКТ_КУРАТОР к нормальной форме Бойса – Кодда, необходимо сделать декомпозицию данной таблицы, т. е. разбить эту таблицу на несколько таблиц, в данном примере – на две таблицы: таблицу КУРАТОР (ФИО, Направление) (табл. 3.10) и таблицу ПРОЕКТ_КУРАТОР (Проект, ФИО) (табл. 3.11).

Таблица 3.10

Данные таблицы КУРАТОР

| ФИО | Направление |
|---|---|
| Иванов И.И. | Разработка |
| Сергеев С.С. | Бухгалтерия |
| Петров П.П. | Бухгалтерия |
| Тимофеев Т.Т. | Реализация |
| Андреев А.А. | Разработка |

Таблица 3.11

Данные таблицы ПРОЕКТ_КУРАТОР

| Проект | ФИО |
|---|---|
| 1 | Иванов И.И. |
| 1 | Сергеев С.С. |
| 2 | Иванов И.И. |
| 2 | Петров П.П. |
| 2 | Тимофеев Т.Т. |
| 3 | Андреев А.А. |

Таким образом, в таблице КУРАТОР у нас хранится список кураторов и направлений, которые они могут курировать, а в таблице ПРОЕКТ_КУРАТОР отражается связь проектов и кураторов. В полученных таблицах условия 1 – 3 не выполняются, поэтому для них ЗНФ и НФБК полностью эквивалентны. Несложно заметить, что в таблицах КУРАТОР и ПРОЕКТ_КУРАТОР аномалии ввода и удаления отсутствуют.

***Четвертая нормальная форма*** (4НФ): нормальная форма Бойса – Кодда позволяет устранить любые аномалии, вызванные функциональными зависимостями. Поэтому процесс нормализации очень часто заканчивается приведением таблиц к НФБК.

Однако в таблицах, кроме функциональных, могут присутствовать и другие, более сложные, виды зависимостей между атрибутами. Так, в результате теоретических исследований был выявлен еще один тип зависимости – многозначная зависимость, которая при проектировании баз данных также может вызвать проблемы, связанные с избыточностью данных. Поэтому при разработке больших систем, когда необходимо обеспечить максимальное сокращение объемов хранимых данных, желательно провести дальнейшую нормализацию.

Прежде чем перейти к рассмотрению четвертой нормальной формы, введем *понятие многозначной зависимости*.

Атрибут $A$ многозначно определяет атрибут $B$ той же таблицы, если для каждого значения атрибута $A$ существует хорошо определенное множество соответствующих значений $B$.

Математически многозначная зависимость обозначается следующим образом: $A \text{ –>> } B$.

Другими словами, при многозначной зависимости одному значению некоторого атрибута соответствует устойчиво постоянное множество значений другого атрибута. Например, если таблица содержит данные о граничащих друг с другом странах, то, зная название страны, можно определить названия всех соседних с нею стран.

Рассмотрим конкретную ситуацию. Например, преподаватель университета может вести занятия по нескольким предметам и работать в нескольких комиссиях. Эту ситуацию моделирует таблица

ПРЕПОДАВАТЕЛЬ (<u>Фамилия, Комиссия, Предмет</u>)

В табл. 3.12 приведены данные реляционной таблицы ПРЕПОДАВАТЕЛЬ, в которой каждое значение атрибута «Комиссия» сочетается с каждым значением атрибута «Предмет» как минимум в одной строке.

Таблица 3.12

Данные реляционной таблицы ПРЕПОДАВАТЕЛЬ

| Фамилия | Комиссия | Предмет |
|---|---|---|
| Иванов | Государственная аттестационная | Информатика |
| Иванов | Государственная аттестационная | Базы данных |
| Иванов | Государственная аттестационная | Программирование |
| Иванов | Приемная | Информатика |
| Иванов | Приемная | Базы данных |
| Иванов | Приемная | Программирование |
| Петров | Дисциплинарная | Математический анализ |
| Петров | Дисциплинарная | Статистика |
| Петров | Приемная | Математический анализ |
| Петров | Приемная | Статистика |

Эта таблица находится в третьей нормальной форме и нормальной форме Бойса – Кодда, так как единственный возможный ключ таблицы – составной атрибут «Фамилия» + «Комиссия» + «Предмет».

Однако в табл. 3.12 существуют две многозначные зависимости:

Фамилия – >> Комиссия
Фамилия – >> Предмет

При этом очевидно, что атрибуты «Комиссия» и «Предмет» не зависят друг от друга. Это приводит к тому, что при добавлении или удалении информации о некотором преподавателе приходится добавлять/удалять из таблицы такое количество записей, которое равно произведению количества комиссий, в работе которых принимает участие преподаватель, на количество предметов, по которым он ведет занятия. Поэтому важный этап процесса нормализации – приведение таблиц к 4НФ.

Таблица находится в *четвертой нормальной форме* в том и только в том случае, если она находится в НФБК и в случае существования многозначной зависимости $A$ – >> $B$ все остальные атрибуты таблицы функционально зависят от $A$.

Таким образом, приведение таблицы к 4НФ требуется при наличии в ней более чем одной многозначной функциональной зависимости.

Так как проблема многозначных зависимостей возникает в связи с многозначными атрибутами, то можно решить эту проблему, поместив каждый многозначный атрибут в отдельную таблицу вместе с ключом, от которого атрибут зависит. В нашем примере таблица ПРЕПОДАВАТЕЛЬ может быть разбита на две таблицы:

КОМИССИЯ (Фамилия, Комиссия)
ПРЕДМЕТ (Фамилия, Предмет)

Ключом каждой полученной таблицы 4НФ являются оба атрибута таблицы.

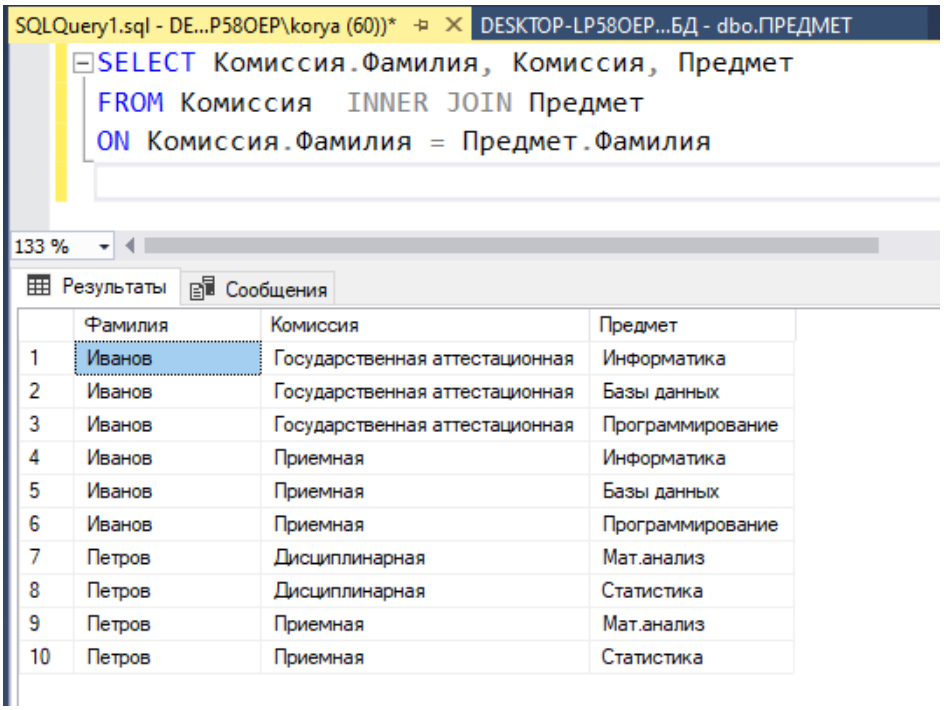
Следует отметить, что в реальности при правильном проектировании базы данных такую ситуацию, когда в одной таблице хранятся абсолютно не связанные друг с другом данные, вряд ли можно получить. Поэтому этот пример чисто теоретический и приводится для демонстрации принципов четвертой нормальной формы.

Во всех рассмотренных примерах при приведении таблиц к 2НФ, 3НФ и 4НФ исходная таблица разбивалась на две взаимосвязанные таблицы без потери данных. Последнее означает, что в случае соединения таблиц, которые были получены в результате разбиения, будет формироваться та же самая информация, что и в исходной таблице до разбиения.

В этом нетрудно убедиться, если написать команду SELECT, содержащую операцию соединения. Например, выполним команду SELECT, которая возвращает фамилии преподавателей, комиссии, в которых они работают, и предметы, которые они преподают, из таблиц КОМИССИЯ и ПРЕДМЕТ:

```
SELECT Комиссия.Фамилия, Комиссия, Предмет
FROM Комиссия INNER JOIN Предмет
ON Комиссия.Фамилия = Предмет.Фамилия
```

Как видно по рис. 3.18, результат запроса совпадает с данными в исходной таблице (см. табл. 3.12).




| | Фамилия | Комиссия | Предмет |
|---|---|---|---|
| 1 | Иванов | Государственная аттестационная | Информатика |
| 2 | Иванов | Государственная аттестационная | Базы данных |
| 3 | Иванов | Государственная аттестационная | Программирование |
| 4 | Иванов | Приемная | Информатика |
| 5 | Иванов | Приемная | Базы данных |
| 6 | Иванов | Приемная | Программирование |
| 7 | Петров | Дисциплинарная | Мат.анализ |
| 8 | Петров | Дисциплинарная | Статистика |
| 9 | Петров | Приемная | Мат.анализ |
| 10 | Петров | Приемная | Статистика |

Рис. 3.18. Результаты запроса с операцией соединения таблиц КОМИССИЯ и ПРЕДМЕТ

Завершая рассмотрение нормальных форм и нормализации, приведем некоторую информацию слов о 5НФ и 6НФ.

Можно показать, что, в отличие от всех ранее рассмотренных примеров, таблицу, находящуюся в 4НФ, нельзя разбить на две таблицы без потери информации. Однако теоретически возможны ситуации, когда таблицу, находящуюся в 4НФ, можно разбить без потери информации на три или более таблицы. Это связано еще с одним редко встречающимся типом зависимостей – *зависимостями соединения*. Для устранения таких зависимостей используется 5НФ. Детально этот вопрос обсуждается в источнике [1]. Заметим лишь, что зависимость соединения является обобщением многозначной зависимости.

Следует отметить, что таблица, находящаяся в 5НФ, не поддается дальнейшей нормализации методом декомпозиции. Таким образом, если таблица находится в 5НФ, то гарантируется, что она не содержит аномалий, которые могут быть исключены посредством ее разбиения на таблицы.

Понятие 6НФ связано с некоторыми особыми типами аномалий, возникающих в весьма специфическом типе реляционных баз данных – хронологических, и используется только для этого типа БД. *Хронологическая база данных* – это база, которая может хранить не только те-

кущие данные, но и данные, относящиеся к прошлым или будущим периодам времени.

Подробнее о 5НФ и 6НФ можно прочитать в книге [1].

А если говорить о реальных данных, то нормализация до 4НФ, как и до всех последующих нормальных форм, в настоящее время практически не встречается. Если 4НФ еще как-то можно представить и даже встретить данные, нормализованные до этой формы, то встретить данные, нормализованные до 5НФ или 6НФ, практически невозможно.

Возникает вопрос: почему не нормализуют данные до 5НФ или 6НФ? Ведь каждая нормальная форма устраняет определенные аномалии, и если сделать полностью нормализованную базу данных, то, по сути, она будет идеальной, не содержащей ни одной аномалии. И это хорошо.

Однако здесь всплывают вопросы, связанные с производительностью системы. Суть дела в том, что при приведении таблицы в 5НФ и 6НФ потребуется исходную таблицу разбить без потери информации более чем на две таблицы. В результате чего в полностью нормализованной базе данных для выполнения запроса потребуется соединить столь много таблиц, что производительность такой системы не сможет удовлетворять практических запросов. Поэтому в процессе проектирования базы данных необходимо следовать здравому смыслу и найти баланс между отсутствием аномалий и приемлемой производительностью.

Также стоит отметить, что таблицы, которые необходимо нормализовать до 5НФ, встречаются крайне редко, т. е. это очень частный случай. Более того, такие таблицы являются следствием не совсем удачного проектирования. Поэтому следует придерживаться общей рекомендации: структуру базы данных строить таким образом, чтобы избежать применения 4НФ и 5НФ.

## Вопросы для самопроверки

1. В чем состоит избыточное и неизбыточное дублирование данных?
2. Какие аномалии необходимо устранить при проектировании реляционной базы данных?

3. Что значит: *X* функционально определяет *Y*?

4. Что такое детерминант?

5. Какие существуют виды зависимостей между атрибутами таблицы?

6. Что такое первая нормальная форма?

7. При каких условиях таблица находится во второй нормальной форме?

8. Как таблица приводится ко второй нормальной форме?

9. При каких условиях таблица находится в третьей нормальной форме?

10. Как таблица приводится к третьей нормальной форме?

11. Какие условия должны выполняться, чтобы таблица находилась в усиленной третьей нормальной форме?

12. Что такое многозначная зависимость?

13. Какие условия должны выполняться, чтобы таблица находилась в четвертой нормальной форме?

14. Что такое нормализация таблиц?

## Практические задания

1. Дана таблица ЭКЗАМЕН.

| Код студента | Фамилия | Код экзамена | Предмет | Дата | Оценка |
|---|---|---|---|---|---|
| 1 | Сергеев | 1 | Математика | 15.06.21 | 4 |
| 2 | Иванов | 1 | Математика | 15.06.21 | 5 |
| 1 | Сергеев | 2 | Физика | 20.06.21 | 5 |
| 2 | Иванов | 2 | Физика | 20.06.21 | 4 |

Какие функциональные зависимости существуют в этой таблице?

2. Для каждой из следующих реляционных таблиц определите, каким нормальным формам удовлетворяют таблицы, и покажите, как разбить таблицу на несколько таблиц, каждая из которых удовлетворяет нормальной форме самого высокого порядка.

а) РАБОТНИК (Идентификатор_работника, Имя, Идентификатор_супруга, Имя_супруга)

Функциональная зависимость:

Идентификатор_супруга → Имя_супруга

б) СЛУЖАЩИЙ (Табельный_номер, Имя, Адрес, Телефон, Профессия)

Функциональная зависимость:

Адрес → Телефон

в) ПРОДАЖА (Дата, Клиент, Товар, Продавец, Город_продавца)

Функциональная зависимость:

Продавец → Город_продавца

3. Определите, каким нормальным формам удовлетворяют таблицы модели, созданные в соответствии с практическим заданием п. 3.1.

## 3.3. Целостность баз данных

### *3.3.1. Понятие целостности данных*

Для пользователей информационной системы недостаточно, чтобы БД просто отражала объекты реального мира. Важно, чтобы такое отражение было однозначным и непротиворечивым. В этом случае говорят, что БД удовлетворяет условию целостности. Поэтому одна из основных проблем при работе с базами данных – поддержка целостности данных.

Под *целостностью* понимают свойство базы данных, означающее, что она содержит полную, непротиворечивую информацию, адекватно отражающую предметную область.

Таким образом, любое изменение в предметной области, значимое для построенной модели, должно отражаться в базе данных.

Например, в базе данных «Библиотека» наиболее важным является процесс взятия книги читателем или возврат любой книги в библиотеку. Система должна отслеживать эти процессы в соответствии с изменениями в реальной предметной области. При этом наличие у экземпляра книги указателя на его отсутствие в библиотеке и одновременное отсутствие записи о конкретном номере читательского билета, за которым числится этот экземпляр книги, является противоречием, такого быть не должно. И в модели данных должны быть предусмотрены средства и методы, которые позволят обеспечивать ди-

намическое отслеживание в базе данных согласованных действий, связанных с согласованным изменением информации.

Выполнение операций ввода, обновления и удаления данных в таблицах базы данных может привести к нарушению целостности данных, т. е. к потере их корректности и непротиворечивости. В системах со многими пользователями целостность может быть нарушена при одновременном обращении к одним и тем же фрагментам данных.

Поддержание целостности БД включает проверку (контроль) целостности и ее восстановление в случае обнаружения противоречий в базе. Для поддержания целостности данных используют ограничения целостности.

*Ограничение целостности* – это некоторое утверждение, которое может быть истинным или ложным в зависимости от состояния базы данных.

Ограничения целостности позволяют свести к минимуму ошибки, возникающие при обновлении и обработке данных.

Обычно ограничения целостности формулируются на естественном языке и/или с использованием предикатов.

Примерами ограничений целостности могут служить следующие утверждения:

- возраст сотрудника не может быть меньше 14 и больше 65 лет;
- каждый сотрудник имеет уникальный табельный номер;
- сотрудник обязан числиться в одном отделе;
- сумма накладной обязана равняться сумме произведений цен товаров на количество товаров для всех товаров, входящих в накладную.

Любое ограничение целостности является семантическим понятием, т. е. появляется как следствие определенных свойств объектов предметной области и/или их взаимосвязей.

База данных находится в *согласованном (целостном) состоянии*, если выполнены (удовлетворены) все ограничения целостности, определенные для базы данных.

В данном определении важно подчеркнуть, что должны быть выполнены не все вообще ограничения предметной области, а только те, которые определены в базе данных.

Вместе с понятием целостности базы данных возникает понятие *реакции системы на попытку нарушения целостности*. Система должна не только проверять, не нарушаются ли ограничения в ходе выполнения различных операций, но и должным образом реагировать, если операция приводит к нарушению целостности. Имеется два типа реакции на попытку нарушения целостности:

1) отказ выполнить «незаконную» операцию;
2) выполнение компенсирующих операций.

Например, если база данных знает, что в атрибуте «Возраст_сотрудника» должны быть целые числа в диапазоне от 14 до 65, то система отвергает попытку ввести значение возраста 66. При этом может генерироваться какое-нибудь сообщение для пользователя.

Работа системы по проверке ограничений изображена на рис. 3.19.


Рис. 3.19. Работа системы по проверке ограничений

Поясним, что представляют собой компенсирующие операции на следующем примере.

Пусть имеется база данных, в которой хранятся данные об отделах и работающих в них сотрудниках. Список отделов хранится в таблице

ОТДЕЛ (Код_отдела, Наименование_отдела, Количество_сотрудников)

Список сотрудников хранится в таблице

СОТРУДНИК (Код_сотрудника, Имя, Код_отдела).

Внешний ключ таблиц – атрибут «Код_отдела».

Ограничение целостности этой базы данных состоит в том, что атрибут «Количество_сотрудников» не может заполняться произвольными значениями. Это поле должно содержать количество сотрудников, реально числящихся в отделе.

С учетом этого ограничения можно заключить, что вставка нового сотрудника в таблицу *не может быть выполнена одной операцией*. При вставке нового сотрудника необходимо одновременно увеличить значение атрибута «Количество_сотрудников» на единицу, а при увольнении – уменьшать на единицу.

Ограничения целостности можно классифицировать несколькими способами:

- по способам реализации;
- времени проверки;
- области действия.

### *3.3.2. Способы реализации ограничений целостности*

Существуют два способа реализации ограничений:

- декларативная поддержка ограничений целостности;
- процедурная поддержка ограничений целостности.

***Декларативная поддержка ограничений целостности*** заключается в определении ограничений средствами языка определения данных DDL (Data Definition Language).

К такому типу поддержки целостности данных относятся:

- целостность сущностей (PRIMARY KEY);
- уникальные значения (UNIQUE KEY);
- обязательные данные (NOT NULL);
- ограничения для атрибутов (полей) (CHECK);
- значения по умолчанию (DEFAULT);
- ссылочная целостность (FOREIGN KEY);
- требования конкретного предприятия (бизнес-правила).

Большая часть перечисленных ограничений задается в командах CREATE TABLE.

*Целостность сущностей:* строки реляционной таблицы представляют в базе данных экземпляры конкретных экземпляров сущностей. Каждая строка любой таблицы должна отличатся от любой другой строки этой таблицы. Вполне очевидно, что если данное требование не соблюдается (т. е. строки в рамках одной таблицы не уникальны), то в базе данных может храниться противоречивая информация об одном и том же объекте.

Ключ таблицы однозначно определяет каждую строку и, следовательно, каждый экземпляр сущности.

*Правило целостности сущностей:* атрибуты, входящие в состав некоторого потенциального (первичного или альтернативного) ключа, не могут принимать неопределенных значений NULL.

Система управления должна отклонить любую попытку (при вводе или обновлении) поместить в БД строку, ключ которой или не определен (NULL), или имеет значение, уже введенное в БД. Таким образом, ограничение целостности сущностей может устанавливаться как на уровне таблицы, так и на уровне столбца.

Стандарт SQL позволяет задавать подобные требования поддержки целостности данных с помощью фразы PRIMARY KEY. В пределах таблицы она может указываться только один раз. Однако существует возможность гарантировать уникальность значений и для любых альтернативных ключей таблицы, что обеспечивает ключевое слово UNIQUE.

*Уникальные значения:* отличие ограничения уникального значения (UNIQUE) от ограничения первичного ключа (PRIMARY KEY) заключается в том, что его цель – не идентифицировать каждую строку, а только гарантировать ее уникальность. Кроме того, при определении альтернативных ключей рекомендуется использовать и спецификаторы NOT NULL.

*Обязательные данные:* некоторые атрибуты всегда должны содержать одно из допустимых значений, другими словами, эти атрибуты не могут иметь неопределенного значения NULL. Для задания ограничений подобного типа стандарт SQL предусматривает использование спецификации NOT NULL.

*Ограничения для атрибутов (полей):* каждый столбец имеет собственный домен – некоторый набор допустимых значений. Стандарт SQL предусматривает два различных механизма определения доменов. Первый состоит в использовании предложения CHECK, позволяющего задать требуемые ограничения для столбца или таблицы в целом, а второй предполагает применение команды CREATE DOMAIN.

Ограничение типа CHECK устанавливается или на атрибут, или на набор атрибутов и требует, чтобы определенное условие было равно TRUE для каждой строки. Если операция изменения данных приводит к тому, что условие принимает значение FALSE, то такая операция не выполняется. Ограничение типа CHECK позволяет задавать специфичные бизнес-правила: например, в БД в таблице ПРОДАВЦЫ для атрибута «Рейтинг» определено условие, ограничивающее значения этого атрибута целыми числами в диапазоне от 1 до 100. Один столбец может иметь несколько ограничений CHECK. Если для таблицы задано несколько ограничений CHECK, то они не должны конфликтовать друг с другом.

*Значения по умолчанию:* ограничение DEFAULT определяет значение по умолчанию, применяемое к каждому столбцу таблицы.

*Требования конкретного предприятия:* существует понятие «корпоративные ограничения целостности» как дополнительные правила поддержки целостности данных, определяемые пользователями, принятые на предприятии или администраторами баз данных. Ограничения предприятия называются бизнес-правилами.

Например, для базы данных «Кадры» к таким правилам могут быть отнесены следующие.

1. На работу могут приниматься граждане России не моложе 14 лет.

2. Неграждане России могут приниматься на работу при наличии эмиграционной карты.

3. Каждый работающий должен дать телефон для связи: он может быть рабочим или домашним.

Стандарт SQL позволяет реализовать бизнес-правила предприятий с помощью предложения CHECK.

*Ссылочная целостность*: указанное ограничение целостности касается внешних ключей. Внешний ключ – это поле (или множество

полей) одной таблицы, являющееся ключом другой (или той же самой) таблицы. Внешние ключи используются для установления логических связей между таблицами. Связь устанавливается путем присвоения значений внешнего ключа одной таблицы значениям ключа другой.

Стандарт языка SQL предусматривает механизм определения внешних ключей с помощью предложения FOREIGN KEY, а предложение REFERENCES определяет имя родительской таблицы, т. е. таблицы, в которой находится соответствующий первичный ключ.

Существует несколько важных моментов, связанных с внешними ключами. Сначала следует проанализировать, допустимо ли использование во внешних ключах неопределенных значений NULL. В общем случае если участие дочерней таблицы в связи обязательно, то рекомендуется запрещать применение неопределенных значений в соответствующем внешнем ключе. В то же время если имеет место необязательное участие дочерней таблицы в связи, то помещение неопределенных значений в атрибут внешнего ключа должно быть разрешено. Например, если в операции фиксации сделок некоторой торговой фирмы необходимо указать покупателя, то атрибут «Код_клиента» не может принимать неопределенное значение NULL. Если допускается продажа или покупка товара без указания клиента, то атрибут «Код_клиента» может принимать неопределенное значение NULL.

Следующая проблема связана с организацией поддержки ссылочной целостности при выполнении операций модификации данных в базе.

В качестве примера рассмотрим две таблицы, в которых *не заданы* правила ссылочной целостности:

ОТДЕЛ (Код_отдела, Наименование_отдела, Количество_сотрудников)

СОТРУДНИК (Код_сотрудника, Имя, Код_отдела)

Внешним ключом для связи между этими таблицами служит атрибут «Код_отдела».

Если в рассмотренном выше примере из родительской таблицы ОТДЕЛ удалить некоторую запись, не удалив при этом связанные с ней записи из дочерней таблицы СОТРУДНИК, то эти записи как бы

повиснут в воздухе. Они, по сути, становятся «мусором», на который тратится память компьютера.

Требование ссылочной целостности означает, что связываемые таблицы должны иметь общие атрибуты (поля), и заключается в том, что внешний ключ не может быть указателем на несуществующие значения в таблице. Для любой строки с конкретным значением внешнего ключа должна обязательно существовать связанная строка в родительской таблице с соответствующим значением первичного ключа. Ссылочная целостность обеспечивает сохранность связи между таблицами при добавлении, удалении или изменении строк.

Ссылочная целостность может нарушиться в результате операций, изменяющих состояние базы данных. Таких операций три – вставка, обновление и удаление строк в таблицах. Так как в определении ссылочной целостности участвуют две таблицы – родительская и дочерняя, а в каждой из них возможны три операции – вставка, обновление, удаление, то нужно рассмотреть шесть различных вариантов.

Для родительской таблицы

- *Вставка строки в родительскую таблицу*. Такая вставка *не может вызвать нарушения ссылочной целостности*. Добавленная строка становится родительским объектом, не имеющим дочерних объектов.
- *Обновление первичного ключа в строке родительской таблицы*. Если значение первичного ключа некоторой строки родительской таблицы будет обновлено и есть строки в дочерней таблице, ссылающиеся на исходное значение первичного ключа, то значения их внешних ключей станут некорректными, что соответствует *нарушению ссылочной целостности*.
- *Удаление строки в родительской таблице*. При удалении строки в родительской таблице удаляется значение первичного ключа. Если есть строки в дочерней таблице, ссылающиеся на удаляемую строку, то значения их внешних ключей станут некорректными. Удаление строк в родительской таблице *может привести к нарушению ссылочной целостности*.

Для дочерней таблицы

- *Вставка строки в дочернюю таблицу*. Нельзя вставить строку в дочернюю таблицу, если вставляемое значение внешнего ключа не-

корректно. Вставка строки в дочернюю таблицу *может привести к нарушению ссылочной целостности*. Для обеспечения ссылочной целостности необходимо убедиться, что значение внешнего ключа новой строки дочерней таблицы равно некоторому конкретному значению, присутствующему в атрибуте первичного ключа одной из строк родительской таблицы.

- *Обновление внешнего ключа в строке дочерней таблицы.* Обновление внешнего ключа в дочерней таблице может привести к появлению некорректного значения внешнего ключа, что является *нарушением ссылочной целостности*. Для сохранения ссылочной целостности необходимо убедиться, что значение внешнего ключа в обновленной строке дочерней таблицы равно некоторому конкретному значению, присутствующему в поле первичного ключа одной из строк родительской таблицы.
- *Удаление строки из дочерней таблицы*. При удалении строки из дочерней таблицы никаких *нарушений ссылочной целостности не происходит.*

Таким образом, ссылочная целостность в принципе может быть нарушена при выполнении одной из четырех операций:

- обновление первичного ключа в строке родительской таблицы;
- удаление строки в родительской таблице;
- вставка строки в дочернюю таблицу;
- обновление внешнего ключа в строке дочерней таблицы.

Существуют две основные стратегии поддержания ссылочной целостности:

- NO ACTION (НЕТ ДЕЙСТВИЙ) – не разрешать выполнение операции, приводящей к нарушению ссылочной целостности. Это самая простая стратегия, требующая только проверки, имеются ли строки в дочерней таблице, связанные с некоторой строкой в родительской таблице;
- CASCADE (КАСКАДИРОВАТЬ) – разрешить выполнение требуемой операции, но внести при этом необходимые поправки в других таблицах так, чтобы не допустить нарушения ссылочной целостности и сохранить все имеющиеся связи. Изменение начинается в родительской таблице и каскадно выполняется в дочерней таблице. В реализации этой стратегии имеется одна тонкость, заключающаяся в том, что дочерняя таблица сама может быть родительской для неко-

торой третьей таблицы. При этом может дополнительно потребоваться выполнение какой-либо стратегии и для этой связи и т. д. Если при этом какая-либо из каскадных операций (любого уровня) не может быть выполнена, то необходимо отказаться от первоначальной операции и вернуть базу данных в исходное состояние. Это самая сложная стратегия, но она хороша тем, что при этом не нарушается связь между строками родительской и дочерней таблиц.

Эти стратегии стандартны и присутствуют во всех СУБД, в которых имеется поддержка ссылочной целостности.

Можно рассмотреть дополнительные стратегии поддержания ссылочной целостности:

- SET NULL (УСТАНОВИТЬ В NULL) – разрешить выполнение требуемой операции, но все возникающие некорректные значения внешних ключей изменить на неопределенное значение NULL. Эта стратегия имеет два недостатка. Во-первых, для нее требуется допустить использование NULL-значений. Во-вторых, строки дочерней таблицы теряют всякую связь со строками родительской таблицы. Установить, с какой строкой родительской таблицы были связаны измененные строки дочерней таблицы, после выполнения операции уже нельзя;
- SET DEFAULT (УСТАНОВИТЬ ПО УМОЛЧАНИЮ) – разрешить выполнение требуемой операции, но все возникающие некорректные значения внешних ключей изменить на некоторое значение, принятое по умолчанию. Достоинство этой стратегии по сравнению с предыдущей в том, что она позволяет не пользоваться неопределенным значением NULL. Недостатки заключаются в следующем. Во-первых, в родительской таблице должна быть некая строка, первичный ключ которой принят как значение по умолчанию для внешних ключей. В качестве такой «строки по умолчанию» обычно принимают специальную строку, заполненную нулевыми значениями (не NULL-значениями!). Эту строку *нельзя удалять* из родительской таблицы и в этой строке *нельзя изменять* значение первичного ключа. Таким образом, не все строки родительской таблицы становятся равнозначными, поэтому приходится прилагать дополнительные усилия для отслеживания этой неравнозначности. Это плата за отказ от использования неопределенных значений NULL. Во-вторых, как и в предыдущем случае, строки дочерней таблицы теряют всякую связь со стро-

ками родительской таблицы. Установить, с какой строкой родительской таблицы были связаны измененные строки дочерней таблицы, после выполнения операции уже нельзя.

В некоторых реализациях СУБД рассматривается еще одна стратегия поддержания ссылочной целостности:

IGNORE (ИГНОРИРОВАТЬ) – выполнять операции, не обращая внимания на нарушения ссылочной целостности.

Конечно, это не стратегия, а отказ от поддержки ссылочной целостности. В этом случае в дочерней таблице могут появляться некорректные значения внешних ключей и вся ответственность за целостность базы данных ложится на пользователя.

Рассмотрим, как применяются стратегии поддержания ссылочной целостности при выполнении операций модификации базы данных.

*При обновлении первичного ключа в строке родительской таблицы* допустимые стратегии следующие:

NO ACTION (НЕТ ДЕЙСТВИЙ) – не разрешать обновление, если имеется хотя бы одна строка в дочерней таблице, ссылающаяся на обновляемый первичный ключ;

CASCADE (КАСКАДИРОВАТЬ) – выполнить обновление и автоматически каскадно изменить во всех строках дочерней таблицы значения соответствующего внешнего ключа;

SET NULL (УСТАНОВИТЬ В NULL) – выполнить обновление и заменить значения соответствующего внешнего ключа во всех ссылающихся строках дочерней таблицы автоматически на неопределенное значение NULL;

SET DEFAULT (УСТАНОВИТЬ ПО УМОЛЧАНИЮ) – выполнить обновление и заменить значения соответствующего внешнего ключа во всех ссылающихся строках дочерней таблицы автоматически на некоторое значение, принятое по умолчанию;

IGNORE (ИГНОРИРОВАТЬ) – выполнить обновление, не обращая внимания на нарушения ссылочной целостности.

*При удалении строки в родительской таблице* допустимые стратегии следующие:

NO ACTION (НЕТ ДЕЙСТВИЙ) – не разрешать удаление, если имеется хотя бы одна строка в дочерней таблице, ссылающаяся на удаляемую строку;

CASCADE (КАСКАДИРОВАТЬ) – выполнить удаление и каскадно удалить строки в дочерней таблице, ссылающиеся на удаляемую строку;

SET NULL (УСТАНОВИТЬ В NULL) – выполнить удаление и во всех строках дочерней таблицы, ссылающихся на удаляемую строку, изменить значения внешних ключей на неопределенное значение NULL;

SET DEFAULT (УСТАНОВИТЬ ПО УМОЛЧАНИЮ) – выполнить удаление и во всех строках дочерней таблицы, ссылающихся на удаляемую строку, изменить значения внешних ключей на некоторое значение, принятое по умолчанию;

IGNORE (ИГНОРИРОВАТЬ) – выполнить удаление, не обращая внимания на нарушения ссылочной целостности.

*При вставке строки в дочернюю таблицу* допустимые стратегии следующие:

NO ACTION (НЕТ ДЕЙСТВИЙ) – не разрешать вставку, если внешний ключ во вставляемой строке не соответствует ни одному значению первичного ключа родительской таблицы;

SET NULL (УСТАНОВИТЬ В NULL) – вставить строку, но в качестве значения внешнего ключа присвоить не предлагаемое пользователем некорректное значение, а неопределенное значение NULL;

SET DEFAULT (УСТАНОВИТЬ ПО УМОЛЧАНИЮ) – вставить строку, но в качестве значения внешнего ключа занести не предлагаемое пользователем некорректное значение, а некоторое значение, принятое по умолчанию;

IGNORE (ИГНОРИРОВАТЬ) – вставить строку, не обращая внимания на нарушения ссылочной целостности.

*При обновлении внешнего ключа в строке дочерней таблицы* допустимые стратегии следующие:

NO ACTION (НЕТ ДЕЙСТВИЙ) – не разрешать обновление, если внешний ключ в обновляемой строке становится не соответствующим ни одному значению первичного ключа родительской таблицы;

SET NULL (УСТАНОВИТЬ В NULL) – изменить значение внешнего ключа, но в качестве значения внешнего ключа присвоить не предлагаемое пользователем некорректное значение, а неопределенное значение NULL.

SET DEFAULT (УСТАНОВИТЬ ПО УМОЛЧАНИЮ) – изменить значение внешнего ключа, но в качестве значения внешнего ключа присвоить не предлагаемое пользователем некорректное значение, а некоторое значение, принятое по умолчанию;

IGNORE (ИГНОРИРОВАТЬ) – обновить строку, не обращая внимания на нарушения ссылочной целостности.

Преимущества декларативной поддержки ограничений целостности БД следующие:

- простота реализации, так как при создании и изменении таблицы не требуется дополнительного программирования, кроме SQL-запросов;
- централизованное размещение, так как все вводимые данные должны удовлетворять этим ограничениям.

В качестве примера рассмотрим реляционную модель, состоящую из следующих таблиц:

ПРОДАВЕЦ (Код_продавца, ФИО_продавца, Город_продавца, Комиссионные_продавца, Руководитель, План_продаж)

ЗАКАЗ (Номер_заказа, Сумма_заказа, Дата_заказа, Код_продавца, Код_заказчика)

ЗАКАЗЧИК (Код_заказчика, ФИО_заказчика, Город_заказчика, Рейтинг_заказчика, Сумма_кредита)

Между таблицами ПРОДАВЕЦ и ЗАКАЗ существует связь типа 1:М. Внешний ключ – атрибут «Код_продавца». Таблицы ЗАКАЗЧИК и ЗАКАЗ связаны через атрибут внешнего ключа «Код_поставщика». Тип связи один-ко-многим.

Стратегии ссылочной целостности для рассматриваемых таблиц приведены в табл. 3.13.

Таблица 3.13

Стратегии ссылочной целостности таблиц
ПРОДАВЕЦ, ЗАКАЗ и ЗАКАЗЧИК

| Родительская таблица | Дочерняя таблица | Тип связи | Операция | | | |
|---|---|---|---|---|---|---|
| | | | Обновление первичного ключа в строке родительской таблицы | Удаление строки в родительской таблице | Вставка строки в дочернюю таблицу | Обновление внешнего ключа в строке дочерней таблицы |
| Продавец | Заказ | 1:M | CASCADE | NO ACTION | NO ACTION | NO ACTION |
| Заказчик | Заказ | 1:M | CASCADE | NO ACTION | NO ACTION | NO ACTION |

***Процедурная поддержка ограничений целостности*** заключается в использовании триггеров, хранимых процедур и функций. Триггеры, хранимые процедуры и функции хранятся непосредственно в БД.

Не все ограничения целостности можно реализовать декларативно. Примером такого ограничения может служить требование, что атрибут «Количество_сотрудников» таблицы ОТДЕЛ должен содержать количество сотрудников, реально числящихся в подразделении. Для реализации этого ограничения необходимо создать триггер, запускающийся при вставке, обновлении и удалении записей в таблице СОТРУДНИК, который корректно изменяет значение атрибута «Количество_сотрудников». Например, при вставке в таблицу СОТРУДНИК новой строки триггер увеличивает на единицу значение атрибута «Количество_сотрудников», а при удалении строки – уменьшает. Заметим, что при модификации записей в таблице СОТРУДНИК могут потребоваться даже более сложные действия. Действительно, обновление записи в таблице СОТРУДНИК может заключаться в том, что мы переводим сотрудника из одного отдела в другой, меняя значение в атрибуте «Код_отдела». При этом необходимо в старом отделе уменьшить количество сотрудников, а в новом – увеличить.

Процедурные ограничения целостности относятся к ограничениям с отложенной проверкой.

Примером ограничения, которое не может быть проверено немедленно, можно назвать ограничение из выше рассмотренного примера. Это происходит оттого, что транзакция, заключающаяся во вставке нового сотрудника в таблицу СОТРУДНИК, состоит не менее чем из двух операций – вставки строки в таблицу СОТРУДНИК и обновления строки в таблице ОТДЕЛ. Ограничение, безусловно, неверно после первой операции и становится верным после второй операции.

По времени проверки ограничения делятся:

- на немедленно проверяемые;
- с отложенной проверкой.

*Немедленно проверяемые ограничения* проверяются непосредственно в момент выполнения операции, которая может нарушить ограничение. Если ограничение нарушается, то такая операция отвергается. Транзакция, внутри которой произошло нарушение немедленно проверяемого утверждения целостности, обычно откатывается. Декларативные ограничения целостности относятся к ограничениям, которые немедленно проверяются. Например, уникальность первичного ключа проверяется в момент вставки строки в таблицу. Если ограничение нарушается, то такая операция отвергается.

*Ограничения с отложенной проверкой* поддерживаются механизмами транзакций и триггеров.

## 3.4. Пример логического проектирования базы данных

В пункте 2.7 описана база данных «Контроль_товара», предназначенная для контроля поступления материалов и товаров на склад промышленного предприятия. Там же приведено описание выполнения этапа инфологического проектирования, в результате которого была построена концептуальная модель этой базы данных (см. рис. 2.21) и составлены словари данных с описанием сущностей (см. табл. 2.6), типов связей (см. табл. 2.7) и атрибутов сущностей (см. табл. 2.8).

Проведем логическое проектирование базы данных для контроля поступления материалов и товаров на склад промышленного предприятия.

**Шаг 1.** Создание таблиц для логической модели данных.

Этот шаг выполняется с помощью формальных правил преобразования концептуальной модели в реляционную, описанных в п. 3.1.

Преобразование начнем со связи «Оформляет», в которой участвуют сущности ПОСТАВЩИК и ДОКУМЕНТ. По табл. 2.7 видно, что мощность связи «Оформляет» один-ко-многим и класс принадлежности обеих сущностей обязательный. Следовательно, для получения реляционных таблиц надо воспользоваться правилом 4. В результате получим две таблицы:

ПОСТАВЩИК (Код_постав, Название, Регион, Рейтинг)
ДОКУМЕНТ (Ном_док, Дата, Код_постав)

Внешний ключ у этих таблиц – атрибут «Код_постав». Причем таблица ПОСТАВЩИК – родительская, а таблица ДОКУМЕНТ – дочерняя.

Далее преобразуем связь «Содержит» (см. табл. 2.7), в которой участвуют сущности ДОКУМЕНТ и ПРИХОД. В соответствии с табл. 2.7 параметры этой связи совпадают с параметрами связи «Оформляет». Поэтому для преобразования связи «Содержит» потребуется правило 4, согласно которому будет получено две таблицы:

ДОКУМЕНТ (Ном_док, Дата, Код_постав)
ПРИХОД (Ном_док, Цена, Количество, Тип)

Внешний ключ для этих таблиц – атрибут «Ном_док». Причем таблица ДОКУМЕНТ в данном случае родительская и будет содержать атрибуты, которые были получены при преобразовании предыдущей связи «Оформляет».

Аналогичным образом преобразуем связь «Поступает» между таблицами СКЛАД и ПРИХОД:

СКЛАД (Ном_ном, Наименование, Цена, Количество)
ПРИХОД (Ном_док, Ном_ном, Цена, Количество, Тип).

Атрибут «Ном_ном» – внешний ключ таблиц СКЛАД и ПРИХОД.

Обращаем внимание, что в качестве первичного ключа таблицы ПРИХОД используется составной ключ «Ном_док» + «Ном_ном», так как только по номеру документа и номенклатурному номеру товара можно однозначно определить каждую строку этой таблицы.

Осталось преобразовать иерархические связи между сущностями ПОСТАВЩИК, ЗАРУБЕЖНЫЙ_ПОСТАВЩИК и ОТЕЧЕСТВЕННЫЙ_ПОСТАВЩИК. Такое преобразование описано в п. 3.1.3. В результате преобразования получаем три таблицы:

ПОСТАВЩИК (Код_постав, Название, Регион, Рейтинг)
ЗАРУБЕЖНЫЙ_ПОСТАВЩИК (Код_постав, Валюта, Язык)
ОТЕЧЕСТВЕННЫЙ_ПОСТАВЩИК (Код_постав, Форма_собственности)

Внешний ключ этих таблиц – атрибут «Код_постав». Он же является и первичным ключом этих таблиц, что обеспечивает между таблицами связь типа один-к-одному.

Понятно, что таблицы ЗАРУБЕЖНЫЙ_ПОСТАВЩИК и ОТЕЧЕСТВЕННЫЙ_ПОСТАВЩИК между собой не связаны, поэтому значения атрибута «Код_постав» в этих таблицах не должны совпадать.

В итоге реляционная модель выглядит следующим образом:

ПОСТАВЩИК (Код_постав, Название, Регион, Рейтинг)
ДОКУМЕНТ (Ном_док, Дата, Код_постав)
ПРИХОД (Ном_док, Ном_ном, Цена, Количество, Тип)
СКЛАД (Ном_ном, Наименование, Цена, Количество)
ЗАРУБЕЖНЫЙ_ПОСТАВЩИК (Код_постав, Валюта, Язык)
ОТЕЧЕСТВЕННЫЙ_ПОСТАВЩИК (Код_постав, Форма_собственности)

Внешние ключи у этих таблиц – атрибуты «Код_постав», «Ном_док», «Ном_ном».

**Шаг 2.** Проверка таблиц с помощью правил нормализации.

Преобразовав инфологическую модель базы данных в реляционную, необходимо проверить, соответствует ли реляционная модель базы данных третьей нормальной форме. Для этого сначала определим функциональные зависимости, существующие в таблицах базы данных. В табл. 3.14 – 3.19 приведены функциональные зависимости во всех таблицах базы данных.

Таблица 3.14

Функциональные зависимости между атрибутами таблицы ПОСТАВЩИК

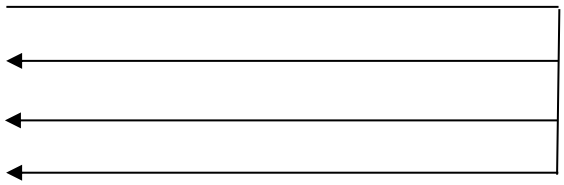
| Наименование атрибутов | Функциональные зависимости |
|---|---|
| Код_постав<br>Название<br>Регион<br>Рейтинг |  |

Таблица 3.15

Функциональные зависимости между атрибутами таблицы ДОКУМЕНТ

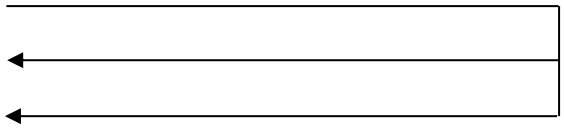
| Наименование атрибутов | Функциональные зависимости |
|---|---|
| Ном_док<br>Дата<br>Код_постав |  |

Таблица 3.16

Функциональные зависимости между атрибутами таблицы ПРИХОД

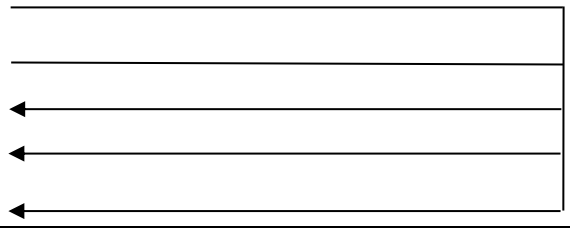
| Наименование атрибутов | Функциональные зависимости |
|---|---|
| Ном_док<br>Ном_ном<br>Цена<br>Количество<br>Тип |  |

Таблица 3.17

Функциональные зависимости между атрибутами таблицы СКЛАД

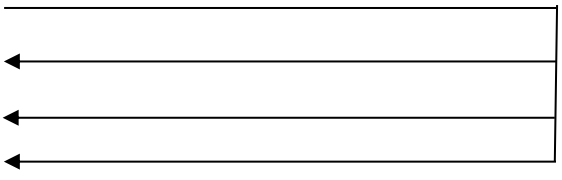
| Наименование атрибутов | Функциональные зависимости |
|---|---|
| Ном_ном<br>Наименование<br>Цена<br>Количество |  |

Таблица 3.18

Функциональные зависимости между атрибутами таблицы ЗАРУБЕЖНЫЙ_ПОСТАВЩИК

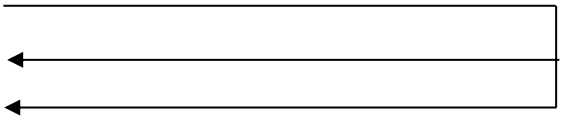
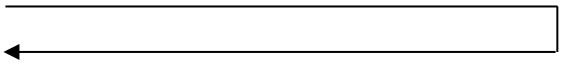
| Наименование атрибутов | Функциональные зависимости |
|---|---|
| Код_постав<br>Валюта<br>Язык |  |

Таблица 3.19

Функциональные зависимости между атрибутами таблицы ОТЕЧЕСТВЕННЫЙ_ПОСТАВЩИК

| Наименование атрибутов | Функциональные зависимости |
|---|---|
| Код_постав<br>Форма_собственности |  |

Можно заметить, что все таблицы и их столбцы содержат только неделимые значения, т. е. таблицы находятся в первой нормальной форме. Таблицы ПОСТАВЩИК, ДОКУМЕНТ, СКЛАД, ЗАРУБЕЖНЫЙ_ПОСТАВЩИК, ОТЕЧЕСТВЕННЫЙ_ПОСТАВЩИК однозначно находятся во второй нормальной форме, так как у этих таблиц первичный ключ не составной. В таблице ПРИХОД первичный ключ составной, но неключевые атрибуты функционально полно зависят от первичного ключа, потому эта таблица находится во второй нормальной форме. Из табл. 3.14 – 3.19 следует, что ни у одной из таблиц транзитивных зависимостей нет, поэтому они находятся в третьей нормальной форме. Кроме того, таблицы, у которых первичный ключ простой, находящиеся в третьей нормальной форме, автоматически находятся и в нормальной форме Бойса – Кодда. В нашей базе данных у всех таблиц, кроме таблицы ПРИХОД, простые первичные ключи. А в таблице ПРИХОД атрибуты составного ключа не зависят от

неключевых атрибутов (см. табл. 3.16). Следовательно, все таблицы проектируемой базы данных находятся в нормальной форме Бойса – Кодда и дальнейшей оптимизации не требуется.

**Шаг 3.** Определение ограничений целостности данных.

На этом шаге логического проектирования необходимо определить ограничения целостности для таблиц базы данных и выбрать стратегии поддержки ссылочной целостности.

Для таблиц могут определяться следующие ограничения:

- запрет на уровне столбца хранения неопределенных значений (NOT NULL);
- уникального значения (UNIQUE KEY);
- первичного ключа (PRIMARY KEY);
- внешнего ключа (FOREIGN KEY);
- значения по умолчанию (DEFAULT);
- значения с проверкой (CHECK).

В табл. 3.20 представлены структура и ограничения таблиц базы данных.

Выбор стратегий поддержки ссылочной целостности определяется спецификой предметной области проектируемой базы данных.

Рассмотрим процесс выбора стратегий на примере связи «Оформляет» (см. рис. 2.21). Для ведения документооборота предприятию необходимо иметь полный набор приходных документов. Поэтому для операции удаления строки в родительской таблице (таблица ПОСТАВЩИК) следует выбрать стратегию NO ACTION, т. е. не разрешать удаление, если имеется хотя бы одна строка дочерней таблицы ДОКУМЕНТ, ссылающаяся на удаляемую строку.

Таблица 3.20

Структура и ограничения таблиц базы данных

| Имя таблицы | Название атрибута | Тип данных и длина | Ключ | Допустимость неопределенных значений | Значения с проверкой | Значения по умолчанию |
|---|---|---|---|---|---|---|
| Поставщик | Код_постав | Целое число | Primary key | NOT NULL | Нет | Нет |
| | Название | Строка длиной до 20 символов | | NOT NULL | Нет | Нет |
| | Регион | Строка длиной до 15 символов | | NULL | Нет | Нет |
| | Рейтинг | Целое число | | NOT NULL | Диапазон от 0 до 100 | 0 |
| Документ | Ном_док | Целое число | Primary key | NOT NULL | Нет | Нет |
| | Дата | Дата | | NOT NULL | Нет | Текущая дата |
| | Код_постав | Целое число | Foreign key | NOT NULL | Нет | Нет |
| Приход | Ном_док | Целое число | Primary key, Foreign key | NOT NULL | Нет | Нет |
| | Ном_ном | Целое число | Primary key, Foreign key | NOT NULL | Нет | Нет |
| | Цена | Денежный формат | | NOT NULL | Нет | Нет |
| | Количество | Целое число | | NOT NULL | Нет | Нет |
| | Тип | Строка длиной до 10 символов | | NOT NULL | Нет | Нет |
| | | | | | | |

*Окончание табл. 3.20*

| Имя таблицы | Название атрибута | Тип данных и длина | Ключ | Допустимость неопределенных значений | Значения с проверкой | Значения по умолчанию |
|---|---|---|---|---|---|---|
| Склад | Ном_ном, | Целое число | Primary key | NOT NULL | Нет | Нет |
| | Наименование | Строка длиной до 20 символов | | NOT NULL | Нет | Нет |
| | Цена | Денежный формат | | NOT NULL | Нет | Нет |
| | Количество | Целое число | | NOT NULL | Нет | Нет |
| Зарубежный_поставщик | Код_постав | Целое число | Primary key, Foreign key | NOT NULL | Нет | Нет |
| | Валюта | Строка длиной до 10 символов | | NOT NULL | Нет | Нет |
| | Язык | Строка длиной до 15 символов | | NOT NULL | Нет | Нет |
| Отечественный_поставщик | Код_постав | Целое число | Primary key, Foreign key | NOT NULL | Нет | Нет |
| | Форма_собственности | Строка длиной до 20 символов | | NOT NULL | Нет | Нет |

Изменения кодов поставщиков на предприятии вряд ли происходят часто, но такое возможно, поэтому для обновления первичного ключа в строке родительской таблицы выбираем стратегию CASCADE.

Для вставки строки в дочернюю таблицу (ДОКУМЕНТ) выбрана стратегия NO ACTION – не разрешать вставку, если внешний ключ во

вставляемой строке не соответствует ни одному значению первичного ключа родительской таблицы. Другие возможные стратегии здесь явно не подходят.

Такая же стратегия выбрана для операции обновления внешнего ключа в строке дочерней таблицы.

Для других связей стратегии ссылочной целостности базы данных выбираются аналогичным образом.

Стратегии ссылочной целостности для таблиц проектируемой базы данных приведены в табл. 3.21.

Таблица 3.21

Стратегии ссылочной целостности таблиц проектируемой базы данных

| Родительская таблица | Дочерняя таблица | Тип связи | Операция | | | |
|---|---|---|---|---|---|---|
| | | | Обновление первичного ключа в строке родительской таблицы | Удаление строки в родительской таблице | Вставка строки в дочернюю таблицу | Обновление внешнего ключа в строке дочерней таблицы |
| Поставщик | Документ | 1:M | CASCADE | NO ACTION | NO ACTION | NO ACTION |
| Документ | Приход | 1:M | NO ACTION | CASCADE | NO ACTION | NO ACTION |
| Склад | Приход | 1:M | NO ACTION | NO ACTION | NO ACTION | NO ACTION |
| Поставщик | Зарубежный_поставщик | 1:1 | CASCADE | CASCADE | NO ACTION | NO ACTION |
| Поставщик | Отечественный_поставщик | 1:1 | CASCADE | CASCADE | NO ACTION | NO ACTION |

В табл. 3.20 и 3.21 приведены декларативные ограничения данных. Однако в базах данных встречаются ограничения, которые невозможно реализовать с помощью декларативных ограничений, например когда требуют выполнить несколько операций модификации. В нашем случае таким ограничением является следующий факт: при поступлении товара на предприятие его количество на складе необходимо увеличить на количество поступавшего товара. Для полученной структуры базы данных это ограничение формулируется

следующим образом: при вставке новой строки в таблицу ПРИХОД необходимо увеличить значение атрибута «Количество» таблицы СКЛАД на значение атрибута «Количество» таблицы ПРИХОД. Такое ограничение можно реализовать с помощью триггеров, о создании и использовании которых речь пойдет в следующей главе.

## Вопросы для самопроверки

1. Какое значение имеет в базах данных понятие «целостность данных»?
2. Какие средства используются в БД для поддержания целостности?
3. Как классифицируются ограничения целостности?
4. Какие существуют способы реализации ограничений целостности?
5. Как обеспечивается целостность сущности?
6. Какие средства используются при определении ограничений в рамках декларативной поддержки целостности?
7. Какие виды ограничений используются при декларативной поддержке целостности?
8. Какие условия гарантируют выполнение требования целостности сущностей?
9. Какие подходы возможны относительно неопределенности внешнего ключа?
10. При каких операциях может быть нарушена ссылочная целостность?
11. Какие основные стратегии используются для поддержки ссылочной целостности?
12. Какие дополнительные стратегии используются для поддержки ссылочной целостности?
13. Какие стратегии поддержки ссылочной целостности применимы при обновлении первичного ключа в строке родительской таблицы?
14. Какие стратегии поддержки ссылочной целостности применимы при удалении строки в родительской таблице?
15. Как делятся ограничения целостности по времени проверки?

## Практические задания

Определите стратегии поддержки ссылочной целостности для таблиц, созданных в соответствии с практическим заданием п. 3.1.

# Глава 4. ФИЗИЧЕСКОЕ ПРОЕКТИРОВАНИЕ БАЗ ДАННЫХ

Физическое проектирование – третий и последний этап создания проекта базы данных. В результате выполнения логического этапа проектирования получают логическую модель данных, включающую в себя реляционную схему и словарь данных. И хотя логическая модель не зависит от конкретной целевой СУБД, она создается с учетом выбранной модели хранения данных. Напомним, что в данном пособии рассматривается реляционная модель.

Основная цель физического проектирования базы данных – описание способа физической реализации логического проекта базы данных. Описание должно включать в себя основные таблицы, файловую организацию, индексы, обеспечивающие эффективный доступ к данным, а также все соответствующие ограничения целостности и средства защиты.

В случае реляционной модели данных под физическим проектированием подразумеваются следующие шаги.

Шаг 1. Выбор СУБД.

Шаг 2. Перенос логической модели в среду выбранной СУБД.

Шаг 2.1. Описание таблиц базы данных.

Шаг 2.2. Проектирование общих ограничений.

Шаг 3. Проектирование организации файлов и индексов.

Шаг 3.1. Анализ транзакции.

Шаг 3.2. Выбор способов организации файлов.

Шаг 3.3. Выбор индексов.

Шаг 4. Проектирование представлений пользователей.

О создании и использовании представлений говорится в учебно-практическом пособии [12].

Шаг 5. Проектирование механизмов защиты.

*Примечание.* Вопросы проектирования механизмов защиты данных относятся к вопросам администрирования БД, которые в силу их многогранности и специфичности в данном пособии не рассматриваются.

Результаты этапа физического проектирования документируются в форме схемы хранения на языке определения данных (DDL).

Из перечисленных выше шагов физического проектирования базы данных становится ясно, в чем отличие физического этапа проек-

тирования от логического этапа. На этапе логического проектирования отвечают на вопрос: «ЧТО должно быть сделано?». А на этапе физического проектирования отвечают на вопрос: «КАК это должно быть сделано?».

Между логическим и физическим проектированием существует постоянная обратная связь, так как решения, принимаемые на этапе физического проектирования с целью повышения производительности системы, способны повлиять на структуру логической модели данных.

Приступая к физическому проектированию базы данных, прежде всего необходимо выбрать конкретную реляционную СУБД.

### 4.1. Выбор СУБД

Система управления базой данных представляет собой набор программных средств, посредством которого осуществляется управление базой данных и доступ к данным.

Характеристики систем автоматизированной обработки информации во многом определяются свойствами их СУБД, поэтому выбор СУБД при разработке информационных систем играет важную роль. Следует учитывать много факторов, влияющих на выбор СУБД. Рассмотрим некоторые из них.

- Тип модели данных, которую поддерживает данная СУБД, её адекватность потребностям рассматриваемой предметной области.
- Масштабируемость, т. е. необходимость учитывать, сможет ли данная система соответствовать росту информационной системы. Причем рост может проявляться в увеличении числа пользователей, объема хранимых данных, объема обрабатываемой информации.
- Доступность, т. е. постоянная возможность получения ответа на запрос. При этом СУБД должна обладать необходимым быстродействием, чтобы представлять ценность для пользователей.
- Надежность, т. е. минимальная вероятность сбоев, а также наличие средств восстановления данных после сбоев, резервирование и дублирование данных.
- Управляемость, т. е. простота администрирования и конфигурирования, а нередко и наличие средств автоматического конфигурирования.

• Наличие средств защиты данных от потери и несанкционированного доступа, т. е. для обеспечения защиты и целостности данных при выборе СУБД необходимо обращать внимание на следующие возможности:

– контроль доступа – важный фактор, позволяющий избежать несанкционированного доступа к данным;

– контроль параллельной обработки – средства поддержания целостности данных при многопользовательском режиме работы;

– управление представлениями данных – автоматические средства ограничения данных таблицы, к которым пользователь имеет право обращаться;

– средства шифровки – могут иметь большое значение для организаций, обладающих очень уязвимыми данными;

– средства резервного копирования и восстановления – очень важны для эффективного функционирования системы базы данных.

• Требования к техническим средствам, т. е. следует учитывать, что для установки СУБД на компьютере к отдельным его устройствам (например, к оперативной памяти, накопителю на жестком магнитном диске) могут быть предъявлены особые требования.

• Техническая поддержка, т. е. необходимо учитывать, что различные СУБД документированы по-разному: более или менее тщательно. По-разному предоставляется и техническая поддержка.

• Стоимость СУБД и дополнительного программного обеспечения.

Как правило, отсутствие какого-либо из этих признаков приводит к тому, что даже у неплохой по другим потребительским свойствам СУБД область применения оказывается весьма ограниченной. Именно поэтому лидеры рынка корпоративных СУБД стремятся к производству продуктов, удовлетворяющих всем вышеуказанным требованиям. Кроме того, как правило, подобные продукты существуют для нескольких платформ, а нередко и в разных редакциях, предназначенных для решения разнообразных задач или обслуживания различного количества данных и пользователей.

## 4.2. Перенос логической модели в среду выбранной СУБД

### *4.2.1. Описание таблиц базы данных*

Цель этого шага состоит в представлении в выбранной СУБД таблиц, определенных в логической модели данных.

На этапе физического проектирования описание каждой таблицы данных логической модели включает в себя следующие элементы:

- имя таблицы;
- список атрибутов;
- первичный ключ (PK) и (если таковые существуют) альтернативные ключи (AK);
- внешние (FK) ключи;
- стратегии (правила) ссылочной целостности для любых внешних ключей.

Для каждого атрибута в словаре данных должна присутствовать следующая информация:

- определение его домена, включающее в себя указание типа данных, размерность внутреннего представления атрибута и любые требуемые ограничения на допустимые значения;
- допустимость значения NULL для данного атрибута;
- принимаемое по умолчанию значение атрибута.

Вся указанная выше информация содержится в словаре данных, полученном на этапе логического проектирования. Различие состоит в том, что при физическом проектировании необходимо учитывать возможности выбранной СУБД. Например, требуется указывать для каждого атрибута конкретный тип данных, который имеется в СУБД. А при выборе стратегии поддержки ссылочной целостности указывать надо только те стратегии, которые имеются в выбранной СУБД.

В качестве примера рассмотрим базу данных «Контроль_товара», логическая модель которой представлена в п. 3.4. Выберем в качестве целевой системы управления базой данных СУБД MS SQL Server. Описание таблиц учебной базы данных после переноса их в СУБД MS SQL Server приведено в табл. 4.1.

Таблица 4.1

Словарь учебной базы данных «Контроль_товара»
для СУБД MS SQL Server

| Имя таблицы | Название атрибута | Тип данных и длина | Декларативные ограничения | | | |
|---|---|---|---|---|---|---|
| | | | Ключ | Ограничение значения NULL | Значения с проверкой CHECK | Ограничение на значение по умолчанию DEFAULT |
| Поставщик | Код_постав | Int | Primary key | NOT NULL | Нет | Нет |
| | Название | VarChar (20) | | NOT NULL | Нет | Нет |
| | Регион | VarChar (15) | | NULL | Нет | Нет |
| | Рейтинг | Int | | NOT NULL | Between 0 AND 100 | 0 |
| Документ | Ном_док | Int | Primary key | NOT NULL | Нет | Нет |
| | Дата | DateTime | | NOT NULL | Нет | GetDate() |
| | Код_постав | Int | Foreign key | NOT NULL | Нет | Нет |
| Приход | Ном_док | Int | Primary key, Foreign key | NOT NULL | Нет | Нет |
| | Ном_ном | Int | Primary key, Foreign key | NOT NULL | Нет | Нет |
| | Цена | Money | | NOT NULL | Нет | Нет |
| | Количество | Int | | NOT NULL | Нет | Нет |
| | Тип | VarChar (10) | | NOT NULL | Нет | Нет |
| | | | | | | |

*Окончание табл. 4.1*

| Имя таблицы | Название атрибута | Тип данных и длина | Декларативные ограничения | | | |
|---|---|---|---|---|---|---|
| | | | Ключ | Ограничение значения NULL | Значения с проверкой CHECK | Ограничение на значение по умолчанию DEFAULT |
| Склад | Ном_ном | Int | Primary key | NOT NULL | Нет | Нет |
| | Наименование | VarChar (20) | | NOT NULL | Нет | Нет |
| | Цена | Money | | NOT NULL | Нет | Нет |
| | Количество | Int | | NOT NULL | Нет | Нет |
| Зарубежный_ поставщик | Код_ постав | Int | Primary key, Foreign key | NOT NULL | Нет | Нет |
| | Валюта | VarChar (10) | | NOT NULL | Нет | Нет |
| | Язык | VarChar (15) | | NOT NULL | Нет | Нет |
| Отечественный_ поставщик | Код_ постав | Int | Primary key, Foreign key | NOT NULL | Нет | Нет |
| | Форма_ собственности | VarChar (20) | | NOT NULL | Нет | Нет |

Рассмотрим теперь, как будут выглядеть ограничения ссылочной целостности для проектируемой базы данных в СУБД MS SQL Server. В этой системе управления базой данных присутствуют все стратегии (за исключением стратегии IGNORE) поддержки ссылочной целостности при выполнении операций с родительскими таблицами. А вот при вставке строки в дочернюю таблицу и обновлении внешнего ключа в строке дочерней таблицы СУБД MS SQL Server не разрешает делать такую вставку и такое обновление, если внешний

ключ во вставляемой или обновляемой строке не соответствует ни одному значению первичного ключа родительской таблицы. Изменить эту стратегию средствами СУБД нельзя. Стратегии ссылочной целостности для таблиц проектируемой базы данных с учетом особенностей СУБД MS SQL Server выглядят, как это показано в табл. 4.2.

Таблица 4.2

Стратегии ссылочной целостности таблиц проектируемой базы данных «Контроль_товара» в СУБД MS SQL Server

| Родительская таблица | Дочерняя таблица | Тип связи | Операция | |
|---|---|---|---|---|
| | | | Обновление первичного ключа в строке родительской таблицы ON UPDATE | Удаление строки в родительской таблице ON DELETE |
| Поставщик | Документ | 1:M | CASCADE | NO ACTION |
| Документ | Приход | 1:M | NO ACTION | CASCADE |
| Склад | Приход | 1:M | NO ACTION | NO ACTION |
| Поставщик | Зарубежный_поставщик | 1:1 | CASCADE | CASCADE |
| Поставщик | Отечественный_поставщик | 1:1 | CASCADE | CASCADE |

Таким образом, табл. 4.1 и 4.2 содержат информацию о всех таблицах спроектированной базы данных, их связях друг с другом, т. е. представляют собой описание проекта базы данных. Используя это описание, легко можно реализовать базу данных в СУБД MS SQL Server.

### *4.2.2. Команда создания таблицы CREATE TABLE*

Ранее мы получили описание таблиц БД в табличной форме в виде словаря данных.

Существует другой способ представления описания таблиц – это SQL-скрипт.

SQL-скрипт объекта базы данных – это команда языка SQL, с помощью которой создается этот объект, сохраненная в текстовом

файле. Иными словами, это сохраненный в текстовом файле с расширением .sql запрос, который представляет собой команду (или команды) создания объекта (или объектов).

Так как SQL-скрипт – это обычный текстовый файл, его можно создать вручную при проектировании объекта БД, сохранить команду SQL в файл и добавлять в него по мере необходимости другие команды SQL.

Однако также возможно автоматически сгенерировать SQL-скрипты объектов базы данных специальными инструментами.

В пособии мы не будем рассматривать эти специальные инструментальные средства создания SQL-скриптов, а изучим команду языка SQL CREATE TABLE. С помощью команды CREATE TABLE можно создать базовую таблицу.

Все последующие описания отдельных элементов логической схемы БД выполним на языке Transact SQL, который используется в СУБД MS SQL Server.

Упрощенный синтаксис команды CREATE TABLE выглядит следующим образом:

CREATE TABLE <имя_таблицы>
( <имя поля1> <тип данных> [ (<размер>) ]
[<ограничения целостности поля1>,…]
[, <имя поля2> <тип данных> [ (<размер>) ]
[<ограничения целостности поля2>],...]
[, <ограничения целостности таблицы>])

*Имя таблицы* играет роль ее идентификатора в базе данных, поэтому оно должно быть уникальным.

После имени таблицы в круглых скобках указываются параметры всех столбцов.

*Имя поля (атрибута, столбца)* таблицы – это правильный идентификатор языка SQL.

В качестве *типа данных* можно использовать один из типов данных, существующих в СУБД. Тип данных, для которого обязательно должен быть указан размер, – это тип CHAR. Реальное количество символов, которое может находиться в поле, изменяется от нуля (если в поле содержится NULL-значение) до заданного в команде CREATE TABLE максимального значения.

Если размер поля не указан, то принимается значение, принятое в данной СУБД по умолчанию для указанного типа. Для всех СУБД размер поля типа CHAR, VARCHAR, NCHAR и NVARCHAR по умолчанию равен единице, поэтому для полей такого типа размер указывать обязательно. Точность для числовых типов по умолчанию равна нулю.

В самом простом виде команда CREATE TABLE должна содержать как минимум имя таблицы, имена и типы столбцов, так как такие параметры команды как размер, ограничения целостности поля и ограничения целостности таблицы необязательны.

К *ограничениям целостности поля* относят:

- ограничение значения NULL;
- PRIMARY KEY – первичный ключ (обязательный, уникальный и единственный на таблицу);
- UNIQUE – уникальность значения поля в пределах столбца таблицы;
- CHECK (<условие>) – условие, которому должно удовлетворять значение поля;
- ограничение на значение по умолчанию DEFAULT <выражение>. Строго говоря, ограничение DEFAULT не имеет ограничительного свойства, так как оно не ограничивает значения, вводимые в поле, а просто конкретизирует значение поля в случае, если оно не было задано;
- REFERENCES <имя таблицы> [(<имя столбца>)] – внешний ключ.

Ограничения целостности имеют приоритет над триггерами, правилами и значениями по умолчанию.

Для ограничений целостности можно задавать имена:

CONSTRAINT <имя> <ограничение целостности>

Если имя не задано, то система создаст имя автоматически. Лучше задавать ограничениям осмысленные имена. В этом случае при выдаче системой сообщения о нарушении установленного ограничения будет указано его имя, а это упрощает обнаружение ошибок.

*Ограничения целостности таблицы* – то же, что и для поля.

Общие ограничения целостности указываются через запятую после последнего поля.

## Примеры создания таблиц с ограничениями

### *Пример создания таблиц с ограничениями PRIMARY KEY и NULL*

Рассмотрим несколько способов определения ограничений.

*Первый способ* подразумевает определение PRIMARY KEY на уровне столбца:

```
CREATE TABLE Table1 (
    Column1 INT IDENTITY(1,1) NOT NULL PRIMARY KEY,
    Column2 VARCHAR(20) NOT NULL,
    Column3 DATE
 )
```

В описании столбца Column3 отсутствует ограничение NOT NULL. Это значит, что для столбца будет действовать ограничение NOT, которое устанавливается по умолчанию. Таким образом, если у нас запрещены значения NULL в столбце, то этот столбец обязательный к заполнению, а если у нас разрешены значения NULL, то столбец можно и не заполнять.

Во *втором способе* для определения ограничения PRIMARY KEY используется ключевое слово CONSTRAINT на уровне столбца:

```
CREATE TABLE TestTable2 (
    Column1 INT IDENTITY(2,3) NOT NULL CONSTRAINT PK_Column1_T2 PRIMARY KEY,
    Column2 VARCHAR(20) NOT NULL,
    Column3 DATE
 )
```

*Третий способ* заключается в определении ограничения на уровне таблицы. Определение ограничения находится после всех столбцов таблицы и содержит имя ограничения, тип ограничения и имя столбца, который будет выполнять роль первичного ключа (Column1):

```
CREATE TABLE Table3 (
    Column1 INT IDENTITY(10,5) NOT NULL,
    Column2 VARCHAR(20) NOT NULL,
    Column3 DATE,
    CONSTRAINT PK_Column1_T3 PRIMARY KEY (Column1)
 )
```

Во всех трех способах описания ограничений команда CREATE TABLE определяет, что столбец Column1 должен быть полем первичного ключа с автоматическим приращением в поле первичного ключа. Система управления базами данных MS SQL Server использует ключевое слово IDENTITY для выполнения функции автоматического приращения.

В приведенных выше примерах в таблице Table1 начальное значение поля Column1 равно единице, и оно будет увеличиваться на единицу для каждой новой записи, в таблице Table2 поле Column1 должно начинаться со значения два и увеличиваться на три, а в таблице Table3 значение поля Column1 начинается с 10 и увеличивается на пять.

При вставке новой строки в таблицу не нужно будет указывать в команде INSERT значение для столбца Column1. Уникальное новое значение будет добавлено автоматически. Обращаем внимание, что сервер не гарантирует непрерывности значений – в реальных данных в таблице могут появляться разрывы.

Составные ключи указываются через запятую после последнего поля:

```
CREATE TABLE Table4 (
    Column1 INT NOT NULL,
    Column2 INT NOT NULL,
    Column3 DATE,
    CONSTRAINT PK_T3 PRIMARY KEY (Column1, Column2)
 )
```

### *Пример создания таблиц с ограничениями CHECK, UNIQUE, DEFAULT*

Для создания таких ограничений можно использовать описанные выше способы.

```
CREATE TABLE Table5 (
    Column1 INT NOT NULL PRIMARY KEY,
    Column2 VARCHAR(20) NOT NULL,
    Column3 DATE UNIQUE DEFAULT (GetDate()),
    Column4 INT NOT NULL CONSTRAINT C_Table5_C1
CHECK (Column4 > 0),
```

```
    CONSTRAINT C_Table5_C2 CHECK (Column1 > Column4),
    CONSTRAINT U_Table5_C3 UNIQUE (Column2)
)
```

Ограничения NOT NULL и PRIMARY KEY были рассмотрены в предыдущих примерах.

Для третьего столбца Column3 задано значение по умолчанию GetDate() с помощью ключевого слова DEFAULT. Функция GetDate() возвращает текущее время и дату. Ограничение DEFAULT помещает значение в поле, если оно не было указано в команде INSERT. Оно относится только к команде добавления записи (INSERT) и не срабатывает во время изменения значений полей (команда UPDATE).

В четвертом столбце определено проверочное ограничение C_Table5_C1, которое подразумевает, что столбец Column4 может содержать только положительные значения. Если будет сделана попытка вставить строку, например, со значением Column1 = –5, СУБД MS SQL Server не разрешит это сделать и выдаст ошибку.

Команда CREATE TABLE содержит два ограничения на уровне таблицы. В первом ограничении с именем C_Table5_C2 задействованы два столбца, и оно означает, что в строках таблицы Table5 значение столбца Column1 всегда должно быть больше значения столбца Column4. Таким образом, для столбца Column4 с помощью двух ограничений C_Table5_C1 и C_Table5_C2 задано ограничение (Column1 <> 0) AND (Column1 > Column4). Второе ограничение с именем U_Table5_C3 позволяет исключить повторяющиеся значения в столбце Column2.

В отличие от ограничения PRIMARY KEY, для таблицы можно задать несколько ограничений UNIQUE. В рассмотренном примере два столбца Column2 и Column3 имеют ограничение UNIQUE.

### *Пример создания ограничения FOREIGN KEY*

Создадим две таблицы: первая будет родительской, а вторая – дочерней, которая ссылается на первую. Тип связи один-ко-многим.

```
CREATE TABLE Table6 (
    Column1 INT NOT NULL CONSTRAINT PK_Table6 PRIMARY KEY,
    Column2 VARCHAR(20) NOT NULL
)
```

```
CREATE TABLE Table7 (
    Column3 INT NOT NULL PRIMARY KEY,
    Column4 INT NOT NULL,
    CONSTRAINT FK_Table7 FOREIGN KEY (Column4)
REFERENCES Table6 (Column1)
    ON DELETE CASCADE
 )
```

В данном примере сначала была создана родительская таблица Table6, а затем – дочерняя таблица Table7. При создании последней определен внешний ключ, т. е. задано ограничение FOREIGN KEY с FK_Table7. Для этого в ограничении FOREIGN KEY указывается ключевое слово CONSTRAINT, имя ограничения, тип, столбец, который будет ссылаться на ключ в другой таблице. Далее после ключевого слова REFERENCES указывается таблица, на которую делается ссылка, и в скобках приводится название столбца, который и будет первичным ключом родительской таблицы. В этом ограничении приведены стратегии ссылочной целостности при выполнении операций удаления строки из родительской таблицы или обновления первичного ключа родительской таблицы. По умолчанию в СУБД MS SQL Server для этих операций устанавливается стратегия NO ACTION.

Для того чтобы изменить действие, заданное по умолчанию, надо в инструкции определения ограничения команды ON DELETE и ON UPDATE указать те действия, которые будут выполнены в случае удаления ключа, и действия, которые будут выполнены, если этот ключ будет обновлен. В нашем примере указана стратегия CASCADE, которая используется при изменении значения ключа родительской таблицы (ON UPDATE).

Так как стратегия при удалении строки, а значит, и ключа родительской таблицы (ON DELETE) не указана, то это означает, что остается стратегия, установленная по умолчанию.

Обратите внимание:

- общие ограничения целостности и составные ключи указываются через запятую после последнего поля;
- если внешний ключ ссылается на первичный ключ другой таблицы, имена полей первичного ключа можно не указывать;
- типы полей внешнего ключа должны совпадать с типами полей первичного (или уникального) ключа, на который он ссылается;

• если внешний ключ составной, список полей, входящий в ключ, указывается после перечисления всех полей таблицы с ключевым словом FOREIGN KEY.

В приложении приведен набор команд CREATE TABLE для создания проектируемой базы данных.

## Вопросы для самопроверки

1. С помощью какой команды можно создать таблицу базы данных?
2. Для какого типа данных при создании таблицы обязательно должен быть указан размер?
3. Что означает свойство IDENTITY?
4. Какие ограничения целостности могут быть заданы при создании таблицы?
5. Какие способы задания ограничений вы знаете?
6. Что означает ограничение PRIMARY KEY?
7. Как задать ограничения на внешний ключ?
8. Каковы особенности первичных и внешних ключей?
9. Что означает ограничение NULL?
10. Для чего используется ограничение CHECK?

## Практические задания

С помощью команды CREATE TABLE опишите таблицу, отобразив все ограничения для обеспечения целостности таблицы:

а) КЛИЕНТ (Код_клиента, Наименование, Годовой_доход, Тип_заказчика)

б) ОТГРУЗКА (Номер_отгрузки, Код_клиента, Вес, Номер_грузовика, Город, Дата).

в) ВОДИТЕЛЬ (Номер_удостоверения, ФИО, Адрес)

г) АВТОМОБИЛЬ (Регистрационный_номер, Марка, Цвет, Номер_водительского_удостоверения)

д) СТУДЕНТЫ (ФИО, Дата_рождения, Номер_группы, Номер_зачетной_книжки)

е) ЭКЗАМЕНЫ (Номер_зачетной_книжки, Дисциплина, Оценка, Зачет)

ж) ПРОДАВЕЦ (Код_продавца, Имя, Город, Комиссионные)

з) ЗАКАЗЧИК (Код_покупателя, ФИО, Рейтинг, Город, Код_продавца)

и) ПОКУПКА (Номер, Сумма, Дата, Код_продавца, Код_покупателя)

2. Опишите заново таблицы, усовершенствовав их определение ограничениями внешних ключей:

А) КЛИЕНТ (Код_клиента, Наименование, Годовой_доход, Тип_заказчика)

ОТГРУЗКА (Номер_отгрузки, Код_клиента, Вес, Номер_грузовика, Город, Дата)

Б) ВОДИТЕЛЬ (Номер_удостоверения, ФИО, Адрес)

АВТОМОБИЛЬ (Регистрационный_номер, Марка, Цвет, Номер_водительского_удостоверения)

В) СТУДЕНТЫ (ФИО, Дата_рождения, Номер_группы, Номер_зачетной_книжки)

ЭКЗАМЕНЫ (Номер_зачетной_книжки, Дисциплина, Оценка, Зачет)

Г) ПРОДАВЕЦ (Код_продавца, Имя, Город, Комиссионные)

ЗАКАЗЧИК (Код_покупателя, ФИО, Рейтинг, Город, Код_продавца)

ПОКУПКА (Номер, Сумма, Дата, Код_продавца, Код_покупателя)

### 4.3. Процедурная поддержка ограничений целостности базы данных

Выше были рассмотрены вопросы, связанные с представлением в базе данных декларативных ограничений целостности данных. Средства декларативной поддержки целостности определяют ограничения на значения атрибутов, целостность сущностей (потенциальные ключи таблиц) и ссылочную целостность (целостность внешних ключей).

Процедурная поддержка ограничений целостности заключается в использовании программного кода, реализуемого в СУБД в виде так называемых хранимых процедур и триггеров. Процедуры и триггеры позволяют выполнить в СУБД обработку гораздо более изощренных

ограничений целостности данных, не реализуемых с использованием стандартных встроенных средств декларативной поддержки целостности. Например, при добавлении данных покупки товара необходимо внести данные в таблицу заказов. Однако перед этим надо проверить, а есть ли покупаемый товар в наличии. Возможно, при этом понадобится проверить еще ряд дополнительных условий. То есть фактически процесс покупки товара охватывает несколько действий, которые должны выполняться в определенной последовательности.

### *4.3.1. Процедурная логика языка Transact-SQL*

Изначально язык SQL был основным средством работы пользователя с базой данных и позволял выполнять следующий набор операций:

- создание в базе данных новой таблицы;
- добавление в таблицу новых записей;
- изменение записей;
- удаление записей;
- выборку записей из одной или нескольких таблиц;
- изменение структур таблиц.

Со временем язык SQL обогатился новыми командами, обеспечил возможность описания новых объектов базы данных, таких как представления, хранимые процедуры, триггеры, индексы, и управления ими. Вследствие чего он стал приобретать черты, свойственные процедурным языкам программирования.

Далее рассмотрим ту часть основ языка Transact-SQL, которая подразумевает написание кода для реализации некоего функционала (например, в процедуре, триггере или функции), а не просто какого-то запроса к базе данных.

***4.3.1.1. Определение и использование переменных.*** Переменные нужны для того, чтобы сохранить на время какие-то данные, а затем их использовать. Существует две разновидности переменных в SQL – локальные и глобальные. *Локальные переменные* существуют только в пакете, сценарии, хранимой процедуре, триггере, а *глобальные* используются для получения информации о сервере или какой-либо информации во всей базе данных.

К глобальным переменным в СУБД MS SQL Server относятся:

- @@ROWCOUNT – хранит количество записей, обработанных предыдущей командой;
- @@ERROR – возвращает код ошибки для последней команды;
- @@SERVERNAME – имя локального SQL-сервера;
- @@VERSION – номер версии СУБД MS SQL Server;
- @@IDENTITY – последнее значение счетчика, используемое в операции вставки (INSERT).

Локальные переменные определяются с помощью команды DECLARE:

DECLARE @имя_переменной тип_данных
[, @имя_переменной тип_данных, . . . ]

При создании локальной переменной необходимо указать ее имя, которое начинается с символа @, и тип данных. На тип данных локальных переменных накладывается одно ограничение: они не могут иметь тип text или image. С помощью одной команды DECLARE можно объявить несколько переменных.

При объявлении переменной она получает неопределенное значение NULL.

Система управления базами данных MS SQL Server поддерживает объявление и инициализацию переменных в рамках одной команды, как показано ниже.

DECLARE @k INT = 5;

Для присвоения значения локальной переменной можно использовать команду SET или команду SELECT:

SET <@имя_переменной> = <выражение>/команда [SELECT]
FROM список_таблиц
WHERE <выражение>]
[GROUP BY...]
[HAVING ...]
[ORDER BY...]

SELECT <имя_переменной>=<выражение>/команда[SELECT]
FROM список_таблиц
WHERE <выражение>]
[GROUP BY...]
[HAVING ...]
[ORDER BY...]

Рассмотрим пример определения двух локальных переменных. Они применяются для подсчета количества строк в таблице. Функция CONVERT используется для конвертирования числа строк в текстовый формат данных.

```
DECLARE @mynum INT, @mychar CHAR (2)
SELECT @mynum = COUNT(*) FROM Поставщик
SET @mychar = CONVERT(char(2),@mynum)
SELECT @mynum AS ‘Переменная mynum’, @mychar AS ‘Переменная mychar’
```

Результат выполнения этого программного кода приведен на рис. 4.1.

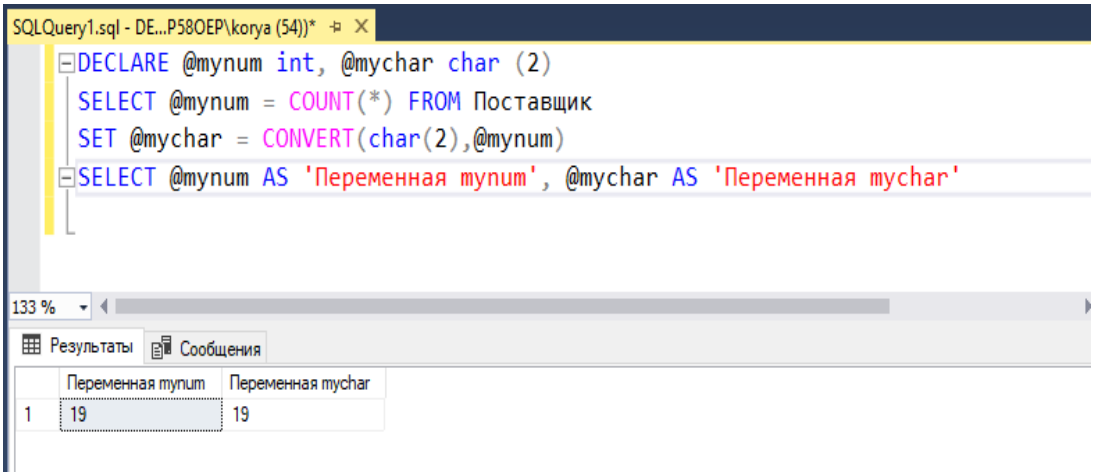



Рис. 4.1. Определение и использование переменных

В инструкции команды SET можно также использовать составной оператор присваивания для локальной переменной.

Операция составного присваивания состоит из простой операции присваивания, скомбинированной с какой-либо другой бинарной операцией:

+= сложение и присваивание;

-= вычитание и присваивание;

*= умножение и присваивание;

/= деление и присваивание;

%= остаток от деления и присваивание.

При составном присваивании вначале выполняется действие, соответствующее бинарной операции, а затем результат присваивается левому операнду.

Рассмотрим пример, в котором используется составной оператор присваивания. Вначале создается локальная переменная с именем @NewVar, умножается на два, после отображается новое значение переменной с помощью команды SELECT:

```
DECLARE @NewVar INT = 5
SET @NewVar *= 2
SELECT @NewVar
```

На рис. 4.2 представлен результат выполнения составного оператора присваивания.

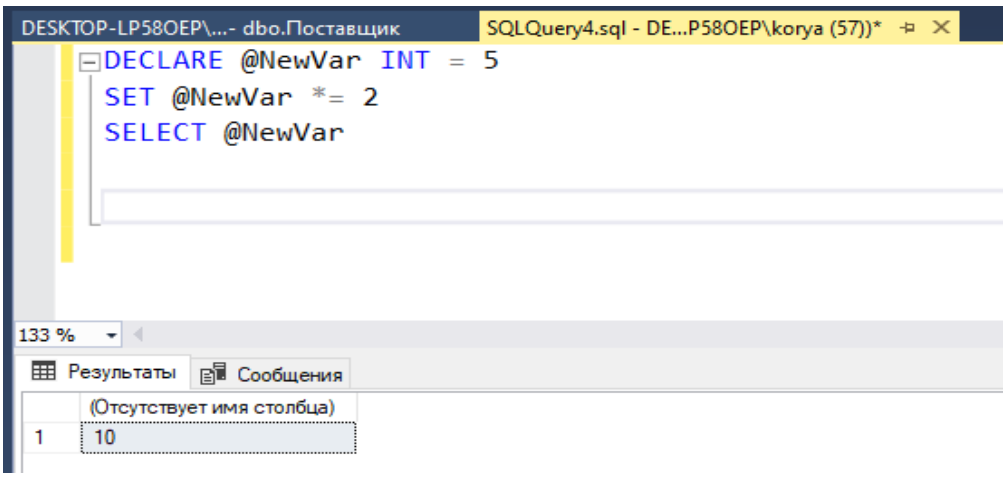



Рис. 4.2. Результат выполнения составного оператора присваивания

Тот же самый результат будет получен, если использовать обычный оператор присваивания:

```
DECLARE @NewVar INT =5
SET @NewVar = @NewVar * 2
SELECT @NewVar
```

Команда SET может одновременно работать только с одной переменной, поэтому, чтобы присвоить значения нескольким переменным, понадобится соответствующее количество таких команд.

В примере, представленном ниже, используются две отдельные команды SET, которые присваивают переменным название поставщика, код которого равен 21, а также его регион:

```
DECLARE @NameSup VARCHAR(20), @RegionSup VARCHAR(15)
SET @NameSup = (SELECT Название
```

```
        FROM Поставщик
        WHERE Код_постав = 21)
SET @RegionSup = (SELECT Регион
        FROM Поставщик
        WHERE Код_постав = 21)
SELECT @NameSup AS [Название поставщика], @RegionSup AS Регион
```

Результат операций присваивания значений двум переменным с помощью двух команд SET приведен на рис. 4.3.

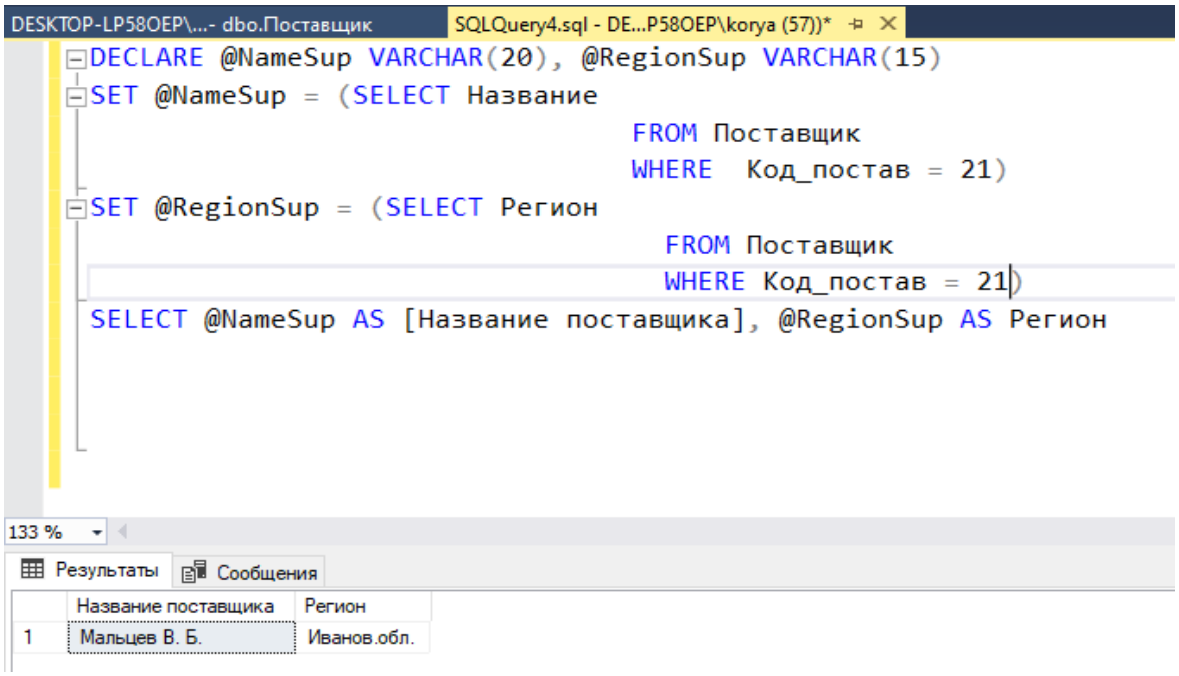



Рис. 4.3. Присваивание значений двум переменным с помощью двух команд SET

Команда SELECT позволяет присваивать нескольким переменным значения, полученные из одной строки. Например, присваивание, которое рассмотрено выше, можно выполнить с помощью одной команды SELECT:

```
DECLARE @NameSup VARCHAR(20), @RegionSup VARCHAR(15)
SELECT @NameSup = Название, @RegionSup = Регион
        FROM Поставщик
        WHERE Код_постав = 21
SELECT @NameSup AS [Название поставщика], @RegionSup AS Регион
```

На рис. 4.4 приведен результат присваивания значений двум переменным с помощью одной команды SELECT.

Команда SET более безопасна, чем SELECT, поскольку для извлечения данных из таблицы она использует вложенные скалярные подзапросы. Напоминаем, что скалярный подзапрос, возвращающий больше одного значения, завершается ошибкой.

В то же время если для присваивания используется команда SELECT, а внутренняя команда SELECT возвращает несколько значений, то в переменную помещается только последнее возвращенное значение.

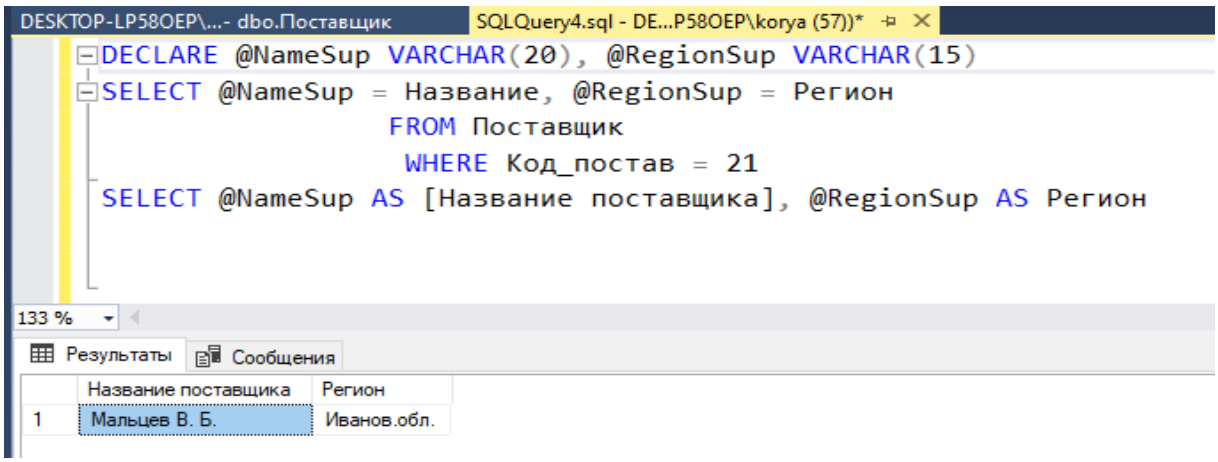



Рис. 4.4. Присваивание значений двум переменным
с помощью одной команды SELECT

Как уже отмечалось выше, локальные переменные можно объявлять в пакете.

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

Для того чтобы дать понять SQL-серверу, что передается пакет команд, необходимо указывать ключевое слово GO после всех команд, которые требуется объединить в пакет.

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

Допустим, пример, который мы использовали выше, объединим в пакет, а потом попробуем получить значения переменных:

DECLARE @mynum INT, @mychar CHAR (2)
SELECT @mynum = COUNT(*) FROM Поставщик
SET @mychar = CONVERT(char(2),@mynum)
SELECT @mynum AS ‘Переменная mynum’, @mychar AS ‘Переменная mychar’
GO
SELECT @mynum AS ‘Переменная mynum’

В результате выполнения всего этого кода произойдет ошибка (рис. 4.5), связанная с тем, что переменная @mynum, которая присутствует в последней команде SELECT, не объявлена.

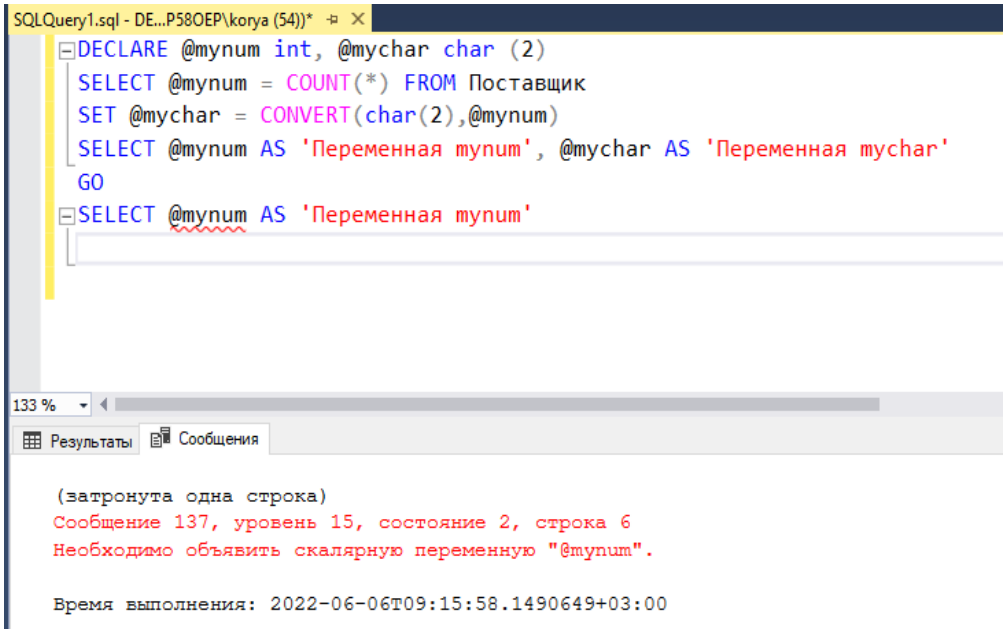



Рис. 4.5. Пример использования локальной переменной вне пакета, в котором она объявлена

***4.3.1.2. Условные команды IF и CASE.*** Эти команды предназначены для организации ветвления вычислительного процесса. Команда IF есть, наверное, во всех языках программирования.

Синтаксис этой команды:
IF <логическое_выражение>
<команда_SQL> | <блок _команд>
[ELSE
<команда_SQL> | <блок _команд>]

Предложение ELSE – необязательная часть команды IF.

Команда IF подразумевает проверку выполнения условия, и если проверка пройдена, то выполняется команда, идущая следом, если нет, то не выполняется ничего, но можно указать ключевое слово ELSE, и тогда будут выполняться команды, указанные после этого слова.

Например, выведем соответствующее сообщение о наличии в базе данных товаров, у которых номенклатурный номер находится в диапазоне от 100 до 150:

```
DECLARE @var1 VarChar (50)
IF EXISTS (SELECT * FROM Склад WHERE Ном_ном BETWEEN 100 AND 150)
   SET @var1 = ‘Сведения о таких товарах в БД есть’
ELSE
   SET @var1 = ‘Сведений о таких товарах в БД нет’
SELECT @var1 AS ‘Значение переменной var1’
```

Результат выполнения команды IF приведен на рис. 4.6.

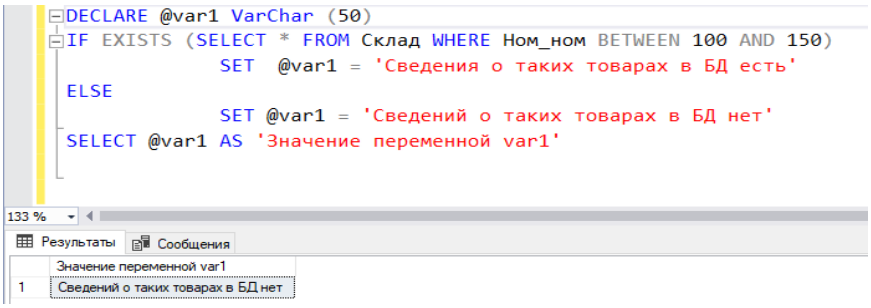



Рис. 4.6. Результат выполнения команды IF

В Transact-SQL-команда IF может иметь только одну ассоциированную с ней команду. Как быть, если требуется выполнить несколько команд?

Ключевые слова BEGIN и END применяются для обозначения набора команд, который должен быть выполнен как один блок:

```
BEGIN
        <Команда 1>
        <Команда 2>
         . . . . . .
        <Команда N>
END
```

Модифицируем предыдущий пример (про IF EXISTS) так, чтобы при наличии товаров с номенклатурными номерами от 10 040 до 10 110 в таблице СКЛАД, помимо присвоения значения переменной @Var, увеличивалась у этих товаров цена на 5 %, а также выводилось количество строк, которые мы обновили, используя глобальную переменную @@ROWCOUNT:

```
DECLARE @var1 VarChar (50), @Var2 INT
SET @Var2 = 0
IF EXISTS (SELECT * FROM Склад WHERE Ном_ном BETWEEN 10040 AND 10110)
        BEGIN
            SET @var1 = ‘Сведения о таких товарах в БД есть’
            UPDATE Склад SET Цена = 1.05* Цена
                WHERE Ном_ном BETWEEN 10040 AND 10110
            SET @Var2 = @@ROWCOUNT
    END
ELSE
     SET @var1 = ‘Сведений о таких товарах в БД нет’
SELECT @var1 AS ‘Наличие записей’,
          @Var2 AS ‘Затронуто строк’
```

На рис. 4.7 представлен результат выполнения этого программного кода.

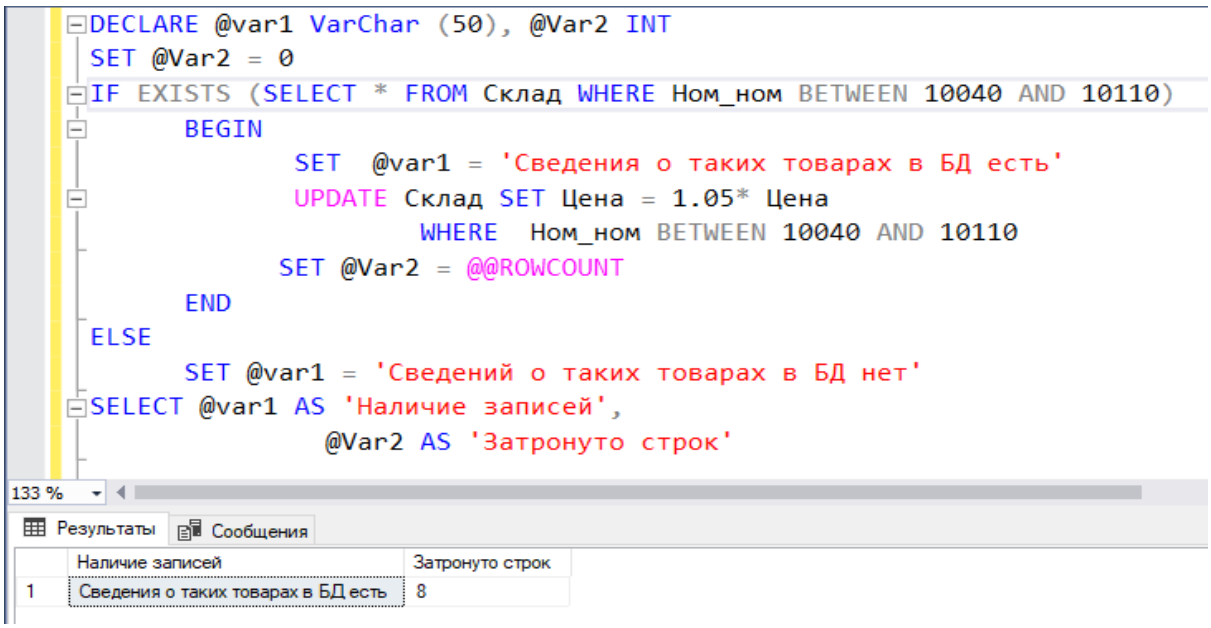



Рис. 4.7. Результат выполнения модифицированного примера

Другая команда, предназначенная для организации ветвления вычислительного процесса, – команда CASE. Данная команда применяется совместно с командой SELECT и предназначена для замены многократного использования команды IF.

Например:

```
DECLARE @Var1 INT, @Var2 VARCHAR(20)
SET @Var1 = 1
SELECT @Var2 = CASE @Var1
   WHEN 1 THEN 'Один'
   WHEN 2 THEN 'Два'
   ELSE 'Неизвестное'
   END
 SELECT @Var2 AS ‘Число’
```

Результат выполнения команды CASE приведен на рис. 4.8.

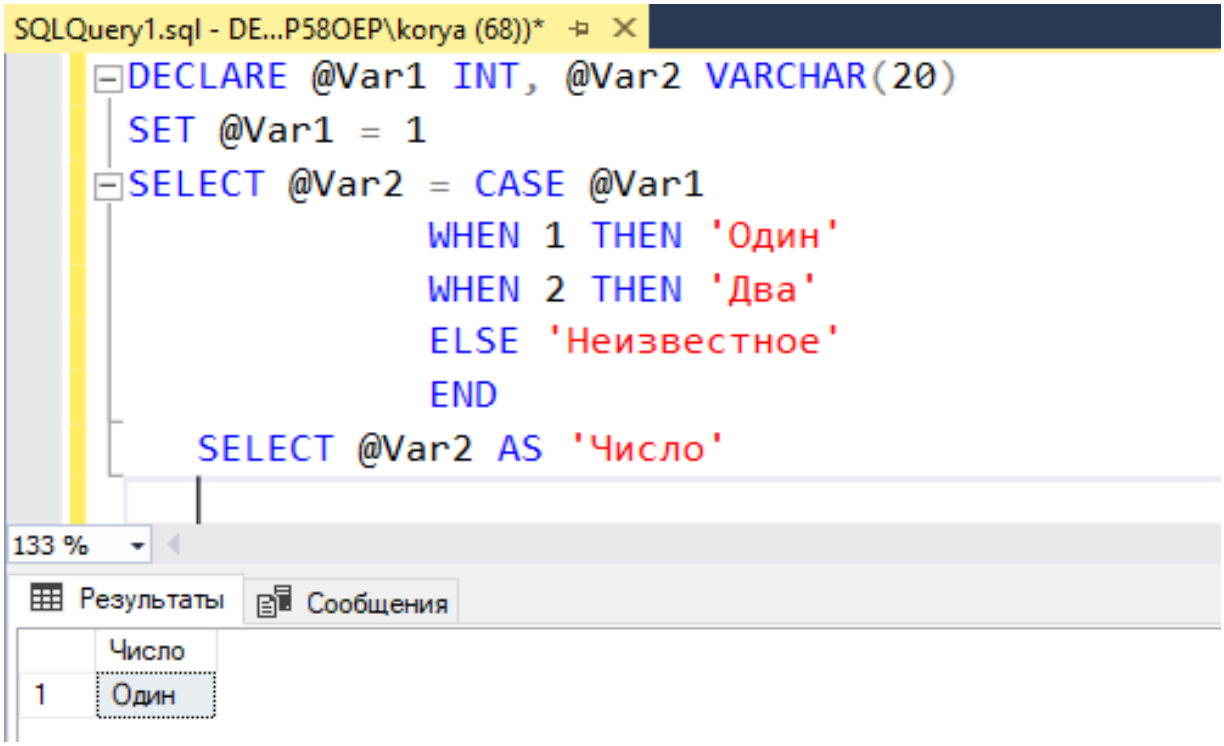



Рис. 4.8. Результат выполнения команды CASE

Как видно на рис. 4.8, в результате выполнения этого программного кода переменная @Var2 получит значение «Один».

***4.3.1.3. Команда цикла WHILE.*** Как известно, циклы предназначены для многократного повторения одних и тех же действий. В языке Transact-SQL есть одна команда цикла – WHILE (цикл с предусловием).

Синтаксис команды:

```
WHILE <логическое выражение>
    <Команда SQL>
```

Команда, следующая за предложением WHILE, выполняется циклически, пока логическое выражение имеет значение ИСТИНА.

Тело цикла WHILE может состоять только из одной команды SQL. Если в тело цикла необходимо включить несколько команд, используется конструкция BEGIN – END.

Например, предположим, что требуется увеличить плановые объемы продаж для каждого офиса торговой компании до тех пор, пока их сумма по всей компании не станет меньше 300 000 руб. План каждого офиса может быть увеличен на сумму, кратную 1000 руб.

```
WHILE (SELECT SUM(План) FROM Офис) < 300 000
UPDATE Офис
  SET План = План + 1000
  SELECT * FROM Офис
```

Результат выполнения команды WHILE представлен на рис. 4.9.

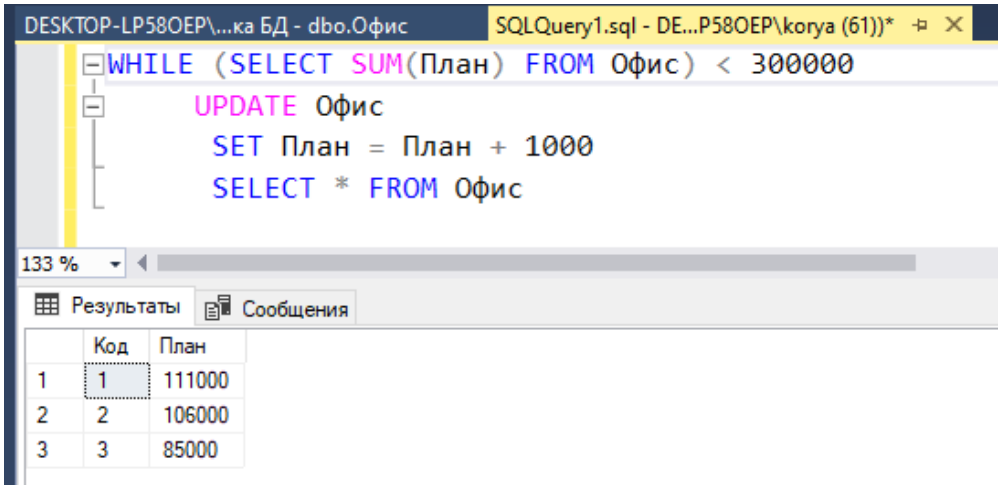



Рис. 4.9. Результат выполнения команды WHILE

Выполнение цикла можно контролировать с помощью ключевых слов BREAK и CONTINUE.

Рассмотрим следующий пример:

```
DECLARE @Cnt INT = 1, @result INT = 0, @CountRow INT
DECLARE @Cnt INT = 1, @Result INT = 0, @CountRow INT
SELECT @CountRow = COUNT(*) FROM TestTable2
WHILE @Cnt <= @CountRow
    BEGIN
        SET @Cnt = @Cnt + 1
        SET @Result = @Result +1
        IF @Cnt = 10
        BREAK
    ELSE
        CONTINUE
    END
SELECT @Cnt AS ‘Cnt’, @Result AS ‘Result’
```

В данном примере сначала объявляются переменные. Обратите внимание, что переменные Cnt и Result сразу инициализируются. Таким способом можно задавать значения переменных. Затем определяется, сколько строк в таблице TestTable2, и после этого проверяется условие: количество строк в таблице больше или равно значению счетчика. Если условие выполняется, то осуществляется вход в цикл. В цикле увеличиваются значения счетчика @Cnt и переменной @Result. А далее проверяется условие @Cnt = 10. Алгоритм работы цикла WHILE, а следовательно, и результат работы всей программы зависят от количества записей в файле TestTable2. Предположим, что файл TestTable2 содержит меньше 10 записей, например пять. В этом случае цикл WHILE за счет команды CONTINUE продолжит работу до тех пор, пока значение счетчика не станет больше количества строк в таблице (рис. 4.10).

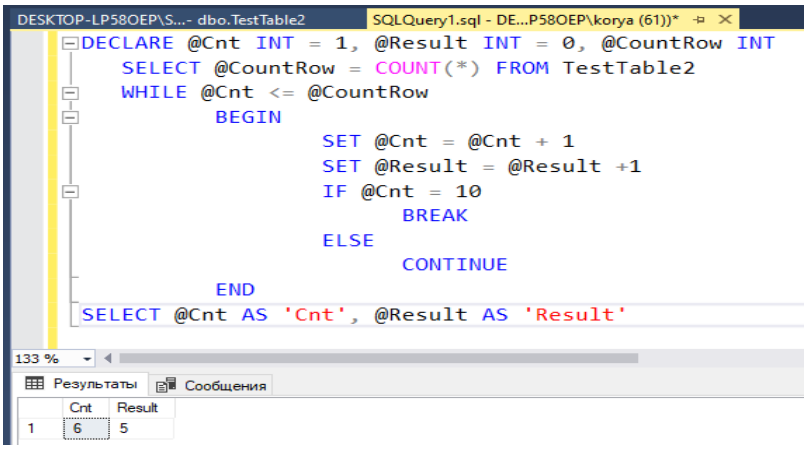



Рис. 4.10. Пример использования команды CONTINUE

Теперь рассмотрим случай, когда файл TestTable2 содержит больше 10 записей. Для этого добавим в файл семь записей, доведя количество записей в нем до 12. Тогда при достижении счетчиком значения 10 цикл WHILE принудительно завершается с помощью команды BREAK (рис. 4.11).

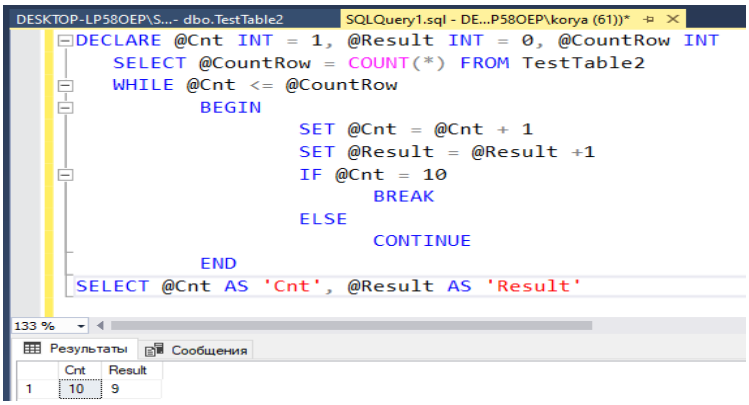



Рис. 4.11. Пример использования команды BREAK

***4.3.1.4. Команда RETURN.*** Команда RETURN может использоваться в любой точке для выхода из процедуры, пакета или блока команд. Все, что идет после этой команды, не выполняется.

```
DECLARE @Cnt INT = -1, @result varchar(15)
IF @Cnt < 0
      RETURN
SET @result = 'Cnt больше 0'
SELECT @result AS ‘Результат’
```

В этом примере если значение Cnt меньше нуля, то команды, следующие за командой RETURN, не выполнятся и мы не увидим столбец с именем «Результат» (рис. 4.12).

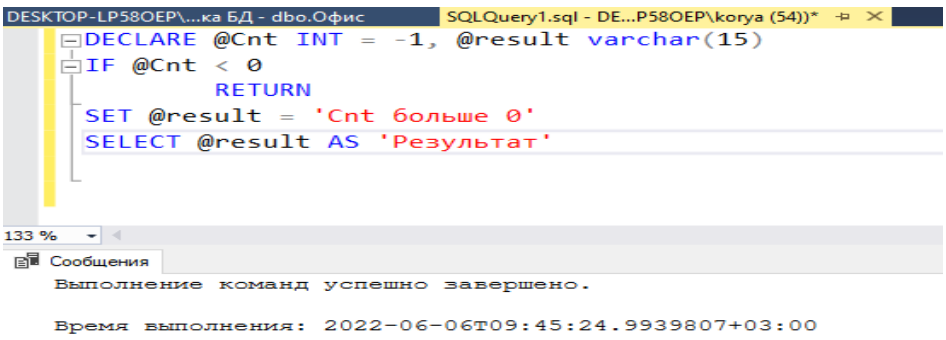



Рис. 4.12. Пример использования команды RETURN

***4.3.1.5. Конструкция TRY…CATCH.*** При выполнении кода, написанного на языке Transact-SQL, может возникнуть ошибка, которую необходимо обработать. В языке Transact-SQL, как и во многих других языках программирования, есть возможность отслеживать и перехватывать ошибки с помощью конструкции, состоящей из блоков TRY и CATCH.

Группа команд на языке Transact-SQL может быть заключена в блок TRY. Если ошибка возникает в блоке TRY, управление передается следующей группе команд, заключенной в блок CATCH.

```
BEGIN TRY
< команда SQL> | <группа команд SQL>
END TRY
BEGIN CATCH
[ < команда SQL> | <группа команд SQL> ]
END CATCH
```

В этом примере:

- команда SQL – любая из команд языка Transact-SQL;
- группа команд SQL – любая группа команд языка Transact-SQL в пакете или группа-команд, заключенная в блок BEGIN…END.

Таким образом, все, что необходимо проверять на ошибки, т. е. код, в котором могут возникнуть ошибки, помещается в блок TRY. Начало данного блока обозначается инструкцией BEGIN TRY, а окончание блока – END TRY.

Все, что требуется выполнять в случае появления ошибки, т. е. те команды, которые необходимо выполнить, если в блоке TRY возникла ошибка, помещается в блок CATCH. Его начало обозначается BEGIN CATCH, а окончание – END CATCH.

За блоком TRY сразу же должен следовать блок CATCH. Размещение каких-либо команд между инструкциями END TRY и BEGIN CATCH вызовет синтаксическую ошибку.

Если ошибки в блоке TRY не возникают, то после выполнения последней команды в блоке TRY управление передается команде, расположенной сразу после инструкции END CATCH. Если же в коде, заключенном в блоке TRY, происходит ошибка, управление передается первой команде в соответствующем блоке CATCH. Если инструкция END CATCH – последняя инструкция хранимой процедуры

или триггера, управление передается обратно команде, вызвавшей эту хранимую процедуру или триггер.

Для того чтобы получить информацию об ошибках, которые повлекли за собой выполнение блока CATCH, можно использовать следующие функции:

- ERROR_NUMBER() – возвращает номер ошибки;
- ERROR_MESSAGE() – возвращает описание ошибки;
- ERROR_STATE() – возвращает код состояния ошибки;
- ERROR_SEVERITY() – возвращает степень серьезности ошибки;
- ERROR_PROCEDURE() – возвращает имя хранимой процедуры или триггера, в которых произошла ошибка;
- ERROR_LINE() – возвращает номер строки инструкции, которая вызвала ошибку.

Если эти функции вызвать вне блока CATCH, они вернут значение NULL.

Когда код в блоке CATCH завершен, управление передается команде, стоящей сразу после инструкции END CATCH.

Для демонстрации того, как работает конструкция TRY…CATCH, напишем простую SQL-команду, в которой намеренно допущена ошибка, например попытаемся выполнить операцию деления на ноль.

```
BEGIN TRY
   DECLARE @TestVar1 INT = 10,
                  @TestVar2 INT = 0,
                  @Rez INT
   SET @Rez = @TestVar1 / @TestVar2
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() AS [Номер ошибки],
                 ERROR_MESSAGE() AS [Описание ошибки]
END CATCH
```

Результат работы конструкции TRY…CATCH приведен на рис. 4.13.

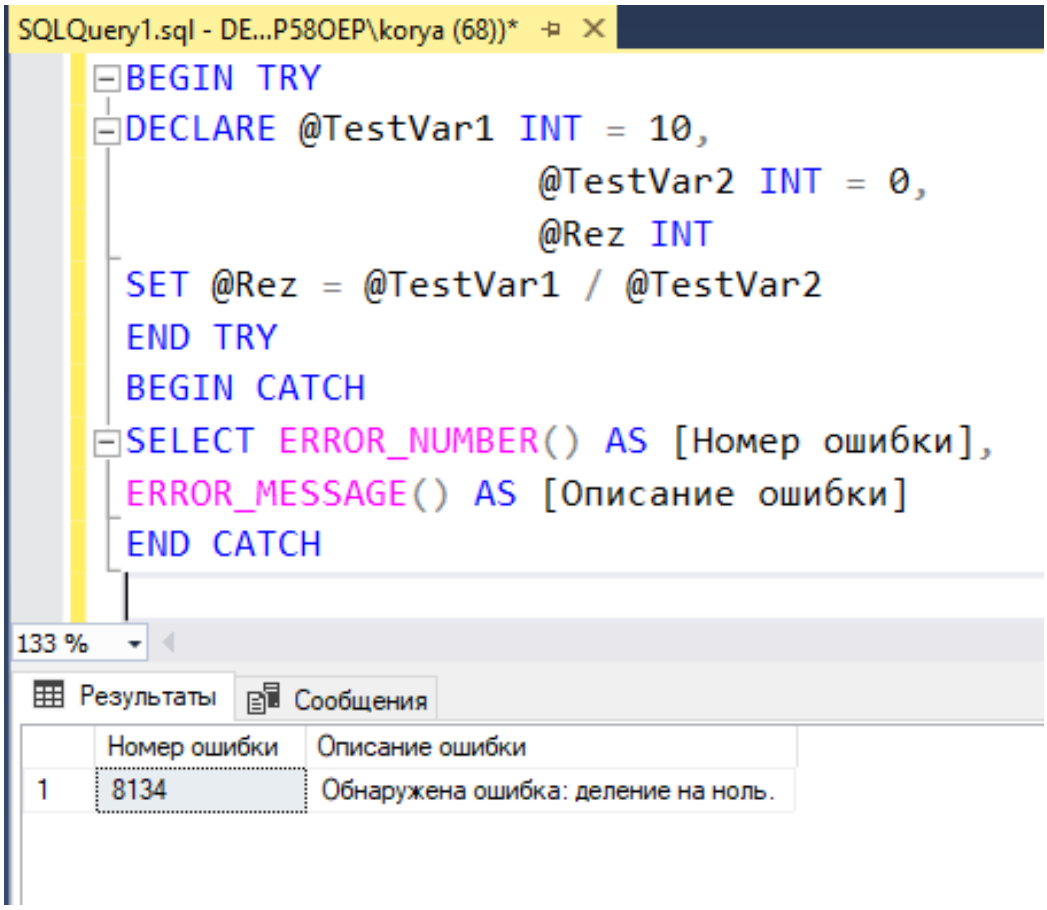



Рис. 4.13. Результат работы конструкции TRY…CATCH

В этом примере возникла ситуация деления на ноль. А так как программный код, в котором произошла ошибка, был помещен в блок TRY, эта ошибка была перехвачена и получены номер ошибки и описание ошибки.

### *4.3.2. Хранимые процедуры*

***4.3.2.1. Понятие хранимой процедуры.*** Хранимые процедуры представляют собой набор команд, состоящий из одной или нескольких команд языка SQL и сохраняемый в базе данных в откомпилированном виде. Это свойство хранимых процедур предоставляет важную выгоду, заключающуюся в устранении повторных компиляций процедуры и получении соответствующего улучшения производительности.

Хранение процедур в том же месте, где они исполняются, обеспечивает уменьшение объема передаваемых по сети данных и повышает общую производительность системы.

Применение хранимых процедур упрощает сопровождение программных комплексов и внесение изменений в них. Обычно все ограничения целостности в виде правил и алгоритмов обработки данных реализуются на сервере баз данных и доступны конечному приложению в виде набора хранимых процедур, которые и представляют интерфейс обработки данных. Для обеспечения целостности данных, а также в целях безопасности приложение обычно не получает прямого доступа к данным – вся работа с ними ведется путем вызова тех или иных хранимых процедур.

Подобный подход делает весьма простой модификацию алгоритмов обработки данных, тотчас же становящихся доступными для всех пользователей сети, и обеспечивает возможность расширения системы без внесения изменений в само приложение: достаточно изменить хранимую процедуру на сервере баз данных. Разработчику не нужно перекомпилировать приложение, создавать его копии, а также инструктировать пользователей о необходимости работы с новой версией. Пользователи вообще могут не подозревать о том, что в систему внесены изменения.

Хранимые процедуры существуют независимо от таблиц или каких-либо других объектов баз данных. Они вызываются клиентской программой, другой хранимой процедурой или триггером. Разработчик может управлять правами доступа к хранимой процедуре, разрешая или запрещая ее выполнение. Изменять код хранимой процедуры разрешается только ее владельцу или члену фиксированной роли базы данных. При необходимости можно передать права владения ею от одного пользователя к другому.

***4.3.2.2. Хранимые процедуры в среде MS SQL Server. Типы хранимых процедур.*** В СУБД MS SQL Server имеется несколько типов хранимых процедур: системные, временные и пользовательские.

*Системные хранимые процедуры* предоставляет MS SQL Server, в их названии имеется префикс sp_. Они применяются для доступа к информации в системном каталоге и ее модификации.

*Временные хранимые процедуры* существуют лишь некоторое время, после чего автоматически уничтожаются сервером. Они делятся на локальные и глобальные. Локальные временные хранимые про-

цедуры могут быть вызваны только из того соединения, в котором созданы. При создании такой процедуры ей необходимо дать имя, начинающееся с одного символа #. Как и все временные объекты, хранимые процедуры этого типа автоматически удаляются при отключении пользователя, перезапуске или остановке сервера. Глобальные временные хранимые процедуры доступны для любых соединений сервера, на котором имеется такая же процедура. Для определения глобальной временной хранимой процедуры достаточно дать ей имя, начинающееся с символов ##. Удаляются эти процедуры при перезапуске или остановке сервера, а также при закрытии соединения, в контексте которого они были созданы.

*Пользовательские хранимые процедуры*, реализующие те или иные действия, создаются пользователем.

Ниже речь пойдет о пользовательских хранимых процедурах.

***Создание, изменение и удаление хранимых процедур.*** Хранимые процедуры создаются, изменяются и удаляются таким же образом, как и все другие объекты баз данных, т. е. при помощи языка DDL.

*Создание новой и изменение* имеющейся хранимой процедуры осуществляются с помощью следующей команды:

```
CREATE | ALTER PROC[EDURE] <имя_процедуры>
[<@ имя_параметра> <тип_данных>
[= <значение_по_умолчанию>] [OUTPUT] ]
 [WITH RECOMPILE | ENCRYPTION | EXECUTE AS 'user_name']
[FOR REPLICATION]
AS
<Тело процедуры>
```

Рассмотрим параметры данной команды.

Используя префиксы sp_, #, ## в имени, создаваемую процедуру можно определить в качестве системной или временной.

Как видно из синтаксиса команды, в ней не указывается имя владельца, которому будет принадлежать создаваемая процедура, а также имя базы данных, где она должна быть размещена. Таким образом, чтобы разместить создаваемую хранимую процедуру в конкрет-

ной базе данных, необходимо выполнить команду CREATE PROCEDURE в контексте этой базы данных. При обращении из тела хранимой процедуры к объектам той же базы данных можно использовать укороченные имена, т. е. без указания имени базы данных. Когда же требуется обратиться к объектам, расположенным в других базах данных, указание имени базы данных обязательно.

При создании хранимой процедуры можно определить необязательный список входных и выходных параметров. Таким образом, процедура будет принимать соответствующие аргументы при каждом ее вызове и может возвращать значения, содержащие определенную пользователем информацию, или может не возвращать ничего, только выполнять заложенный в ней набор команд.

Имена параметров, как и имена локальных переменных, должны начинаться с символа @. В одной хранимой процедуре можно задать множество параметров, разделенных запятыми. В теле процедуры не должны применяться локальные переменные, чьи имена совпадают с именами параметров этой процедуры.

Для определения типа данных, который будет иметь соответствующий параметр хранимой процедуры, годятся любые типы данных СУБД MS SQL Server, включая определенные пользователем.

Необязательный параметр [= значение_по_умолчанию] определяет значение по умолчанию для соответствующего параметра процедуры. Таким образом, при вызове процедуры можно не указывать явно значение соответствующего параметра.

Наличие ключевого слова OUTPUT означает, что соответствующий параметр предназначен для возвращения данных из хранимой процедуры в вызывающую процедуру или систему. Однако это вовсе не означает, что параметр не подходит для передачи значений в хранимую процедуру. Указание ключевого слова OUTPUT предписывает серверу при выходе из хранимой процедуры присвоить текущее значение параметра локальной переменной, которая была указана при вызове процедуры в качестве значения параметра. Отметим, что при указании ключевого слова OUTPUT значение соответствующего параметра при вызове процедуры может быть задано только с помощью локальной переменной. Не разрешается использование любых выражений или констант, допустимое для обычных параметров.

Как уже упоминалось ранее, предварительно компилированная форма процедуры сохраняется в базе данных и используется при каждом ее вызове. Если же по каким-либо причинам хранимую процедуру требуется компилировать при каждом ее вызове, при объявлении процедуры используется опция WITH RECOMPILE. Применение опции WITH RECOMPILE сводит на нет одно из наиболее важных преимуществ хранимых процедур: улучшение производительности благодаря одной компиляции. Поэтому опцию WITH RECOMPILE следует использовать только при частых изменениях используемых хранимой процедурой объектов базы данных.

Ключевое слово WITH ENCRYPTION предписывает серверу выполнить шифрование кода хранимой процедуры, что может обеспечить защиту от использования авторских алгоритмов, реализующих работу хранимой процедуры.

Предложение EXECUTE AS 'user_name' определяет контекст безопасности, в котором должна исполняться хранимая процедура после ее вызова. Задавая этот контекст, с помощью СУБД можно управлять выбором учетных записей пользователей для проверки полномочий доступа к объектам, на которые ссылается данная хранимая процедура.

Параметр FOR REPLICATION востребован при репликации данных и включении создаваемой хранимой процедуры в качестве статьи в публикацию.

Ключевое слово AS размещается в начале тела хранимой процедуры, т. е. набора команд SQL, с помощью которых и будет реализовываться то или иное действие. В теле процедуры могут применяться практически все команды SQL, объявляться транзакции, устанавливаться блокировки и вызываться другие хранимые процедуры. Выход из хранимой процедуры можно осуществить посредством команды RETURN.

Хранимые процедуры могут обращаться к несуществующим таблицам. Это свойство позволяет выполнять отладку кода процедуры, не создавая сначала соответствующие таблицы и даже не подключаясь к конечному серверу.

В отличие от основных хранимых процедур, которые всегда сохраняются в текущей базе данных, возможно создание временных

хранимых процедур, которые всегда помещаются во временную системную базу данных tempdb. Одним из поводов для создания временных хранимых процедур может быть желание избежать повторяющегося исполнения определенной группы инструкций при соединении с базой данных. Можно создавать локальные или глобальные временные процедуры. Для этого имя локальной процедуры задается с одинарным символом # (#proc_name), а имя глобальной процедуры – с двойным (##proc_name).

Локальную временную хранимую процедуру может выполнить только создавший ее пользователь и только в течение соединения с базой данных, в которой она была создана. Глобальную временную процедуру могут выполнять все пользователи, но только до тех пор, пока не завершится последнее соединение, в котором она выполняется (обычно это соединение создателя процедуры).

*Удаление хранимой процедуры* осуществляется командой:

DROP PROCEDURE {имя_процедуры}

***Выполнение хранимой процедуры.*** Каждая процедура создается один раз, а выполняется многократно. Хранимая процедура реализуется посредством команды EXECUTE пользователем, который является владельцем процедуры или обладает правом EXECUTE для доступа к этой процедуре.

Для выполнения хранимой процедуры используется команда

[ EXEC [UTE]] <имя_процедуры>
[[<@имя_параметра> =]<значение> | <@имя_переменной>
[OUTPUT] | [DEFAULT]]

Если вызов хранимой процедуры не единственная команда в пакете, то присутствие команды EXECUTE обязательно. Более того, эта команда требуется для вызова процедуры из тела другой процедуры или триггера.

Все параметры команды EXECUTE имеют такое же логическое значение, как и одноименные параметры инструкции CREATE PROCEDURE.

Использование ключевого слова OUTPUT при вызове процедуры разрешается только для параметров, которые были объявлены при создании процедуры с ключевым словом OUTPUT.

Когда же при вызове процедуры для параметра указывается ключевое слово DEFAULT, то будет использовано значение по умолчанию. Естественно, указанное слово DEFAULT разрешается только для тех параметров, для которых определено значение по умолчанию.

Из синтаксиса команды EXECUTE видно, что имена параметров могут быть опущены при вызове процедуры. В этом случае пользователь должен указывать значения для неименованных параметров в том же порядке, в каком они перечислялись при создании процедуры в команде CREATE PROCEDURE. Присвоить параметру значение по умолчанию, просто пропустив его при перечислении, нельзя. Если же требуется опустить параметры, для которых определено значение по умолчанию, достаточно явного указания имен параметров при вызове хранимой процедуры. Более того, таким способом можно перечислять параметры и их значения в произвольном порядке.

Отметим, что при вызове процедуры указываются либо имена параметров со значениями, либо только значения без имени параметра. Их комбинирование не допускается.

Рассмотрим примеры использования команд CREATE PROCEDURE и EXECUTE. Для иллюстрации основных возможностей этих команд и всех других команд, которые будут рассматриваться в пособии, будем использовать учебную базу данных «Контроль_товара» промышленного предприятия, которая отражает процесс поступления материалов и товаров на его склад. Учебная база данных «Контроль_товара» описана в п. 4.2.1 и приложении.

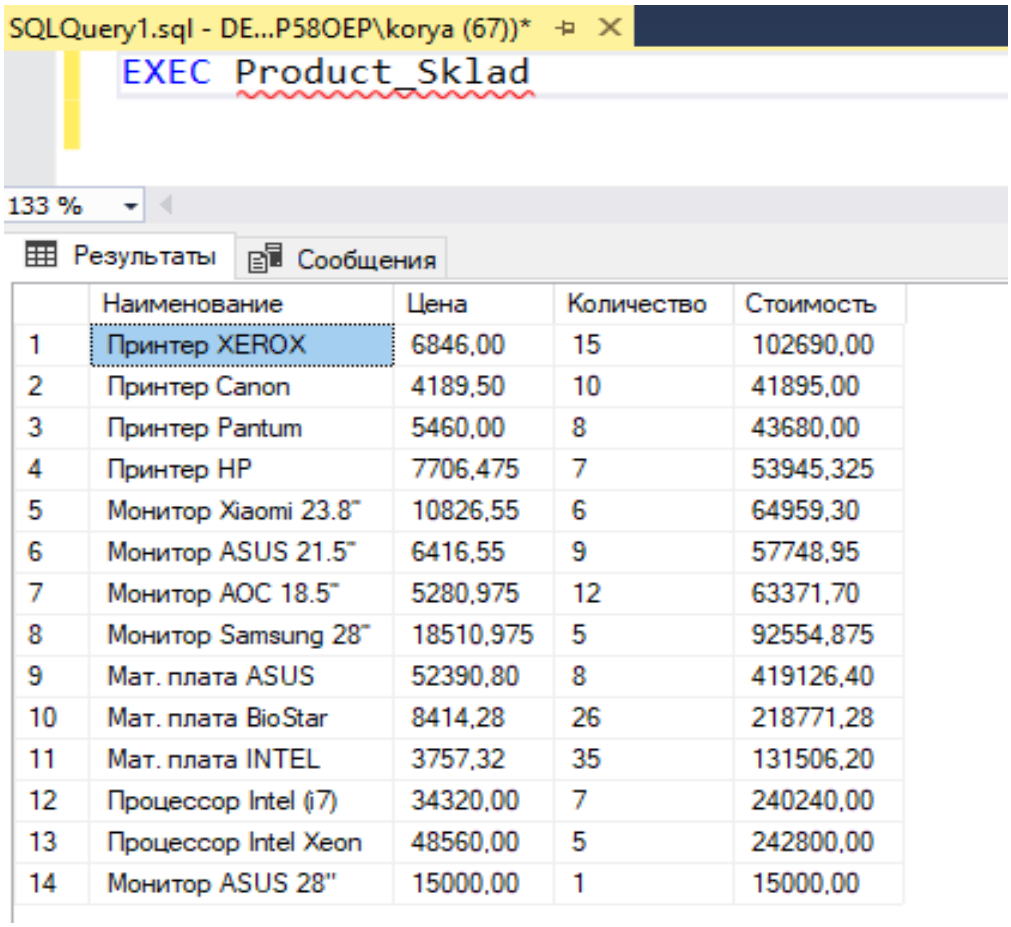
Ниже приведен пример создания простой процедуры без параметров, которая содержит команду SELECT для отображения названия, цены, количества и стоимости всех товаров, хранящихся на складе.

```
CREATE PROC Product_Sklad
AS
SELECT Наименование, Цена, Количество,
Цена * Количество AS Стоимость
FROM Склад
```

Если запустить данный набор, то будет создана хранимая процедура. Для обращения к процедуре можно использовать команду

```
EXEC Product_Sklad
```

Вызов процедуры Product_Sklad и результат ее выполнения представлены на рис. 4.14.




| | Наименование | Цена | Количество | Стоимость |
|---|---|---|---|---|
| 1 | Принтер XEROX | 6846,00 | 15 | 102690,00 |
| 2 | Принтер Canon | 4189,50 | 10 | 41895,00 |
| 3 | Принтер Pantum | 5460,00 | 8 | 43680,00 |
| 4 | Принтер HP | 7706,475 | 7 | 53945,325 |
| 5 | Монитор Xiaomi 23.8" | 10826,55 | 6 | 64959,30 |
| 6 | Монитор ASUS 21.5" | 6416,55 | 9 | 57748,95 |
| 7 | Монитор AOC 18.5" | 5280,975 | 12 | 63371,70 |
| 8 | Монитор Samsung 28" | 18510,975 | 5 | 92554,875 |
| 9 | Мат. плата ASUS | 52390,80 | 8 | 419126,40 |
| 10 | Мат. плата BioStar | 8414,28 | 26 | 218771,28 |
| 11 | Мат. плата INTEL | 3757,32 | 35 | 131506,20 |
| 12 | Процессор Intel (i7) | 34320,00 | 7 | 240240,00 |
| 13 | Процессор Intel Xeon | 48560,00 | 5 | 242800,00 |
| 14 | Монитор ASUS 28'' | 15000,00 | 1 | 15000,00 |

Рис. 4.14. Вызов процедуры Product_Sklad и результат ее выполнения

Ниже приводится хранимая процедура с одним параметром, которая выводит имя поставщика и его регион по заданному имени региона.

```
CREATE proc From_supplier
@Reg VarChar (15)
AS
SELECT Название, Регион
FROM Поставщик
WHERE Регион = @Reg
```

Для запуска этой хранимой процедуры необходимо указать входной параметр. Так, для того чтобы получить список поставщиков из Московской области, надо выполнить следующую команду:

```
EXEC From_supplier ‘Московск.обл.’
```

Вы увидите строки, возвращенные в результате вызова этой хранимой процедуры (рис. 4.15).

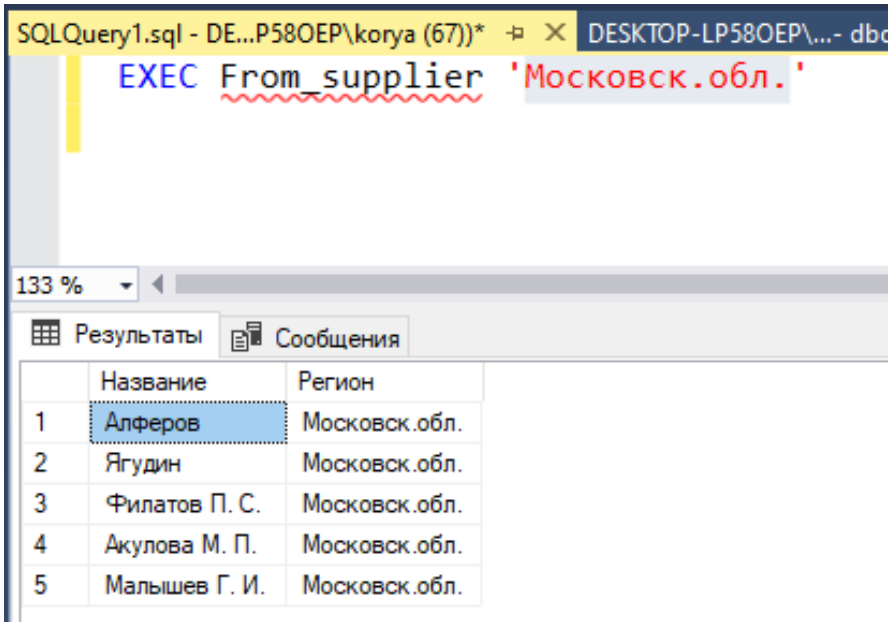



Рис. 4.15. Результат выполнения процедуры From_supplier

Если параметр не указан, СУБД MS SQL Server выведет сообщение об ошибке.

Для параметра можно задавать значение по умолчанию, которое будет применяться, когда этот параметр не указан в обращении к процедуре. Например, чтобы использовать для хранимой процедуры значение по умолчанию ‘Владимир.обл’, изменим текст создаваемой процедуры следующим образом:

```
CREATE proc From_supplier1
@Reg VarChar (15) = ‘Владимир.обл’
AS
SELECT Name, Region
FROM Supplier
WHERE Region = @Reg
```

Если теперь обратиться к процедуре командой без входного параметра

```
EXEC From_supplier1,
```

то для параметра @Reg по умолчанию будет использоваться значение ‘Владимир.обл’ и в результате выполнения процедуры будет получен список поставщиков из Владимирской области (рис. 4.16).

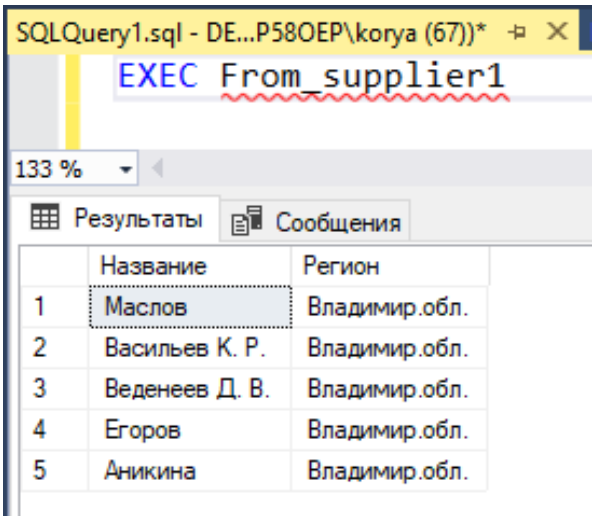



Рис. 4.16. Список поставщиков из Владимирской области, полученный в результате вызова хранимой процедуры From_supplier1

Но если при вызове процедуры указать входной параметр, то его значение заместит значение, определенное по умолчанию, например:

EXEC From_supplier1 ‘Московск.обл.’

или

EXEC From_supplier1 @Reg = ‘Московск.обл.’

При таком обращении к процедуре будет получен список поставщиков из региона Московской области (рис. 4.17).

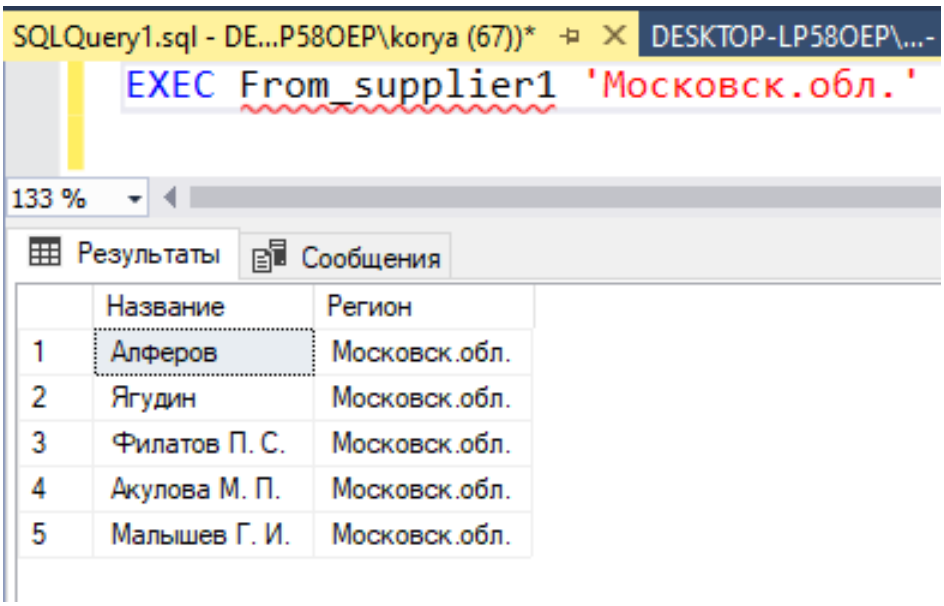



Рис. 4.17. Список поставщиков из Московской области, полученный в результате вызова хранимой процедуры From_supplier1

Рассмотрим более сложный пример, когда процедура содержит два параметра. Эта процедура выводит имя поставщика, его регион и тип поставляемых им товаров по заданным типу товара и региону поставщика.

```
CREATE PROC Tp_Reg
@Tp VarChar (20),
@region VarChar (15)
AS
SELECT distinct a.Название, a.Регион, c.Тип
FROM Поставщик a, Документ b, Приход c
WHERE c.Тип = @Tp
AND a.Регион = @region
AND a. Код_постав = b. Код_постав
AND b. Ном_док = c. Ном_док
```

Вызов этой процедуры осуществляется следующим образом:

EXEC Tp_Reg ‘Принтер’, ‘Владимир.обл.’

или

EXEC Tp_Reg @Tp = ‘Принтер’, @region = ‘Владимир.обл.’

Вызов процедуры Tp_Reg и результат ее выполнения представлены на рис. 4.18.

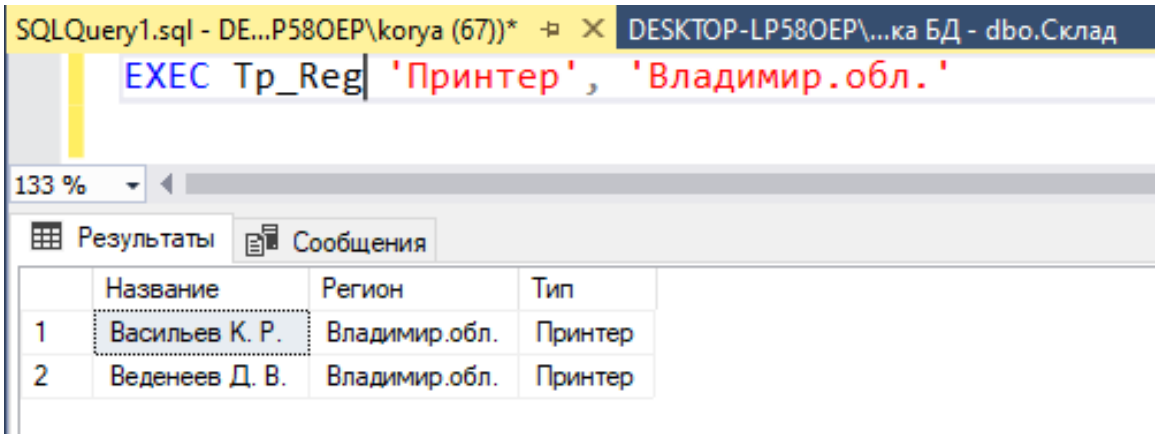



Рис. 4.18. Вызов процедуры Tp_Reg и результат ее выполнения

Рассмотрим пример процедуры, которая содержит входные и выходные параметры. Эта процедура служит для определения общей стоимости товаров, поступивших за конкретный месяц.

```
CREATE PROC Summ_Coming
  @m INT,
  @s REAL OUTPUT
AS
   SELECT @s=Sum(Цена * Количество)
     FROM Приход INNER JOIN Документ
     ON Приход.Ном_док = Документ. Ном_док
     GROUP BY Month(Дата)
   HAVING Month(Дата)=@m
```

Для обращения к процедуре можно использовать команды

```
DECLARE @st REAL
EXEC Summ_Coming 5,@st OUTPUT
SELECT @st AS ‘Общая стоимость товаров, поступивших за месяц’
```

Результат выполнения процедуры Summ_Coming приведен на рис. 4.19.

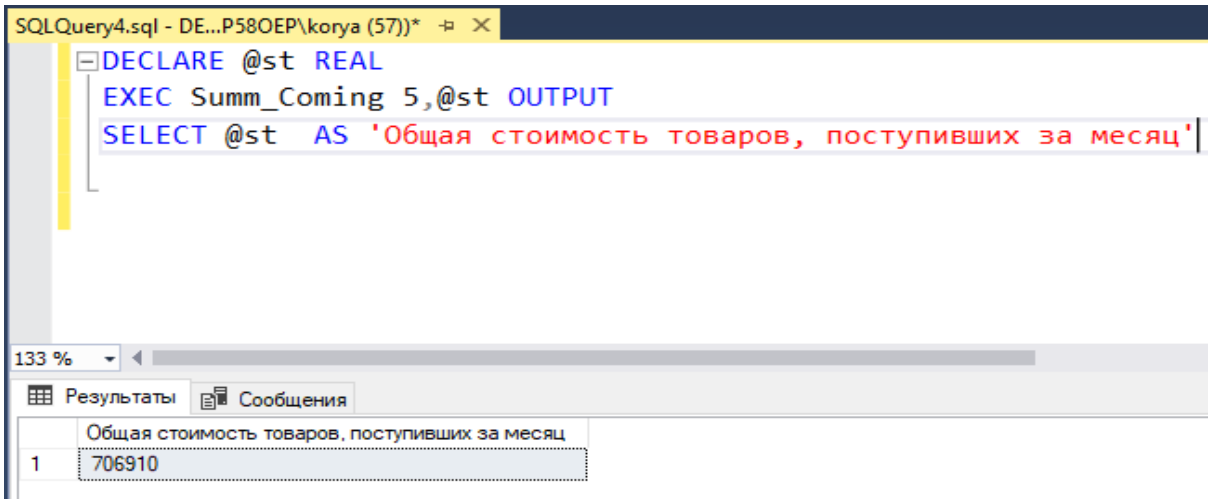



Рис. 4.19. Результат выполнения процедуры Summ_Coming

Этот блок команд позволяет определить стоимость товаров, проданных в мае (входной параметр – месяц – указан равным пяти).

Рассмотрим пример использования хранимой процедуры для выполнения операций в спроектированной базе данных учета поступающих товаров на склад предприятия.

При поступлении товара на склад необходимо выполнить следующие операции с базой данных:

1) добавить новую запись в таблицу ДОКУМЕНТ;

2) добавить новую запись в таблицу ПРИХОД;

3) изменить значение поля Количество в таблице СКЛАД.

Все эти операции могут быть выполнены одной заранее созданной хранимой процедурой:

```
CREATE PROC New_Product
@N_D INT,            /*номер документа*/
@Dt  DateTime,       /*дата поступления*/
@Kod_C INT,          /*код поставщика*/
@N_N INT,            /*номенклатурный номер*/
@P_C  MONEY,         /*цена поступившего товара*/
@Kolvo INT,          /*количество поступившего товара*/
@Typ VarChar(20)     /*Тип товара*/
AS
/*Добавляем новую запись в таблицу Документ*/
INSERT INTO Документ(Ном_док, Дата, Код_постав)
 VALUES(@N_D, @Dt, @Kod_C)
/*Добавляем новую запись в таблицу Приход */
INSERT INTO Приход(Ном_док, Ном_ном, Цена, Количество, Тип)
VALUES(@N_D, @N_N, @P_C, @Kolvo, @Typ)
/*Обновляем запись в таблице Склад*/
UPDATE Склад
 SET Количество = Количество + @Kolvo
 WHERE Ном_ном= @N_N
```

При вызове этой процедуры, задавая параметры, можно явно указать их имена, что позволяет перечислить параметры в любом порядке.

```
EXEC New_Product @N_D = 65,
                 @Dt= '2022-08-01',
                 @Kod_C = 21,
                 @N_N = 2,
                 @P_C = 2500,
                 @Kolvo = 7,
                @Typ =’Принтер’
```

## Вопросы для самопроверки

1. Что такое хранимая процедура?
2. Где выполняются хранимые процедуры?
3. Как активизируются хранимые процедуры?
4. В чем преимущества использования хранимых процедур?
5. Какие типы хранимых процедур имеются в СУБД MS SQL Server?
6. С какой целью используются параметры хранимых процедур?

## Практические задания

1. Дана база данных КЛИЕНТЫ:

КЛИЕНТ (Код_клиента, Наименование, Годовой_доход, Тип_заказчика [Производитель, Оптовый_продавец, Торговая_ компания])

ОТГРУЗКА (Номер_отгрузки, Код_клиента, Вес, Номер_грузовика, Город, Дата)

ВОДИТЕЛЬ (Номер_отгрузки, Имя_водителя)

ГОРОД (Название, Число_жителей)

а) Напишите хранимую процедуру с параметрами, в которой определяется средний вес грузов, перевозимых водителем X в город Y?

б) Напишите хранимую процедуру с параметром, определяющую средний вес груза заданного клиента.

в) Напишите хранимую процедуру с параметрами, которая определяет общий суммарный вес доставленного груза в указанный город каждым водителем.

2. Даны три таблицы:

ПРОДАВЕЦ (Код_продавца, Имя, Город, Комиссионные)

ЗАКАЗЧИК (Код_покупателя, ФИО, Рейтинг, Город, Код_продавца)

ПОКУПКА (Номер, Сумма, Дата, Код__продавца, Код_покупателя)

а) Напишите хранимую процедуру с параметрами, которая выводит рейтинг и имя каждого заказчика в городе X, делавшего покупки в городе Y.

б) Напишите хранимую процедуру с параметрами, которая определяет названия клиентов, отправлявших грузы в заданный город.

### *4.3.3. Триггеры*

***4.3.3.1. Основные сведения о триггерах MS SQL Server.*** Триггеры могут быть определены как объекты базы данных, которые выполняют одну или несколько команд Transact-SQL в ответ на событие, происходящее в таблице, базе данных или на сервере.

Событием, которое приводит к срабатыванию триггера, может быть или команда DML (Data Manipulation Language – язык манипулирования данными), или команда DDL (Data Definition Language – язык определения данных). Таким образом, существует два типа триггеров: триггеры DML и триггеры DDL.

Событиями для триггера DML являются команды UPDATE, INSERT или DELETE, которые изменяют данные в таблице. Поэтому каждый триггер DML привязан к конкретной таблице.

Триггер DDL срабатывает в ответ на события (ALTER, DROP, CREATE), которые пытаются изменить схему базы данных. Поэтому триггер DDL не создается для конкретной таблицы, но он применим ко всем таблицам в базе данных.

В процессе эксплуатации базы данных изменение ее схемы требуется крайне редко, в отличие от изменения данных, хранящихся в ней. Поэтому в практике баз данных наиболее часто используются триггеры DML. Далее в пособии речь будет идти только о триггерах DML.

*Триггер* – это специальный тип хранимой процедуры, которая запускается автоматически СУБД при модифицировании какой-либо таблицы одной из трех команд: UPDATE, INSERT или DELETE.

Исключительно важно в этом определении слово *автоматически*. Ни пользователь, ни приложение не могут активизировать триггер, он выполняется автоматически, когда пользователь или приложение проводят с базой данных операции обновления, добавления или удаления данных. К тому же триггер не имеет аргументов.

Операции обновления, вставки и удаления называются *триггерными событиями*.

Когда происходит запуск триггера, говорят, что он *активизируется*.

Каждый триггер DML, как уже отмечалось выше, привязывается к конкретной таблице, но он может осуществлять доступ и к другим таблицам и объектам даже других баз данных. Триггеры нельзя со-

здать по временным или системным таблицам, а только по определенным пользователем таблицам или представлениям. Таблица, по которой определяется триггер, называется *таблицей триггера.*

При работе с триггерами следует знать следующие правила.

- Триггер запускается только после завершения команды, которая вызвала его активизацию.
- Если какая-либо команда пытается выполнить операцию, которая нарушает какое-либо ограничение по таблице или является причиной какой-то другой ошибки, то связанный с ней триггер не будет активизирован.
- Триггер рассматривается как часть одной транзакции вместе с командой, которая вызывает его. Поэтому из триггера можно вызвать команду отката транзакции, и эта команда выполнит откат как триггера, так и соответствующего события модификации данных.

Триггеры используются для поддержания целостности данных в базе данных. С помощью декларативных ограничений целостности не всегда можно добиться нужного уровня функциональности. Часто требуется реализовать сложные алгоритмы проверки данных, гарантирующие достоверность и реальность данных. Например, триггеры следует использовать для проведения более сложных проверок по данным, чем это допускается при использовании ограничения CHECK. Дело в том, что триггеры могут обращаться к столбцам таблиц, отличных от таблицы, по которой они определены, в то время как ограничения CHECK действуют в рамках только одной таблицы.

Обращаем внимание, что триггер не следует использовать как замену декларативного ограничения. Например, не надо создавать триггер, который проверяет наличие значения в столбце первичного ключа одной таблицы, чтобы определить, можно ли вставить это значение в соответствующий столбец другой таблицы. В этой ситуации прекрасно подойдет ограничение FOREIGN KEY.

Триггеры также полезно использовать для выполнения нескольких операций в ответ на одно событие модификации данных.

***4.3.3.2. Создание триггера.*** Для создания триггера необходимо быть владельцем базы данных и, более того, владельцем таблицы, для которой триггер создается.

Триггер создается командой CREATE TRIGGER, базовый формат которой имеет следующий синтаксис:

```
CREATE TRIGGER <имя_триггера>
ON <имя_таблицы> | <имя_представления>
AFTER | INSTEAD OF [INSERT | UPDATE | DELETE]
AS
<команда SQL>
```

- имя_триггера – указывается имя триггера, которое должно соответствовать стандартным соглашениям об именах объектов MS SQL Server и быть уникальным в базе данных. Как правило, имя триггера отражает тип операций и имя таблицы, над которой производится операция.
- имя_таблицы | имя_представления – каждый триггер ассоциируется с определенной таблицей или представлением, имя которой (которого) указывается после слова ON.
- AFTER | INSTEAD OF – устанавливается тип триггера. Можно использовать один из двух типов. Триггеры типа AFTER вызываются после выполнения действия, запускающего триггер. Триггеры типа INSTEAD OF выполняются вместо операции, запускающей триггер, т. е., по сути, операция (добавление, изменение или удаление) вообще не выполняется. Триггеры AFTER можно создавать только для таблиц, а триггеры INSTEAD OF – как для таблиц, так и для представлений.
- INSERT, UPDATE, DELETE – определяют операции, которые активизируют триггер. Для триггера AFTER допускается применение сразу нескольких операций, например UPDATE и INSERT. В этом случае операции указываются через запятую. Для триггера INSTEAD OF можно определить только одну операцию.
- AS – ключевое слово, задающее начало определения тела триггера.
- команда SQL – в Transact-SQL тело триггера может содержать любое количество команд SQL, если они заключены в ключевые слова BEGIN и END.

При вызове триггера будут выполнены команды SQL, указанные после ключевого слова AS. В тело триггера можно поместить практически все команды языка SQL, включая программные конструкции, такие как IF и WHILE.

В большинстве СУБД действуют следующие ограничения.

• Нельзя использовать в теле триггера операции создания объектов базы данных (новой базы данных, новой таблицы, новой хранимой процедуры, нового триггера, новых представлений).

• Нельзя использовать в триггере команду удаления объектов DROP для всех типов базовых объектов базы данных.

• Нельзя использовать в теле триггера команды изменения базовых объектов ALTER TABLE, ALTER DATABASE.

• Нельзя изменять права доступа к объектам базы данных, т. е. выполнять команду GRANT или REVOKE.

• В отличие от хранимых процедур, триггер не может возвращать никаких значений, он запускается автоматически сервером и не может связаться самостоятельно ни с одним клиентом.

• Внутри триггера не допускается выполнение восстановления резервной копии БД или журнала транзакций.

Большинство этих ограничений связано с тем, что в случае возвращения в начальное состояние результатов работы операций обновления, добавления или удаления, вызвавших активизацию триггера, откатить (ROLLBACK) результаты выполнения команд внутри триггера невозможно.

Специально для триггеров Transact-SQL определяет две виртуальные таблицы, структура которых идентична структуре таблицы, с которой связан триггер (одинаковые столбцы и типы данных). Они хранятся в оперативной памяти, а не на диске. Эти таблицы называются Deleted и Inserted. Они заполняются строками из модифицируемой таблицы, причем их конкретное содержимое зависит от выполняемой операции:

• команда INSERT – в таблице Inserted содержатся все строки, которые пользователь пытается вставить в таблицу. В таблице Deleted не будет ни одной строки. После завершения триггера все строки из таблицы Inserted переместятся в исходную таблицу;

• команда DELETE – в таблице Deleted будут содержаться все строки, которые пользователь попытается удалить. Триггер может проверить каждую строку и определить, разрешено ли ее удаление. В таблице Inserted не окажется ни одной строки;

• команда UPDATE – при ее выполнении в таблице Deleted находятся старые значения строк, которые будут удалены при успешном завершении триггера. Новые значения строк содержатся в табли-

це Inserted. Эти строки добавятся в исходную таблицу после успешного выполнения триггера.

К этим двум виртуальным таблицам можно обращаться из тела триггера и их данные можно использовать в триггере наряду с данными всех остальных таблиц. Можно использовать эти временные таблицы, чтобы увидеть влияние, оказанное каким-либо событием модификации данных на исходную таблицу. После завершения работы триггера эти таблицы больше недоступны.

Для получения информации о количестве строк, которое будет изменено при успешном завершении триггера, можно использовать глобальную переменную @@ROWCOUNT. Она возвращает количество строк, обработанных последней командой. Следует помнить, что триггер запускается не при попытке изменить конкретную строку, а в момент выполнения команды изменения. Одна такая команда воздействует на множество строк, поэтому триггер должен обрабатывать все эти строки.

Если триггер обнаружил, что из 100 вставляемых, изменяемых или удаляемых строк только одна не удовлетворяет тем или иным условиям, то никакая строка не будет вставлена, изменена или удалена. Такое поведение обусловлено требованиями транзакции: должны быть выполнены либо все модификации, либо ни одной.

Триггер выполняется как неявно определенная транзакция, поэтому внутри триггера допускается применение команд управления транзакциями. В частности, при обнаружении нарушения ограничений целостности для прерывания выполнения триггера и отмены всех изменений, которые пытался выполнить пользователь, необходимо использовать команду ROLLBACK TRANSACTION. Для фиксации изменений, внесенных при выполнении транзакции, следует использовать команду COMMIT TRANSACTION.

***4.3.3.3. Примеры использования триггеров.*** Приведем примеры создания нескольких типов триггеров. Примеры не слишком сложны и разнообразны, но они дают понятие о способах применения триггеров.

***Создание триггера типа INSERT.*** В этом примере создадим триггер типа INSERT (триггер, который активизируется при выполнении команды INSERT) по таблице ПРИХОД. Когда в таблицу

ПРИХОД добавляется новая запись, необходимо количество имеющегося товара на складе увеличить на поступившее количество.

Следующая команда Transact-SQL создает триггер InsertPrixod, который вызывает автоматическое выполнение описанных выше изменений:

```
CREATE TRIGGER InsertPrixod
  ON Приход
  AFTER INSERT
   AS
    UPDATE Склад
        SET S.Количество = S.Количество + I. Количество
        FROM Склад S, Inserted I
```

Обратите внимание, что фрагмент кода обращается к таблице Inserted, причем в списке таблиц базы данных эта таблица отсутствует. Столбцы таблицы Inserted в точности совпадают со столбцами таблицы ПРИХОД. В данном случае таблица Inserted содержит копии каждой строки таблицы ПРИХОД, которые будут добавлены только в случае успешного завершения транзакции. Эта таблица и ее значения применяются при выполнении любой операции сравнения для проверки правильности транзакции.

Глобальная переменная @@ERROR используется для обработки ошибок при выполнении транзакции. О транзакциях более подробно будет рассказано в параграфе 4.3.5.

***Создание триггера типа UPDATE.*** Теперь создадим UPDATE-триггер, который будет просматривать столбец «Цена» при модификации таблицы СКЛАД, чтобы убедиться, что цена товара не возросла более чем на 10 %. В противном случае будет использована команда ROLLBACK TRAN, которая выполнит откат данного триггера и команды, вызвавшей триггер. Таблицы Deleted и Inserted используются в данном примере для проверки изменения цены. Ниже приводится определение этого триггера:

```
CREATE TRIGGER UpdatePrice
ON Склад
AFTER UPDATE
AS
```

```
DECLARE @orig_price money, /*Существующая цена*/
                @new_price money /*Новая цена*/
SELECT    @orig_price = Цена from Deleted
SELECT    @new_price = Цена from Inserted
IF (@new_price > (@orig_price * 1.10))
ROLLBACK TRAN
```

Можно также создать триггер, выполняющий работу только в случае обновления конкретного столбца. Для принятия решения о продолжении обработки в триггере может быть применена команда IF UPDATE.

Например, снова создадим предыдущий триггер, но на этот раз будем использовать команду IF UPDATE, чтобы указать, что триггер проверяет столбец «Цена», только если был обновлен сам этот столбец:

```
CREATE TRIGGER UpdatePrice1
ON Склад
AFTER UPDATE
AS
IF UPDATE (Цена)
  BEGIN
    DECLARE @orig_price money, /*Существующая цена*/
                    @new_price money /*Новая цена*/
    SELECT    @orig_price = Цена
       FROM  Deleted
    SELECT  @new_price = Цена
      FROM  Inserted
    IF (@new_price > (@orig_price * 1.10))
      ROLLBACK TRAN
END
```

Теперь в случае модификации одного или нескольких атрибутов в таблице СКЛАД, за исключением атрибута «Цена», триггер пропустит команды между ключевыми словами BEGIN и END в команде IF UPDATE, т. е. фактически будет пропущен весь триггер.

***Триггеры вставки и обновления.*** Триггеры вставки (INSERT) и обновления (UPDATE) особенно удобны, поскольку они могут поддерживать условия ссылочной целостности и обеспечивать правильность данных перед их вводом в таблицу.

В приведенном ниже примере триггер выполняется всегда, когда в таблицу ДОКУМЕНТ вставляется строка или выполняется ее модификация. Если дата, указанная в приходном документе, больше 25 декабря каждого года, то строка в таблицу ДОКУМЕНТ не вводится. При попытке добавления или модификации более чем одной строки также происходит отмена действия. Для подсчета числа добавляемых или модифицированных строк используется системная переменная @@ROWCOUNT.

```
CREATE TRIGGER Ins_UpdDoc
ON Документ
AFTER INSERT, UPDATE
AS
/* Проверка количества модифицируемых строк и запрещение
добавления или модификации более одной строки за один раз */
IF @@ROWCOUNT > 1
    BEGIN
      ROLLBACK TRAN
      RAISERROR('За один раз можно изменить только одну строку', 16, 10)
    END
/* Объявить необходимые локальные переменные */
DECLARE @nDay DateTime,
            @nMonth DateTime
/* Найти информацию о добавленной записи */
SELECT  @nDay = DatePart(Dd, I. Дата)
    FROM Inserted I
    SELECT  @nMonth = DatePart(Mm, I. Дата)
    FROM Inserted I
/* Проверить критерий отказа */
IF (@nDay > 25 AND @nMonth = 12)
BEGIN
    ROLLBACK TRAN
    RAISERROR('Принимаются товары, поступившие до 25 декабря включительно', 16, 10)
END
```

Если теперь попытаться вставить или обновить запись в таблице, то при несоблюдении заданного условия получим соответствующее сообщение об ошибке, которое формируется командой RAISERROR.

Применение команды RAISERROR – это самый простой способ передачи вызывающему процессу или пользователю подробной и конкретной информации об ошибке. Команда RAISERROR дает возможность указать текст сообщения, уровень серьезности ошибки, состояние информации и скомбинировать все это для пользователя в описательное сообщение. Степень серьезности ошибки – указание на то, какие меры следует принимать с учетом этой ошибки. В рассматриваемом примере параметр уровня серьезности ошибки задан 16, что говорит о том, что эту ошибку должен исправить пользователь. Обозначение состояния представляет собой произвольную величину. Этот параметр команды RAISERROR был введен в действие с учетом того, что одна и та же ошибка может возникнуть в нескольких местах кода. А параметр с обозначением состояния предоставляет возможность передать вместе с сообщением своего рода маркер участка кода, который показывает, где именно произошла ошибка. Число, обозначающее состояние, может находиться в пределах от 1 до 127. В нашем примере параметр состояния информации равен 10.

***Триггер удаления.*** Как уже ранее отмечалось, удаление строк из родительской таблицы не приводит к нарушению ссылочной целостности базы данных. Однако в некоторых случаях целесообразно запретить выполнение такой операции, если удаляемая строка ссылается на строки в родительской таблице.

Таблица ПРИХОД – дочерняя по отношению к таблице ДОКУМЕНТ. Разработаем триггер, который выполняется всегда, когда пользователь пытается удалить строку из таблицы ПРИХОД.

```
CREATE TRIGGER DelPr
ON Приход
AFTER DELETE
AS
/* Проверка количества модифицируемых строк и запрещение удаления более одной строки за один раз */
IF @@ROWCOUNT > 1
   BEGIN
     ROLLBACK TRAN
    RAISERROR('За один раз можно удалить только одну строку', 16, 10)
    END
```

```
 /* Объявление временной переменной для сохранения уничтожаемой информации*/
DECLARE @N_Ном_док Int
/* Получение значения удаляемой строки */
SELECT @N_Ном_док = D. Ном_док
FROM Приход P, Deleted D
WHERE P.Ном_док = D. Ном_док
IF EXISTS (SELECT * FROM Документ
            WHERE Ном_док = @N_Ном_док)
  BEGIN
   ROLLBACK TRAN
   RAISERROR('Эта информация не может быть удалена, поскольку имеется соответствующая запись в таблице ДОКУМЕНТ', 16, 10)
END
```

Обратите внимание, что фрагмент кода обращается к новой таблице Deleted. В данном случае эта таблица содержит копию удаляемой строки. Эта таблица и ее значения применяются при выполнении любого сравнения для проверки правильности транзакции.

***Триггеры INSTEAD OF.*** Триггер типа INSTEAD OF заменяет соответствующее действие, которое запустило его. Этот триггер выполняется после создания соответствующих таблиц Inserted и Deleted, но перед выполнением проверки ограничений целостности или каких-либо других действий.

Триггеры INSTEAD OF можно создавать как для таблиц, так и для представлений. Когда команда Transact-SQL ссылается на представление, для которого определен триггер INSTEAD OF, СУБД выполняет этот триггер вместо выполнения любых действий с любой таблицей. Данный тип триггера всегда использует информацию в таблицах Inserted и Deleted, созданных для представления.

Создадим представление с именем SellersOrders со ссылкой на таблицы ПОСТАВЩИК и ДОКУМЕНТ.

```
CREATE VIEW SupDoc
AS
SELECT P.Код_постав, P.Название, D.Ном_док, D.Дата
FROM Поставщик P INNER JOIN Документ D
ON P.Ном_док = D.Ном_док
```

Попытаемся удалить из этого представления строку, у которой значение атрибута Название = ‘Петров И.В.’:

```
DELETE FROM SupDoc
WHERE Название = ‘Петров И.В.’
```

Так как представление SupDoc немодифицируемое, то появится сообщение об ошибке: «Невозможно обновить представление или функцию “SupDoc”, так как изменение влияет на несколько базовых таблиц».

Чтобы обойти эту ситуацию, создадим на базе представления для операции удаления триггер типа INSTEAD OF с именем DeletePos:

```
CREATE TRIGGER DeletePos
ON SupDoc
INSTEAD OF DELETE
AS
/*Удаление строки из таблицы ПРОДАВЕЦ */
DELETE FROM ПРОДАВЕЦ
WHERE Название IN (SELECT Название FROM deleted)
```

И если теперь ввести команду для удаления из этого представления строки, удовлетворяющей условию «Название = ‘Петров И.В.’», то произойдет активизация триггера INSTEAD OF. В результате из таблицы ПОСТАВЩИК будет удалена строка, соответствующая продавцу Петрову И.В.

## Вопросы для самопроверки

1. Что такое триггер?
2. Каковы компоненты триггера?
3. Триггеры каких типов существуют?
4. Допускается ли внутри триггера применение команд управления транзакциями?
5. В чем преимущества использования триггеров?
6. Какие основные правила программирования триггеров AFTER?
7. Какие основные правила программирования триггеров INSTEAD OF?
8. Какую информацию содержат таблицы Inserted и Deleted?

## Практические задания

Дана реляционная схема базы данных КЛИЕНТЫ:

КЛИЕНТ (Код_клиента, Наименование, Годовой_доход, Тип_заказчика [Производитель, Оптовый_продавец, Торговая_компания])

ОТГРУЗКА (Номер_отгрузки, Код_клиента, Вес, Номер_грузовика, Город, Дата)

ВОДИТЕЛЬ (Номер_отгрузки, Имя_водителя)

ГОРОД (Название, Число_жителей)

1. Напишите триггер, который при вводе новой записи в таблицу ОТГРУЗКА добавляет в таблицу ГОРОД новую запись, если такой город в таблице ГОРОД отсутствует.

2. Напишите триггер, запрещающий оформлять отгрузку на водителя, если он в данном месяце перевез грузов весом более 100 000 кг.

3. Напишите триггер, который запрещает удалять клиента из таблицы КЛИЕНТ, если на него в данный день оформлена отгрузка.

### *4.3.4. Функции пользователя*

***4.3.4.1. Понятие функции пользователя.*** Современные СУБД дают возможность разработчикам баз данных создавать собственные функции, добавляющие и расширяющие функции, предоставляемые системой.

Основное назначение функций – это реализация каких-либо небольших конечных алгоритмов для многократного их использования. Причем обращаться к функциям можно непосредственно в выражениях.

Таким образом, с помощью функций решается вопрос о сохранении разработанного кода для дальнейшего применения. Эту задачу можно было бы реализовать с помощью хранимых процедур, однако их архитектура не позволяет использовать процедуры непосредственно в выражениях, так как они требуют промежуточного присвоения возвращенного значения переменной, которая затем и указывается в выражении. Естественно, подобный метод применения программного кода не слишком удобен.

Функции пользователя представляют собой самостоятельные объекты базы данных, такие, например, как хранимые процедуры или триггеры. Функция пользователя располагается в определенной базе данных и доступна только в ее контексте.

В функциях можно обращаться к данным и проводить различные расчеты, при этом можно программировать на языке Transact-SQL, используя переменные, условные конструкции, циклы, и даже вызывать другие функции.

В СУБД MS SQL Server имеются следующие *классы функций пользователя*:

- Scalar – функции возвращают обычное скалярное значение, чем похожи на встроенные функции. Каждая функция может включать в себя множество команд, объединяемых в один блок с помощью конструкции BEGIN...END;
- Inline – функции содержат всего одну команду SELECT и возвращают пользователю набор данных в виде таблицы (значения типа данных TABLE). Результат функции похож на объект просмотра, но имеет большие возможности благодаря использованию параметров;
- Multi-statement – многооператорные функции, которые возвращают таблицу, созданную одной или несколькими командами языка Transact-SQL, чем напоминают хранимые процедуры. В теле функции может находиться множество команд SQL (INSERT, UPDATE и т. д.). Именно с их помощью и формируется набор данных, который должен быть возвращен после выполнения функции. В отличие от процедур, на такие функции можно ссылаться в предложении WHERE как на объект просмотра.

Вызов функции любого класса осуществляется указанием ее имени, после которого в круглых скобках можно задать один или несколько аргументов. Аргумент – это значение или выражение, которое передается параметру функции, объявленному в теле функции. При вызове функции, когда для ее параметров не определены значения по умолчанию, для всех этих параметров необходимо предоставить аргументы в том же самом порядке, в каком эти параметры определены в команде CREATE FUNCTION.

Пользовательские функции сходны с хранимыми процедурами, но, в отличие от них, могут применяться в запросах так же, как и си-

стемные встроенные функции. Пользовательские функции, возвращающие таблицы, могут стать альтернативой просмотрам. Просмотры ограничены одним выражением SELECT, а пользовательские функции способны включать в себя дополнительные выражения, что позволяет создавать более сложные и мощные конструкции.

***4.3.4.2. Скалярные функции Scalar.*** Создание и изменение функции данного типа выполняется с помощью команды

```
CREATE | ALTER} FUNCTION <имя_функции>
([<@имя_параметра> <тип_данных>
 [=<значение_по_умолчанию >]])
 RETURNS <скалярный_тип_данных>
 [AS]
BEGIN
<тело_функции>
RETURN <скалярное_выражение>
END
```

Рассмотрим назначение параметров команды.

Функция может содержать один или несколько входных параметров либо не содержать ни одного. Каждый параметр должен иметь уникальное в пределах создаваемой функции имя и начинаться с символа @. После имени указывается тип данных параметра. Дополнительно можно указать значение, которое будет автоматически присваиваться параметру по умолчанию, если пользователь явно не указал значения соответствующего параметра при вызове функции. Передаваемые параметры указываются в круглых скобках. В этом состоит отличие объявлений параметров функций и процедур.

С помощью конструкции RETURNS <скалярный_тип_данных> указывается, какой тип данных будет иметь возвращаемое функцией значение.

Между ключевыми словами BEGIN...END указывается набор команд, они и будут телом функции.

Когда в ходе выполнения кода функции встречается ключевое слово RETURN, выполнение функции завершается и как результат ее вычисления возвращается значение, указанное непосредственно после слова RETURN. Отметим, что в теле функции разрешается использо-

вание нескольких команд RETURN, которые могут возвращать различные значения. В качестве возвращаемого значения допускаются как обычные константы, так и сложные выражения. Единственное условие – тип данных возвращаемого значения должен совпадать с типом данных, указанным после ключевого слова RETURNS.

В качестве примера создадим функцию с именем GetSum скалярного типа для вычисления суммарного количества товара, поступившего на предприятие за определенную дату.

```
CREATE FUNCTION GetSum
(@dt DATETIME)
RETURNS INT
AS
BEGIN
 DECLARE @s INT
 SET @s = (SELECT SUM(Количество)
           FROM Документ INNER JOIN Приход
           ON Документ.Ном_док = Приход.Ном_док
           WHERE Дата =@dt)
 RETURN (@s)
END
```

В качестве входного параметра используется дата. Функция возвращает значение целого типа, полученное из команды SELECT путем суммирования количества товара из таблицы ПРИХОД. Условие отбора записей для суммирования – равенство даты сделки значению входного параметра функции.

Проиллюстрируем обращение к функции пользователя: определим количество товара, поступившего за 20.10.2020.

Следующий пример использует функцию GetSum в команде SELECT:

SELECT dbo.GetSum(’20.10.2020’) AS ‘Количество товара, поступившего на предприятие’

Команда SELECT возвращает результат выполнения функции GetSum (рис. 4.20).

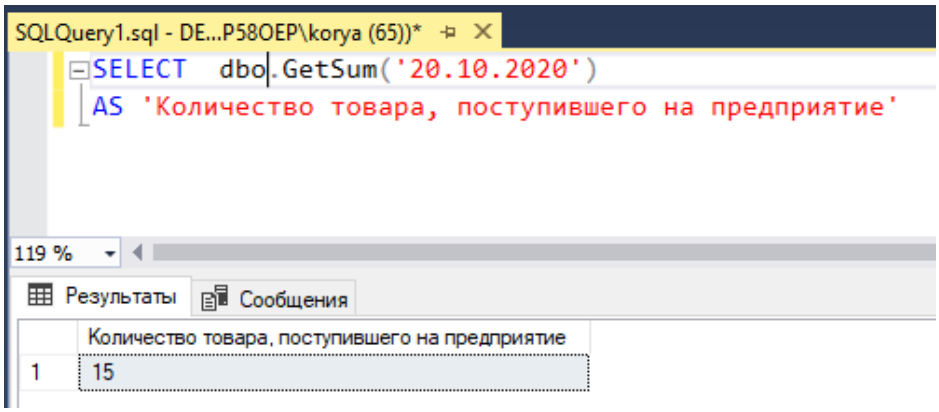



Рис. 4.20. Пример использования функции в команде SELECT

В командах Transact-SQL имена функций необходимо задавать, используя составные имена, состоящие из двух частей: имя_схемы. имя_функции. Так как при создании учебной базы данных не была указана схема, то в ней используется схема dbo – схема по умолчанию для всех пользователей. Поэтому в рассмотренном примере в имени функции при ее вызове приведено имя схемы dbo.

Функции можно использовать не только в команде SELECT, но и напрямую, присваивая значение переменной, например:

```
DECLARE @kol INT
SET @kol = dbo.GetSum(’20.10.2020’)
SELECT @kol AS ‘Количество товара, поступившего на предприятие’
```

В этом примере объявляется переменная @kol типа INT. Именно такой тип возвращает функция. В следующей строке с помощью команды SET переменной @kol присваивается результат выполнения функции GetSum. Команда SELECT выводит значение переменной @kol (рис. 4.21).

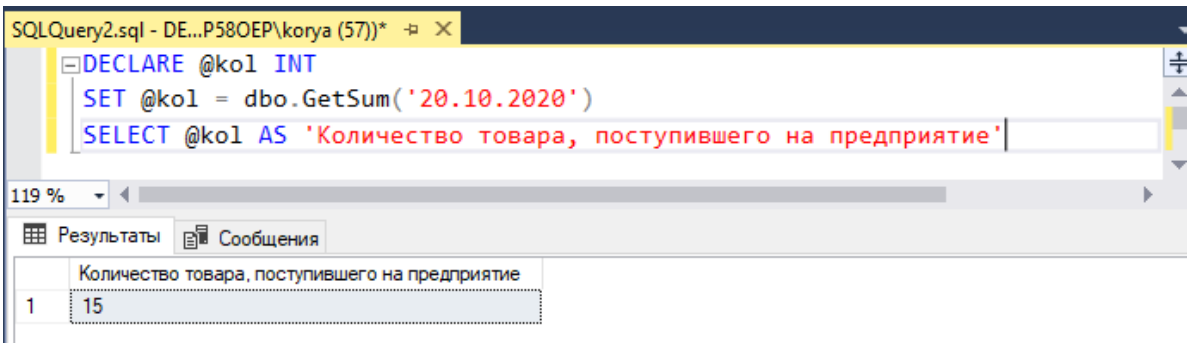



Рис. 4.21. Пример использования функции для присваивания значения переменной

***4.3.4.3. Табличные функции Inline.*** Создание и изменение функции этого типа выполняется с помощью команды

```
CREATE | ALTER FUNCTION <имя_функции>
( [ <@имя_параметра> <тип_данных>
 [=<значение по умолчанию>]])
RETURNS TABLE
 [AS]
RETURN [ ( ] <Команда SELECT> [ ) ]
```

Основная часть параметров, используемых при создании табличных функций, аналогична параметрам скалярной функции. Тем не менее создание табличных функций имеет свою специфику.

После ключевого слова RETURNS всегда должно указываться ключевое слово TABLE. Таким образом, функция данного типа должна возвращать значение типа данных TABLE.

Обращаем внимание, что определение типа TABLE не содержит списка столбцов, т. е. структура значения типа TABLE не указывается явно. Система управления базами данных автоматически создает для значения TABLE ту структуру, которую имеет таблица, возвращаемая запросом SELECT в теле функции.

Когда пишется команда SELECT, образующая тело функции, то все столбцы, возвращаемые этим запросом, должны содержать имена. Это в первую очередь касается вычисляемых столбцов. Если один из столбцов не имеет имени, то результатом выполнения команды CREATE FUNCTION будет ошибка.

Создадим функцию табличного типа для определения наименований товара с наибольшей стоимостью его остатков на складе.

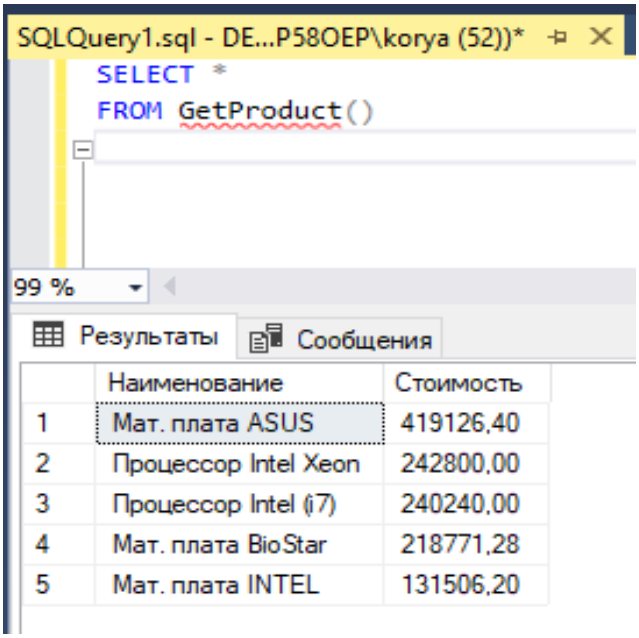
```
CREATE FUNCTION GetProduct()
RETURNS TABLE
AS
RETURN
(
SELECT TOP 5 Наименование, Количество*Цена AS 'Стоимость'
  FROM Склад
  ORDER BY 2 DESC
)
```

Созданная функция не содержит входных параметров. Несмотря на это, после имени должны идти круглые скобки, в которых не надо ничего писать. Если не указать скобок, то сервер вернет ошибку и функция не будет создана.

Возвращаемое функцией значение типа TABLE может быть использовано непосредственно в запросе, т. е. в предложении FROM. Например:

```
SELECT *
FROM GetProduct()
```

Результат выполнения запроса приведен на рис. 4.22.




| | Наименование | Стоимость |
|---|---|---|
| 1 | Мат. плата ASUS | 419126,40 |
| 2 | Процессор Intel Xeon | 242800,00 |
| 3 | Процессор Intel (i7) | 240240,00 |
| 4 | Мат. плата BioStar | 218771,28 |
| 5 | Мат. плата INTEL | 131506,20 |

Рис. 4.22. Пример использования
табличной функции в предложении FROM

В этом примере команда SELECT выводит все строки таблицы, полученной функцией GetProduct(). Вывод можно ограничить определенными строками с помощью условий в предложении WHERE. В следующем примере выбираются из результата функции только те строки, в которых атрибут «Стоимость» содержит значение больше 200 000 рублей:

```
SELECT *
FROM GetProduct()
WHERE Стоимость > 200000
```

Результат запроса представлен на рис. 4.23.

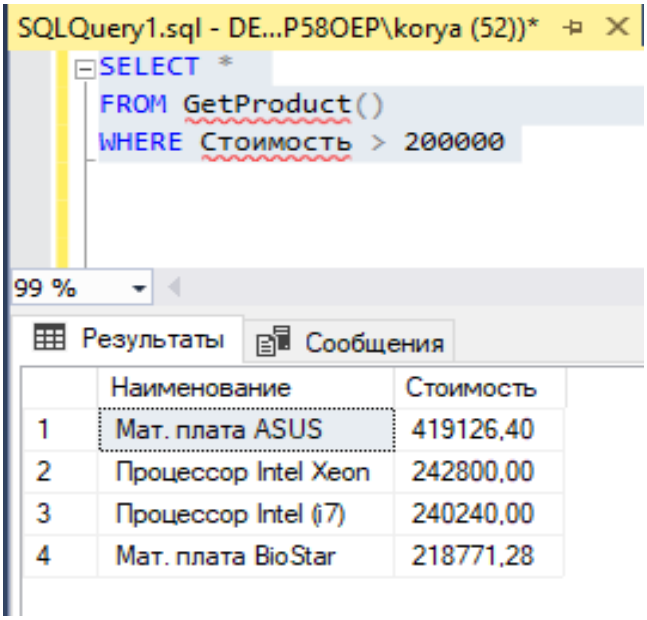



Рис. 4.23. Пример вывода ограниченного количества строк, полученных табличной функцией

***4.3.4.4. Многооператорные функции Multi-statement.*** Создание и изменение функций типа Multi-statement выполняется с помощью следующей команды:

```
CREATE | ALTER FUNCTION < имя_функции>
([<@имя_параметра> <тип_данных>
 [ = <значение по умолчанию> ]])
RETURNS <@имя_параметра> TABLE
 <определение_таблицы>
 [AS]
BEGIN
<тело_функции>
RETURN
END
```

Использование большей части параметров рассматривалось при описании предыдущих функций.

Отметим, что функции данного типа, как и табличные, возвращают значение типа TABLE. Однако, в отличие от табличных функций, при создании функций Multi-statement необходимо явно задать

структуру возвращаемого значения. Она указывается непосредственно после ключевого слова TABLE и, таким образом, является частью определения возвращаемого типа данных. Синтаксис конструкции <определение_таблицы> полностью соответствует одноименным структурам, используемым при создании обычных таблиц с помощью команды CREATE TABLE.

Набор возвращаемых данных должен формироваться с помощью команд INSERT, выполняемых в теле функции. Кроме того, в теле функции допускается использование различных конструкций языка SQL, которые могут контролировать значения, размещаемые в выходном наборе строк. При работе с командой INSERT требуется явно указать имя того объекта, куда необходимо вставить строки. Поэтому в функциях типа Multi-statement, в отличие от табличных, необходимо присвоить какое-то имя объекту с типом данных TABLE. Оно и указывается как возвращаемое значение.

Завершение работы функции происходит, если появляется ключевое слово RETURN. В отличие от функций скалярного типа, при использовании команды RETURN не нужно указывать возвращаемое значение. Сервер автоматически возвратит набор данных типа TABLE, имя и структура которого были указаны после ключевого слова RETURNS. В теле функции может быть указано более одной команды RETURN.

Необходимо отметить, что работа функции завершается только при наличии команды RETURN. Это утверждение верно и в том случае, когда речь идет о достижении конца тела функции: самой последней командой должна быть команда RETURN.

Как уже отмечалось, тело многооператорной функции Multi-statement может содержать несколько команд языка SQL, что позволяет программировать любые действия и при этом получать результат в виде таблицы.

Рассмотрим простой пример, иллюстрирующий эту возможность многооператорных функций. Создадим функцию, возвращающую таблицу, состоящую из двух столбцов, в которых указывается название поставщика и общая сумма его поставок.

```
CREATE FUNCTION tabl_new
(
 --входной параметр
```

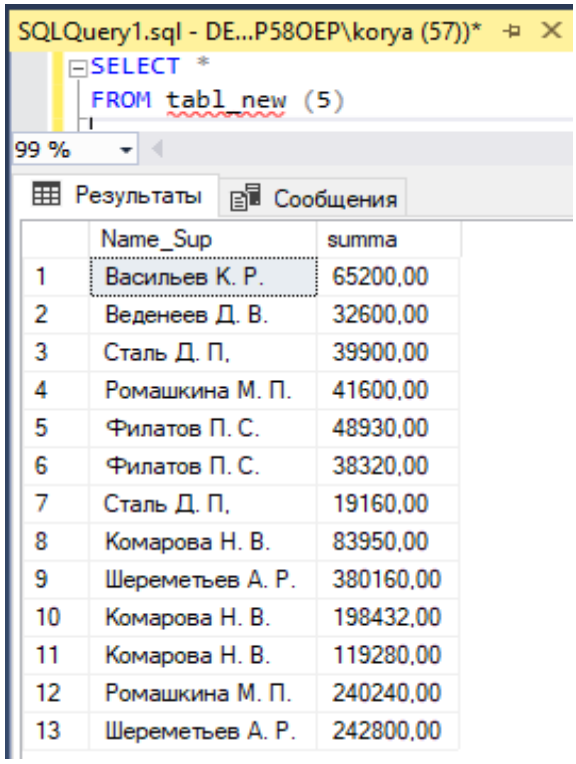
```
 @Inp_value INT
)
-- таблица с перечислением полей и их типов
RETURNS @ret TABLE
          (Name_Sup VarChar (20),
           summa  MONEY)
AS
BEGIN
     DECLARE @var MONEY
     IF @Inp_value >=0
                SET @var=1000
        ELSE
                SET @var=0
       --вставляем данные в возвращаемый результат
      INSERT @ret
      SELECT Поставщик.Название, Приход.Цена * Приход.Количество
      FROM Поставщик INNER JOIN Документ ON Поставщик.Код_постав = Документ.Код_постав INNER JOIN Приход ON Документ.Ном_док = Приход.Ном_док
      WHERE Приход.Цена * Приход.Количество > @var
      --возвращаем результат
      RETURN
END
```

В данном примере в качестве результата объявлена переменная @ret, которая является таблицей из двух полей "Name_Sup" типа VarChar длиной в 20 символов и «summa» типа MONEY. В теле функции в эту таблицу записываются значения из таблиц ПОСТАВЩИК, ДОКУМЕНТ и ПРИХОД, причем формирование таблицы зависит от значения входного параметра. Команда RETURN завершает выполнение функции.

В использовании такая функция ничем не отличается от рассмотренных ранее функций. Например, следующий запрос выбирает все данные, которые возвращает функция:

```
SELECT *
FROM tabl_new (5)
```

Результат запроса представлен на рис. 4.24.




| | Name_Sup | summa |
|---|---|---|
| 1 | Васильев К. Р. | 65200,00 |
| 2 | Веденеев Д. В. | 32600,00 |
| 3 | Сталь Д. П. | 39900,00 |
| 4 | Ромашкина М. П. | 41600,00 |
| 5 | Филатов П. С. | 48930,00 |
| 6 | Филатов П. С. | 38320,00 |
| 7 | Сталь Д. П. | 19160,00 |
| 8 | Комарова Н. В. | 83950,00 |
| 9 | Шереметьев А. Р. | 380160,00 |
| 10 | Комарова Н. В. | 198432,00 |
| 11 | Комарова Н. В. | 119280,00 |
| 12 | Ромашкина М. П. | 240240,00 |
| 13 | Шереметьев А. Р. | 242800,00 |

Рис. 4.24. Пример использования многооператорной функции

## Вопросы для самопроверки

1. Что такое функция?
2. Функции каких типов существуют?
3. В каких случаях используют скалярную функцию?
4. Чем отличаются функции Multi-statement от функции Inline?
5. Как активизируются функции?
6. В чем преимущества использования функций?

## Практические задания

1. Дана реляционная схема базы данных ГОСТИНИЦА:

ТИП НОМЕРА (Тип_номера, Количество_мест, Цена)
НОМЕР (Номер, Тип_номера)
КЛИЕНТ (Код_клиента, Город, ФИО)
КЛИЕНТ ПРОЖИВАЕТ (Код_клиента, Номер, Дата_прибытия, Дата_убытия)

Напишите функцию, в которой определяется, сколько раз останавливался в гостинице в 2022 г. Сидоров из Рязани.

2. Даны три таблицы:

ПРОДАВЕЦ (Код_продавца, Имя, Город, Комиссионные)

ЗАКАЗЧИК (Код_покупателя, ФИО, Рейтинг, Город, Код_продавца)

ПОКУПКА (Номер, Сумма, Дата, Код_продавца, Код_покупателя)

а) Напишите функцию для определения, максимальной суммы продаж продавца Петрова.

б) Напишите функцию, которая выводит максимальный рейтинг в каждом городе.

3. Дана база данных КЛИЕНТЫ:

КЛИЕНТ (Код_клиента, Наименование, Годовой_доход, Тип_заказчика [Производитель, Оптовый_продавец, Торговая_компания])

ОТГРУЗКА (Номер_отгрузки, Код_клиента, Вес, Номер_грузовика, Название_города, Дата)

ВОДИТЕЛЬ (Номер_отгрузки, Имя_водителя)

ГОРОД (Название, Число_жителей)

Напишите функцию, определяющую средний вес груза каждого клиента.

### *4.3.5. Транзакции и управление транзакциями*

***4.3.5.1. Понятие транзакции.*** Транзакции – одна из самых важных концепций баз данных. Механизм транзакций тесно связан с природой реляционных СУБД, поскольку именно транзакции обеспечивают:

- сохранение целостности данных;
- параллельную работу пользователей с базой данных;
- восстановление данных при откатах и сбоях.

Чтобы понять, как работают транзакции и для чего они нужны, рассмотрим пример, который наглядно показывает необходимость использования транзакций.

Допустим, имеется некоторая база данных, в которой хранятся заказы клиентов. Оформление каждого заказа приводит к нескольким изменениям в базе данных:

1) добавление нового заказа в таблицу ЗАКАЗЫ;

2) обновление фактического объема продаж для служащего, принявшего заказ;

3) обновление фактического объема продаж для офиса, в котором работает данный служащий;

4) обновление количества товара, имеющегося в наличии.

Чтобы в базе данных не возникли противоречия, четыре указанных изменения следует выполнить как одно целое. Если из-за системного сбоя или другой ошибки получится, что одна часть изменений была внесена, а другая – нет, то это нарушит целостность хранимых данных. При последующих вычислениях результаты окажутся неверными. Поэтому несколько изменений базы данных, которые вызваны одним событием, необходимо вносить по принципу «либо все, либо ничего». Это обеспечивается средствами обработки транзакций, имеющимися в SQL.

*Транзакция* – это последовательность команд языка SQL, выполняющаяся как единое целое и переводящая базу данных из одного согласованного состояния в другое согласованное состояние.

Команды, входящие в транзакцию, выполняют взаимосвязанные действия. Каждая команда решает часть общей задачи, но для того чтобы задачу можно было считать решенной, требуется выполнить все эти команды.

В реляционной СУБД транзакции подчиняются следующему правилу: *либо все команды будут выполнены успешно, либо ни одна из них не должна быть выполнена.*

Как следует из рис. 4.25, СУБД должна подчиняться этому правилу, даже если во время выполнения транзакции произойдет ошибка в программе или аппаратный сбой. В любом случае СУБД должна гарантировать, что при восстановлении базы данных после сбоя частичное выполнение транзакции в ней отражено не будет.

Целостность транзакции зависит, в частности, от программиста SQL. Программист должен знать, когда начинать и когда заканчивать транзакцию и в какой последовательности, чтобы модификации дан-

ных обеспечивали логическую согласованность и содержательность данных. Далее рассмотрим, как начинать и заканчивать транзакции.

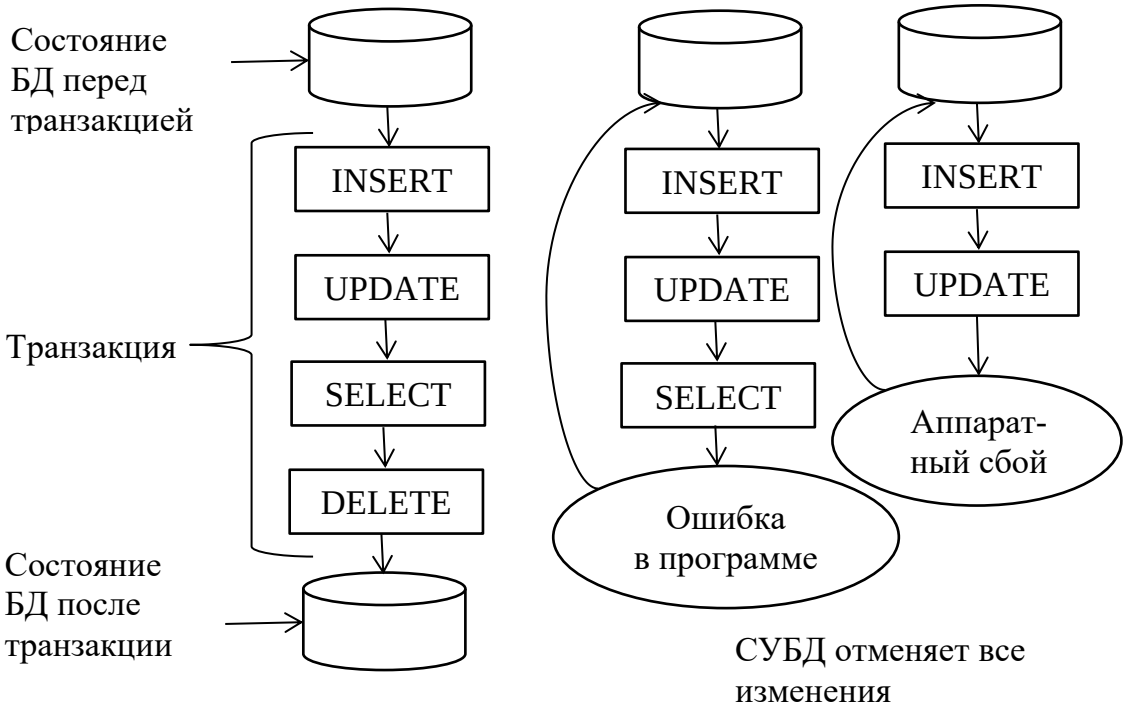



Рис. 4.25. Механизм транзакции в SQL

***4.3.5.2. ACID-свойства транзакций.*** При выполнении транзакции система управления базами данных должна придерживаться определенных правил обработки набора команд, входящих в транзакцию. В частности, разработаны четыре правила, известные как требования ACID, которые гарантируют правильность и надежность работы системы: атомарность (неделимость) (Atomicity), согласованность (Consistency), изолированность (Isolation), устойчивость (Durability).

***Атомарность.*** Транзакция должна выполняться как элементарная (атомарная) операция – отсюда термин «атомарность». Чтобы транзакция считалась успешно выполненной, должна быть выполнена каждая команда этой транзакции. При неудачном выполнении одной из команд вся транзакция считается неуспешной и происходит отмена всех модификаций, внесенных с момента начала транзакции.

***Согласованность.*** Означает, что после окончания транзакции все данные остаются в согласованном состоянии (сохраняется целостность данных), будь то успешно или неуспешно завершенная

транзакция. Перед началом любой транзакции база данных должна быть в согласованном состоянии, а это означает поддержку целостности данных и правильность внутренних структур. После выполнения транзакции база данных тоже должна быть в согласованном состоянии: в новом состоянии для успешной транзакции или в том же состоянии, что и перед началом транзакции, в случае неуспешного ее завершения.

***Изолированность.*** Означает, что каждая транзакция действует так же, как если бы она была единственной в системе. Иными словами, модификации, выполняемые в одной транзакции, изолируются от модификаций, выполняемых параллельно в другой транзакции. Тем самым на любую транзакцию не влияет никакое значение, изменяемое другой транзакцией, пока это изменение не будет зафиксировано. В случае неуспешной транзакции ее модификации не будут оказывать влияния, поскольку будет выполнен откат (отмена) соответствующих изменений. Таким образом, конкурирующие за доступ к базе данных транзакции физически обрабатываются последовательно, изолированно друг от друга, но для пользователей это выглядит так, будто они выполняются параллельно.

***Устойчивость.*** Означает, что результаты завершённой транзакции не могут быть потеряны. После своего завершения она сохраняется в системе, которую ничто не может вернуть в исходное (до начала транзакции) состояние, т. е. происходит фиксация транзакции, означающая, что впоследствии данные могут быть извлечены из нее, независимо от последующих возможных сбоев в работе системы.

Указанные выше правила выполняет сервер. Программист лишь выбирает нужный уровень изоляции, заботится о соблюдении логической целостности данных и бизнес-правил. На него возлагаются обязанности по созданию эффективных и логически верных алгоритмов обработки данных. Он решает, какие команды должны выполняться как одна транзакция, а какие могут быть разбиты на несколько последовательно выполняемых транзакций. Следует по возможности использовать небольшие транзакции, т. е. включающие в себя как можно меньше команд и изменяющие минимум данных. Соблюдение этого требования позволит наиболее эффективным образом обеспечить одновременную работу с данными множества пользователей.

***4.3.5.3. Управление транзакциями в СУБД MS SQL Server.*** Под управлением транзакциями понимается способность управлять различными операциями над данными, которые выполняются внутри реляционной СУБД. Прежде всего имеется в виду выполнение команд INSERT, UPDATE и DELETE. Например, после создания таблицы (выполнения команды CREATE TABLE) не нужно фиксировать результат: создание таблицы фиксируется в базе данных автоматически. Точно так же с помощью отмены транзакции не удастся восстановить только что удаленную командой DROP TABLE таблицу.

MS SQL Server предоставляет различные способы обработки транзакций, которые можно определить для каждого соединения с базой данных. Любое соединение может использовать тот режим, который необходим для выполнения специфических для этого соединения требований. Вот эти режимы:

- автофиксация транзакций;
- явные транзакции;
- неявные транзакции.

***Режим автофиксации транзакций.*** По умолчанию MS SQL Server работает в режиме *автоматического начала транзакций*, когда каждая команда рассматривается как отдельная транзакция. Если команда выполнена успешно, то ее изменения фиксируются. Если при выполнении команды произошла ошибка, то сделанные изменения отменяются и система возвращается в первоначальное состояние. Отменить успешно выполненную команду невозможно.

Рассмотрим работу сервера в режиме автофиксации транзакций. Создадим таблицу с помощью следующей команды:

```
CREATE TABLE table1
(i INT NOT NULL PRIMARY KEY,
col1 VarChar(20) NOT NULL )
```

Теперь вставим в таблицу table1 три новые строки. Для этого выполним следующие команды, объединив их в пакет:

```
INSERT INTO table1 (i,col1) VALUES (1,'1 строка')
INSERT INTO table1 (i,col1) VALUES (2,NULL)
INSERT INTO table1 (i,col1) VALUES (3,'3 строка')
GO
```

В результате получим сообщение (рис. 4.26), которое информирует о том, что MS SQL Server не разрешает вставку значения NULL в столбец col1, потому что для этого столбца задано условие NOT NULL.

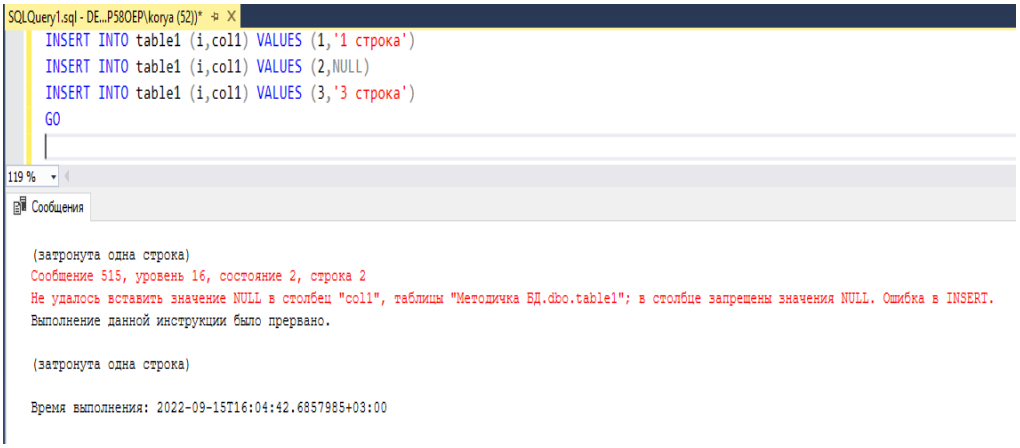



Рис. 4.26. Сообщение об ошибке при выполнении команды INSERT

Запрос

SELECT * FROM table1

позволяет проверить, были ли записи успешно вставлены в таблицу.

Как видно по рис. 4.27, вставка второй строки не выполнена, но первая и третья строки успешно вставлены.

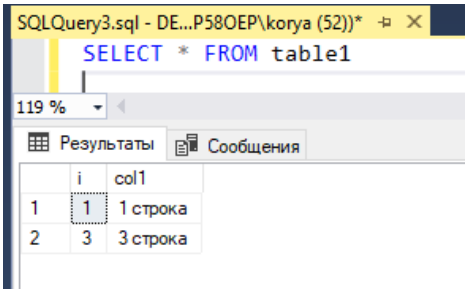



Рис. 4.27. Результат работы транзакции с автофиксацией

Когда MS SQL Server использует транзакции с автофиксацией, каждая команда рассматривается как транзакция. Если одна команда генерирует ошибку, соответствующая ей транзакция автоматически подвергается откату. Если команда успешно и без ошибок выполняется, то транзакция автоматически фиксируется. Следовательно, первая

и третья команды были зафиксированы, а вторая команда, вызвавшая ошибку, была отменена. Обратите внимание на то, что пакетное выполнение не определяет, следует ли обрабатывать все инструкции в пакете как единую транзакцию.

***Явные транзакции.*** Для явной транзакции разработчик определяет начало транзакции и момент, в который она должна быть зафиксирована или подвергнута откату, используя команды, представленные в табл. 4.3.

Таблица 4.3

Команды управления транзакциями и их применение

| Синтаксис | Применение |
|---|---|
| BEGIN TRAN[SACTION] [имя_транзакции\|@переменная] | Определяет начало транзакции и в журнале транзакций фиксирует первоначальные значения изменяемых данных и момент начала транзакции |
| COMMIT TRAN[SACTION] [имя_транзакции \| @переменная] | Предписывает серверу зафиксировать все изменения, сделанные в транзакции, после чего в журнале транзакций помечается, что изменения зафиксированы и транзакция завершена |
| SAVE TRAN[SACTION] [имя_транзакции\|@переменная] | Создание внутри транзакции точки сохранения: СУБД сохраняет состояние БД в текущей точке и присваивает сохраненному состоянию имя точки сохранения. Транзакция остается активной |
| ROLLBACK TRAN[SACTION] [имя_транзакции \| имя_точки_сохранения \| @переменная] | Прерывание транзакции: когда сервер встречает эту команду, происходит откат транзакции, восстанавливается первоначальное состояние системы и в журнале транзакций отмечается, что транзакция была отменена. Приведенная команда отменяет все изменения, сделанные в БД после оператора BEGIN TRANSACTION, или отменяет изменения, сделанные в БД после точки сохранения, возвращая транзакцию к месту, где был выполнен оператор SAVE TRANSACTION |

Следует помнить, что команды управления транзакциями не влияют на последовательность выполнения программы.

Кроме команд, представленных в табл. 4.3, приведем две системные переменные, которые можно использовать при управлении транзакциями:

@@TRANCOUNT – возвращает количество активных транзакций;

@@NESTLEVEL – возвращает уровень вложенности транзакций.

Применим явные транзакции. Выполним следующую команду, чтобы удалить данные таблицы table1:

```
DELETE INTO table1
```

Вставим те же три записи, что и в предыдущем случае, в таблицу table1. На этот раз команды INSERT вставим в явную транзакцию, поскольку необходимо, чтобы в таблицу или были вставлены все записи, или не было вставлено ни одной из них:

```
BEGIN TRAN
    INSERT INTO table1 (i,col1) VALUES (1,'1 строка')
    INSERT INTO table1 (i,col1) VALUES (2,NULL)
    INSERT INTO table1 (i,col1) VALUES (3,'3 строка')
COMMIT TRAN
```

В результате выполнения этой транзакции будет получено такое же сообщение (см. рис. 4.26), как и раньше, в котором сообщается, что MS SQL Server не разрешает вставку значения NULL в столбец col1, потому что для этого столбца задано условие NOT NULL.

Если теперь выполнить команду

```
SELECT * FROM table1,
```

чтобы проверить, были ли вставлены записи, то увидим тот же результат, что и в режиме автофиксации (см. рис. 4.27). Две из трех записей вставлены в таблицу, а одна, нарушающая ограничение NULL, не вставлена.

Почему при наличии ошибки при выполнении команды, входящей в транзакцию, не произошел откат транзакции? Дело в том, что ошибки, связанные с нарушением ограничений целостности данных, не влекут за собой автоматический откат всех изменений, внесенных текущей транзакцией. Откатить транзакцию можно только с помо-

щью команды ROLLBACK TRANSACTION. Как отмечалось ранее, обязанность разработчика – не только определить длину транзакции, но и то, должен ли выполняться откат. Поэтому в транзакцию необходимо добавить обработчик ошибок. Без обработчика ошибок MS SQL Server после ошибки просто обработает следующую команду, потому что пакет не отменяется. В рассмотренном примере MS SQL Server обрабатывает каждую команду INSERT и после этого обрабатывает команду COMMIT TRAN. Следовательно, получаем тот же результат, что и в режиме автофиксации.

***Обработка ошибок при выполнении транзакции.*** Обрабатывать ошибки при выполнении транзакции можно разными способами.

Первый способ заключается в использовании системной переменной @@ERROR, которая возвращает номер ошибки для последней выполненной команды Transact-SQL. Переменная имеет значение 0, если в предыдущей команде Transact-SQL не возникли ошибки.

Поскольку переменная @@ERROR очищается и сбрасывается для каждой выполняемой команды, надо сразу проверять ее значение после команды или сохранять значение в локальную переменную для последующей проверки.

Ниже приведен пример транзакции с обработкой ошибок с помощью переменной @@ERROR.

```
BEGIN TRAN
  DECLARE @OshibkiInsert1 int, @OshibkiInsert2 int,
                    @OshibkiInsert3 int
  INSERT INTO table1 (i,col1) VALUES (1,'1 строка')
              SET @OshibkiInsert1 = @@ERROR
  INSERT INTO table1 (i,col1) VALUES (2,NULL)
              SET @OshibkiInsert2 = @@ERROR
  INSERT INTO table1 (i,col1) VALUES (3,'3 строка')
             SET @OshibkiInsert3 = @@ERROR
 IF @OshibkiInsert1 = 0 AND @OshibkiInsert2 = 0 AND
                   @OshibkiInsert3 = 0
   COMMIT TRAN
   ELSE
   ROLLBACK TRAN
```

В листинге команда BEGIN TRAN сообщает серверу о начале транзакции. Далее создаются три локальные переменные: для фиксации ошибок @OshibkiInsert1, @OshibkiInsert2, @OshibkiInsert3. После первого запроса значение переменной @@ERROR присваивается переменной @OshibkiInsert1:

```
SET @OshibkiInsert1 = @@ERROR
```

То же самое делается после второго и третьего запросов.

Далее проверяются значения всех локальных переменных, которые должны быть равными нулю при отсутствии ошибок:

```
IF @OshibkiInsert1 = 0 AND @OshibkiInsert2 = 0 AND @OshibkiInsert3 = 0
```

В этом случае транзакция фиксируется при помощи команды COMMIT TRAN. В противном случае, если значение хотя бы одной из локальных переменных (@OshibkiInsert1, @OshibkiInsert2, @OshibkiInsert3) оказывается отличным от нуля, транзакция откатывается командой ROLLBACK TRAN. Термин «откат» означает, что все последствия транзакции будут отменены; данные на сервере будут выглядеть так, словно никаких изменений не было.

После выполнения транзакции ни одна запись не будет добавлена в таблицу table1 (рис. 4.28).

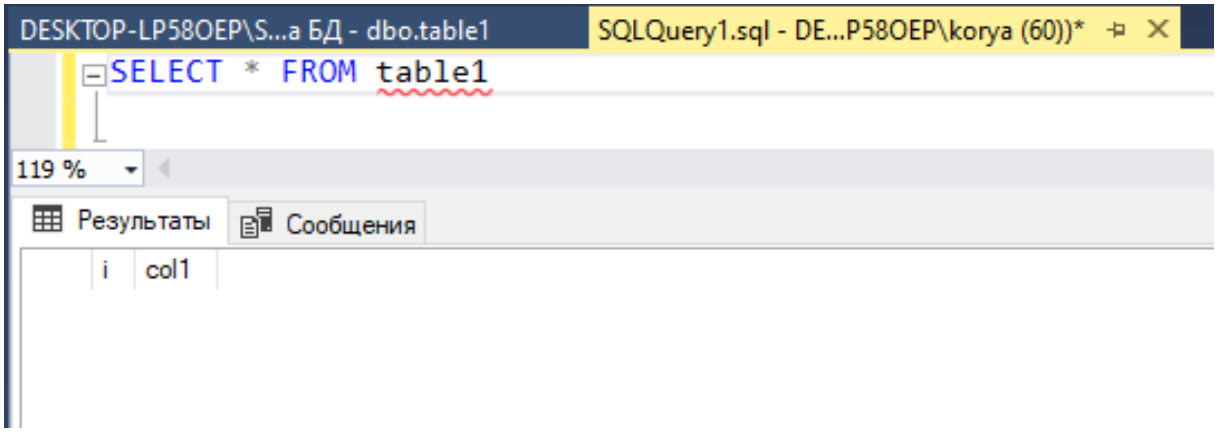



Рис. 4.28. Результат выполнения транзакции с обработкой ошибок

Рассмотрим еще один способ обработки ошибок. Чтобы добавить обработчик ошибок, можно использовать блоки TRY и CATCH.

Запустим транзакцию с обработчиком ошибок, как показано ниже.

```
BEGIN TRY
  BEGIN TRAN
    INSERT INTO table1 (i,col1) VALUES (1,'1 строка')
    INSERT INTO table1 (i,col1) VALUES (2,NULL)
    INSERT INTO table1 (i,col1) VALUES (3,'3 строка')
  COMMIT TRAN
END TRY
BEGIN CATCH
    ROLLBACK TRAN
END CATCH
```

Если выполнить команду SELECT, чтобы проверить, был ли выполнен откат транзакции:

```
SELECT * FROM table1,
```

то увидим, что эта команда, так же как и в предыдущем примере, не возвратила ни одной записи (см. рис. 4.28). Произошел откат всей транзакции. Когда во второй команде INSERT произошло нарушение ограничения, MS SQL Server перешел к блоку CATCH и выполнил откат транзакции.

Остается только одна проблема – этот код не возвращает каких-либо сообщений, которые информировали бы о том, что произошла ошибка. Это поведение управляется в блоке CATCH, в котором можно использовать особые функции для возвращения ошибок. Можно также использовать функцию RAISERROR для задания пользовательского текста сообщения об ошибке.

***Неявные транзакции.*** При работе в режиме неявного начала транзакций MS SQL Server автоматически начинает новую транзакцию, как только завершается предыдущая. В этом режиме явно не задается начало транзакции, но должен быть явно определен момент ее завершения. Транзакция продолжается до тех пор, пока пользователь явно не укажет команду отката (ROLLBACK TRANSACTION) или фиксации (COMMIT TRANSACTION) транзакции, после чего сервер автоматически начинает новую транзакцию. В итоге создается непрерывная цепь транзакций. Если для соединения устанавливается режим неявных транзакций, сервер автоматически начнет первую транзакцию, как только будет выполнена одна из следующих команд:

- CREATE – создание некоторого объекта базы данных;
- ALTER TABLE – изменение структуры таблицы;
- DELETE – удаление строк данных из таблицы;
- DROP – удаление некоторого объекта базы данных;
- TRUNCATE TABLE – усечение таблицы;
- SELECT – выборка данных из таблиц;
- UPDATE – изменение данных в таблице;
- INSERT – добавление строк в таблицу;
- OPEN – открытие курсора;
- FETCH – извлечение некоторых значений из курсора;
- GRANT – предоставление доступа к объектам базы данных;
- REVOKE – неявный отзыв доступа к объектам базы данных.

Неявные транзакции полезно использовать, когда запускаются сценарии с модификациями данных, которые требуется защитить внутри транзакции. Для этого надо включить неявный режим в начале сценария, выполнить необходимые модификации и отключить этот режим в конце сценария. Во избежание конфликтов параллельных операций следует отключать неявный режим после модификации данных и перед просмотром данных. Если вслед за операцией фиксирования следует оператор SELECT, то с него начинается новая транзакция в неявном режиме, и соответствующие ресурсы не будут освобождены, пока не будет фиксирована эта транзакция.

Режим неявных транзакций устанавливается на уровне соединения при помощи специальной команды

```
SET IMPLICIT_TRANSACTIONS ON
```

Для отмены режима неявных транзакций и возвращения к режиму автоматически фиксируемых транзакций следует выполнить команду

```
SET IMPLICIT_TRANSACTIONS OFF
```

Таким образом, MS SQL Server работает только в одном из двух режимов определения транзакций: автофиксации транзакций или неявных транзакций. MS SQL Server не может находиться в режиме исключительно явного определения транзакций. Этот режим работает поверх двух других.

***4.3.5.4. Вложенные транзакции.*** Вложенными называются транзакции, выполнение которых инициируется из тела уже активной транзакции.

Для создания вложенной транзакции пользователю не нужны какие-либо дополнительные команды. Он начинает новую транзакцию, не закрыв предыдущую. Завершение транзакции верхнего уровня откладывается до завершения вложенных транзакций. Если транзакция самого нижнего (вложенного) уровня завершена неудачно и отменена, то все транзакции верхнего уровня, включая транзакцию первого уровня, будут отменены. Кроме того, если несколько транзакций нижнего уровня были завершены успешно (но не зафиксированы), однако на среднем уровне (не самая верхняя транзакция) неудачно завершилась другая транзакция, то в соответствии с требованиями ACID произойдет откат всех транзакций всех уровней, включая успешно завершенные. Только когда все транзакции на всех уровнях завершены успешно, происходит фиксация всех сделанных изменений в результате успешного завершения транзакции верхнего уровня.

Каждая команда COMMIT TRANSACTION работает только с последней начатой транзакцией. При завершении вложенной транзакции команда COMMIT применяется к наиболее «глубоко» вложенной транзакции. Даже если в команде COMMIT TRANSACTION указано имя транзакции более высокого уровня, будет завершена транзакция, начатая последней. Иными словами, MS SQL Server, по сути, игнорирует операторы COMMIT внутри вложенных транзакций в том смысле, что внутренние транзакции не фиксируются в ожидании окончательного фиксирования или отката внешней транзакции, чтобы определить статус завершения всех внутренних транзакций.

Если команда ROLLBACK TRANSACTION используется на любом уровне вложенности без указания имени транзакции, то откатываются все вложенные транзакции, включая транзакцию самого верхнего уровня. В команде ROLLBACK TRANSACTION разрешается указывать только имя самой верхней транзакции. Имена любых вложенных транзакций игнорируются, и попытка их указания приведет к ошибке. Таким образом, при откате транзакции любого уровня вложенности всегда происходит откат всех транзакций. Если же требуется откатить лишь часть транзакций, можно использовать команду

SAVE TRANSACTION, с помощью которой создается точка сохранения.

Для демонстрации принципа работы с вложенными транзакциями рассмотрим пример. Допустим, имеется таблица

```
CREATE TABLE TestTable
(id INT PRIMARY KEY,
string VarChar(20) NOT NULL)
```

Кроме того, в базе данных создана хранимая процедура, осуществляющая вставку строк в указанную таблицу:

```
CREATE PROC SimpleInsert
(@id INT, @str VarChar(20))
AS
BEGIN TRAN
  INSERT INTO TestTable
  VALUES (@id, @str)
COMMIT TRAN
```

Обратите внимание, что вставка строки выполняется хранимой процедурой в рамках транзакции.

Посмотрим, как поведет себя хранимая процедура в случае использования внутри других транзакций:

```
BEGIN TRAN
   EXEC SimpleInsert 1, 'Это первая строка'
   SAVE TRAN point1
   EXEC SimpleInsert 2, 'Это вторая строка'
   ROLLBACK TRAN point1
   EXEC SimpleInsert 3, 'Это третья строка'
COMMIT TRAN
EXEC SimpleInsert 4, 'Это четвертая строка'
```

Первые три вызова хранимой процедуры выполняются в рамках одной транзакции. Четвертый вызов выполняется уже после ее фиксации. Если по окончании выполнения этих команд просмотреть содержимое таблицы

```
SELECT * FROM TestTable,
```

получим следующий результат (рис. 4.29).

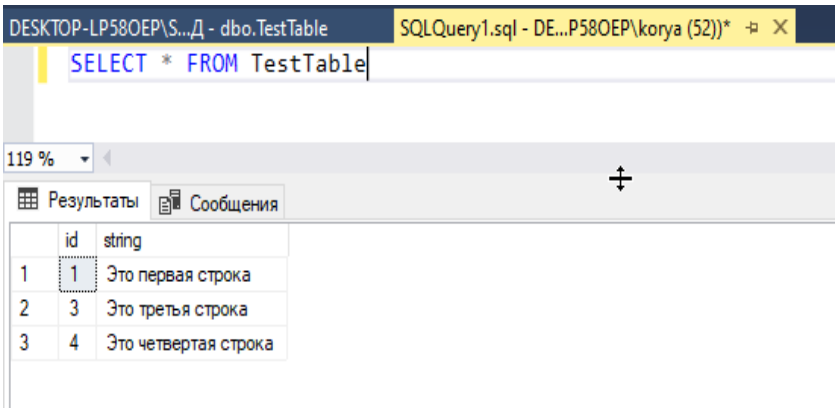



Рис. 4.29. Данные таблицы TestTable

Обратите внимание, что операторы в теле хранимой процедуры выполнялись в рамках самостоятельной транзакции. Однако, будучи вызванной из другой транзакции, транзакция в хранимой процедуре рассматривается как вложенная. Поэтому при частичном откате изменений вызвавшей ее транзакции будет выполнен откат и всех зафиксированных ей изменений.

## Вопросы для самопроверки

1. Что такое транзакция?
2. Чем определяется задание границ транзакций и кто задает эти границы?
3. Какими четырьмя свойствами характеризуются транзакции?
4. Что означает свойство транзакции «атомарность»?
5. Что означает свойство транзакции «согласованность»?
6. Что означает свойство транзакции «изолированность»?
7. Что означает свойство транзакции «долговечность»?
8. Какие виды определения транзакций поддерживает MS SQL Server?
9. Что такое явное определение транзакции?
10. Что такое автоматическое определение транзакции?
11. Что такое неявное определение транзакции?
12. Как явно начинается транзакция?
13. Какие команды завершают транзакцию?
14. Что такое точки сохранения в транзакции?
15. Какие способы обработки ошибок в транзакции вы знаете?
16. Что такое вложенные транзакции?

## Практические задания

1. Дана база данных КЛИЕНТЫ:

КЛИЕНТ (Код_клиента, Наименование, Годовой_доход, Тип_заказчика [Производитель, Оптовый_продавец, Торговая_компания])

ОТГРУЗКА (Номер_отгрузки, Код_клиента, Вес, Номер_грузовика, Город, Дата)

ВОДИТЕЛЬ (Номер_отгрузки, Имя_водителя)

ГОРОД (Название, Число_жителей)

а) Напишите транзакцию добавления новой записи в таблицу ОТГРУЗКА, учитывая, что информация о клиенте отсутствует в таблице КЛИЕНТ.

б) Напишите транзакцию, в которой при вводе новой записи в таблицу ОТГРУЗКА добавляется новая запись в таблицу ГОРОД.

2. Даны три таблицы:

ПРОДАВЕЦ (Код_продавца, Имя, Город, Комиссионные)

ЗАКАЗЧИК (Код_покупателя, ФИО, Рейтинг, Город, Код_продавца)

ПОКУПКА (Номер, Сумма, Дата, Код_продавца, Код_покупателя)

Напишите транзакцию добавления новой записи в таблицу ПОКУПКА, учитывая, что информация о продавце отсутствует в таблице ПРОДАВЕЦ.

2. Дана реляционная схема базы данных ГОСТИНИЦА:

ТИП НОМЕРА (Тип_номера, Количество_мест, Цена)

НОМЕР (Номер, Тип_номера)

КЛИЕНТ (Код_клиента, Город, ФИО)

КЛИЕНТ ПРОЖИВАЕТ (Код_клиента, Номер, Дата_прибытия, Дата_убытия)

Напишите транзакцию оформления нового клиента.

### *4.3.6. Транзакции и работа в многопользовательском режиме*

Данные в БД обычно используются совместно многими пользователями (приложениями). Ситуация, когда несколько пользователей одновременно выполняют операции чтения и записи одних и тех же данных, называется *параллельным (одновременным) доступом*. Таким образом, каждая система управления базами данных должна обладать каким-либо типом механизма управления для решения проблем, возникающих вследствие одновременного конкурентного доступа. И транзакции в силу своих свойств играют важную роль для обеспечения эффективной работы системы в многопользовательском режиме.

Действительно, если каждый сеанс взаимодействия с базой данных реализуется транзакцией, то пользователь начинает с того, что обращается к согласованному состоянию базы данных – состоянию, в котором она могла бы находиться, даже если бы пользователь работал с ней в одиночку.

***4.3.6.1. Проблемы работы в многопользовательском режиме.*** Если в СУБД параллельные транзакции не обрабатывать положенным образом, то при одновременном чтении и изменении одних и тех же данных несколькими пользователями могут возникнуть проблемы одновременного доступа. Хотя такие проблемы довольно многочисленны, их можно разбить на четыре основные категории:

1) потерянное обновление (lost update);
2) «грязное» чтение (dirty read);
3) неповторяющееся чтение (non-repeatable read);
4) фантомное чтение (phantom read).

Рассмотрим ситуации, в которых возможно возникновение данных проблем.

***Потерянное обновление* (lost update).** Проблема возникает, когда несколько пользователей изменяют одну и ту же строку, основываясь на ее начальном значении. Тогда часть данных будет потеряна, так как каждая последующая транзакция перезапишет изменения, сделанные предыдущей.

Предположим, имеется две транзакции, открытые разными пользователями, в которых выполнены следующие команды (рис. 4.30).

| Транзакция 1 | Транзакция 2 |
|---|---|
| SELECT f2 FROM tbl1 WHERE f1=1 | SELECT f2 FROM tbl1 WHERE f1=1 |
| UPDATE tbl1 SET f2=20 WHERE f1=1 | |
| | UPDATE tbl1 SET f2=25 WHERE f1=1 |

Рис. 4.30. Пример проблемы «потерянное обновление»

В транзакции 1 изменяется значение поля f2, а затем в транзакции 2 также изменяется значение этого поля. В результате изменение, выполненное первой транзакцией, будет потеряно.

Данный пример показывает, что проблема потерянного обновления может возникать всякий раз, когда два пользователя извлекают из базы одни и те же данные, используют их для каких-либо расчетов, а затем пытаются обновить эти данные.

***«Грязное» чтение (dirty read).*** Это чтение, при котором происходит считывание еще не фиксированных данных. Чтение нефиксированных данных возникает в том случае, когда одна транзакция модифицирует данные, а вторая транзакция читает эти модифицированные данные до того, как зафиксированы изменения, внесенные в первой транзакции. В случае отката первой транзакции вторая транзакция получит данные, которых нет в базе данных.

Предположим, имеется две транзакции, открытые разными пользователями, в которых выполнены следующие команды (рис. 4.31).

| Транзакция 1 | Транзакция 2 |
|---|---|
| SELECT f2 FROM tbl1 WHERE f1=1 | |
| UPDATE tbl1 SET f2=f2+1 WHERE f1=1 | |
| | SELECT f2 FROM tbl1 WHERE f1=1 |
| ROLLBACK TRAN | |

Рис. 4.31. Пример проблемы «грязное чтение»

В транзакции 1 изменяется значение поля f2, а затем в транзакции 2 выбирается значение поля f2. После этого происходит откат транзакции 1. В результате значение, полученное второй транзакцией, будет отличаться от значения, хранимого в базе данных.

***Неповторяющееся чтение (non-repeatable read).*** Неповторяемое чтение возникает в случае, когда чтение одной строки данных происходит более одного раза в течение одной транзакции, а между чтениями другая транзакция вносит изменения в эту строку. При повторном чтении в первой транзакции будут считываться другие данные, поэтому в пределах одной транзакции получаются неповторяемые результаты.

Предположим, имеются две транзакции, открытые разными пользователями, в которых выполнены следующие команды (рис. 4.32).

| **Транзакция 1** | **Транзакция 2** |
|---|---|
| | SELECT f2 FROM tbl1 WHERE f1=1 |
| UPDATE tbl1 SET f2=f2+1 WHERE f1=1 | |
| | SELECT f2 FROM tbl1 WHERE f1=1 |

Рис. 4.32. Пример проблемы «неповторяющееся чтение»

В транзакции 2 выбирается значение поля f2, затем в транзакции 1 изменяется значение поля f2. При повторной попытке выбора значения из поля f2 в транзакции 2 будет получен другой результат. Эта ситуация особенно неприемлема, когда данные считываются с целью их частичного изменения и обратной записи в базу данных.

***Фантомное чтение (phantom read).*** Фантомное чтение возникает в том случае, когда одна транзакция пытается прочитать строку, которой еще не существует в начале данной транзакции, но которая вставляется второй транзакцией, прежде чем закончится первая транзакция. Если первая транзакция снова выполнит поиск этой строки, то обнаружит ее неожиданное появление. Эта новая строка называется фантомной строкой.

Предположим, имеется две транзакции, открытые разными пользователями, в которых выполнены следующие команды (рис. 4.33).

| **Транзакция 1** | **Транзакция 2** |
|---|---|
| | SELECT SUM(f2) FROM tbl1 |
| INSERT INTO tbl1 (f1,f2) VALUES (15,20) | |
| | SELECT SUM(f2) FROM tbl1 |

Рис. 4.33. Пример проблемы фантомного чтения

В транзакции 2 выполняется команда SELECT, использующая все значения поля f2. Затем в транзакции 1 выполняется вставка новой строки, приводящая к тому, что повторное выполнение команды SELECT в транзакции 2 выдаст другой результат.

Как видно из приведенных примеров, при обновлении базы данных в многопользовательском режиме существует возможность нарушения ее целостности. Чтобы исключить такую возможность, в SQL и используется механизм транзакций. Помимо того, что реляционная СУБД обязана выполнять транзакции по принципу «либо все, либо ничего», она имеет и другое обязательство по отношению к транзакциям: в ходе выполнения транзакции пользователь видит полностью непротиворечивую базу данных. Он не должен видеть промежуточные результаты транзакций других пользователей, и даже завершение таких транзакций не должно отражаться на данных, которые пользователь видит в течение транзакции.

Приведенное выше правило можно сформулировать иначе, пользуясь концепцией параллельного выполнения транзакций.

*Когда две транзакции, A и B, выполняются параллельно, СУБД гарантирует, что результаты их выполнения будут точно такими же, как и в случае, если:*

*1) вначале выполняется транзакция A, а затем транзакция B;*
*или*

*2) вначале выполняется транзакция B, а затем транзакция A.*

Данная концепция называется *сериализацией* транзакций. Это означает, что каждый пользователь может работать с базой данных так, как если бы не было других пользователей, работающих параллельно. На практике в большой корпоративной базе данных параллельно могут выполняться десятки или даже сотни транзакций. Использование концепции сериализации позволяет полностью справиться с такой ситуацией. Этим гарантируется, что для выполнения N-го количества параллельных транзакций СУБД обеспечит те же результаты, как если бы транзакции выполнялись последовательно, одна за другой. Концепция не определяет, какая именно последовательность должна использоваться. Она только указывает, что результаты параллельных транзакций должны быть равны результатам некоторой последовательности тех же транзакций.

***4.3.6.2. Модели одновременного конкурентного доступа.*** Современные СУБД поддерживают две разные модели одновременного конкурентного доступа:

1) пессимистический одновременный конкурентный доступ;
2) оптимистический одновременный конкурентный доступ.

В первой модели для предотвращения одновременного доступа к данным применяются блокировки.

*Блокировкой* называется временное ограничение на выполнение некоторых операций обработки данных.

Блокировки не допускают, чтобы изменение данных одними пользователями влияло на данные других пользователей. После того как действие пользователя приводит к блокировке, другие пользователи не могут выполнять действия, приводящие к конфликту с блокировкой, до тех пор, пока инициатор ее не снимет. Иными словами, система баз данных, использующая такую модель, предполагает, что между двумя или большим количеством транзакций в любое время может возникнуть конфликт, и поэтому блокирует ресурсы (строку, таблицу), как только они потребуются в течение периода транзакции. Пессимистический одновременный конкурентный доступ в основном применяется в системах с большим количеством конфликтов данных, в которых затраты на защиту данных с помощью блокировок меньше затрат на откат транзакций в случае конфликтов параллелизма.

При оптимистическом одновременном конкурентном доступе блокировка данных не используется. Когда пользователю требуется обновить строку, СУБД должна определить, не изменялась ли эта строка другим пользователем после того, как она была открыта для чтения. Система проверяет, вносил ли другой пользователь в строку изменение после считывания. Для такой проверки СУБД сохраняет старые версии строк, и любой процесс при чтении данных использует ту версию строки, которая была активной, когда пользователь начал чтение. Поэтому процесс может модифицировать данные без каких-либо ограничений, поскольку все другие процессы, которые считывают эти же данные, используют свою собственную сохраненную версию. Конфликтная ситуация возможна только при попытке двух операций записи использовать одни и те же данные. В таком случае система выдает ошибку. Как правило, при получении сообщения об ошибке пользователь откатывает транзакцию и начинает ее заново.

Оптимистическая модель одновременного конкурентного доступа применяется в системах с небольшим количеством конфликтов данных (или характеризующихся низкой конкуренцией за данные), в которых затраты на периодический откат транзакции меньше затрат на блокировку данных при считывании.

Компонент Database Engine системы управления базой данных MS SQL Server реализует обе модели одновременного конкурентного доступа.

***4.3.6.3. Блокировка транзакций.*** Как было сказано выше, пессимистический одновременный конкурентный доступ основывается на использовании блокировок.

Каждая транзакция блокирует нужные ей данные, гарантируя этим, что никакая другая транзакция не может изменять те же самые данные. Когда другая транзакция запрашивает модификацию заблокированных данных, система или останавливает эту транзакцию, выдавая ошибку, или переводит транзакцию в состояние ожидания.

Каждая транзакция запрашивает блокировку разных типов ресурсов, например строк, таблиц, от которых эта транзакция зависит. Блокировка не дает другим транзакциям изменять ресурсы, чтобы избежать ошибок в транзакции, запросившей блокировку. Каждая транзакция освобождает свои блокировки, если больше не зависит от блокируемого ресурса.

Пользователю чаще всего не нужно предпринимать никаких действий по управлению блокировками. Всю работу по установке, снятию и разрешению конфликтов выполняет специальный компонент сервера, называемый *менеджером блокировок*. Он автоматически оценивает, какое количество данных необходимо блокировать, и устанавливает соответствующий тип блокировки. Это позволяет поддерживать равновесие между производительностью работы системы блокирования и возможностью пользователей получать доступ к данным.

Блокировка имеет несколько разных свойств:

- длительность;
- режим;
- гранулярность.

Длительность блокировки – это период времени, в течение которого ресурс удерживает определенную блокировку. Длительность

блокировки зависит прежде всего от режима блокировки и выбора уровня изоляции (об уровнях изоляции будет рассказано ниже).

Режимы блокировки определяют разные типы блокировок. Выбор определенного режима блокировки зависит от выполняемых действий и типа ресурса, который требуется заблокировать.

Действия, выполняемые пользователями при работе с данными, сводятся к операциям двух типов: их чтению и изменению. В операции по изменению включаются действия по добавлению, удалению и собственно изменению данных. MS SQL Server может блокировать разные типы ресурсов, в том числе ключи, строки, страницы, таблицы и БД. Строки находятся внутри страниц, которые представляют собой физические блоки информации, содержащие таблицы и индексные данные.

В MS SQL Server существует более 20 типов блокировок. Далее рассмотрим только те типы блокировок, на которые стоит обратить внимание в первую очередь.

Для блокировок ресурсов уровня строки и страницы в MS SQL Server применяются следующие три типа блокировок: разделяемая (shared, S), монопольная (exclusive, X), обновления (update, U).

***Разделяемая блокировка.*** Она блокирует страницу или строку при выполнении операции чтения. Разделяемой такая блокировка называется потому, что она дает доступ к чтению одного и того же ресурса одновременно из нескольких транзакций, т. е. несколько транзакций могут одновременно накладывать разделяемую блокировку на один и тот же ресурс. Важно то, что другие транзакции не могут изменять заблокированный таким образом ресурс, пока не закончено выполнение всех разделяемых блокировок.

***Монопольная блокировка.*** MS SQL Server устанавливает блокировку этого типа, когда происходит добавление, обновление или удаление строк, и удерживает ее до окончания транзакции. Если сервер установил монопольную блокировку на ресурс, то никакая другая транзакция не может прочитать или изменить заблокированные данные. Она предотвращает доступ к данным со стороны других транзакций. Важный момент: монопольные блокировки не снимаются до окончания транзакции, независимо от уровня изоляции. Чем дольше длится транзакция, тем дольше сохраняются монопольные блокировки.

Монопольную блокировку нельзя установить, если на ресурс уже установлена разделяемая или монопольная блокировка другой транзакцией, т. е. на ресурс может быть установлена только одна монопольная блокировка. К тому же если монопольная блокировка установлена, то другие транзакции не смогут выполнить какое-либо блокирование того же ресурса.

***Блокировка обновления.*** Это нечто среднее между разделяемой и монопольной блокировками. Блокировка обновления используется главным образом для поддержки команды UPDATE. В команде UPDATE данные считываются перед изменением. Следовательно, необходим такой тип блокировки, который не предотвращает считывания данных другими транзакциями во время считывания их данной транзакцией. Для этой операции чтения MS SQL Server использует блокировки обновления, которые совместимы с разделяемыми блокировками, но не совместимы с другими блокировками обновления. Кроме того, когда MS SQL Server приступает к изменению данных, ему приходится повышать тип блокировки до монопольной. Следовательно, другие транзакции могут считывать данные в то время, когда эти данные считываются для команды UPDATE, но другим командам UPDATE приходится ждать, пока не будет освобождена блокировка обновления.

Блокировка обновления используется не только для операций UPDATE, но и для удаления данных (DELETE) на этапе поиска удаляемых данных. В случае вставки новых строк (INSERT) в таблицу с кластеризованным индексом данный тип блокировки также применим на этапе поиска правильного положения в индексе. Когда положение найдено, блокировка обновления преобразуется в монопольную блокировку индекса и происходит вставка новой строки. Об индексировании таблиц речь пойдет в п. 4.3.8.

Блокировка обновления может быть установлена на ресурс только при отсутствии на нем другой блокировки обновления или монопольной блокировки. Однако этот тип блокировки можно использовать для объектов с установленной разделяемой блокировкой.

Взаимодействие между транзакциями, работающими с одними и теми же данными, называется *совместимостью блокировок*.

Совместимость блокировок определяет, могут ли несколько транзакций одновременно получить блокировку одного и того же ре-

сурса. Если ресурс уже блокирован другой транзакцией, новая блокировка может быть предоставлена только в том случае, если режим запрошенной блокировки совместим с режимом существующей. В противном случае транзакция, запросившая новую блокировку, ожидает освобождения ресурса, пока не истечет время ожидания существующей блокировки.

Таблица 4.4 показывает совместимость блокировок, сгенерированных на одном уровне изоляции.

Таблица 4.4

Совместимость разных типов блокировок

| Тип запрашиваемой блокировки / Тип установленной блокировки | Разделяемая | Обновления | Монопольная |
|---|---|---|---|
| Разделяемая | Да | Да | Нет |
| Обновления | Да | Нет | Нет |
| Монопольная | Нет | Нет | Нет |

Эта таблица интерпретируется следующим образом. Предположим, транзакция T1 имеет блокировку, указанную в заголовке соответствующей строки таблицы, а транзакция T2 запрашивает блокировку, указанную в соответствующем заголовке столбца таблицы. Значение «Да» в ячейке на пересечении строки и столбца означает, что транзакция T2 может иметь запрашиваемый тип блокировки и она выполняется, а значение «Нет», что транзакция не выполняется.

Взаимодействие транзакций на уровне блокировок можно подытожить так: данные, которые изменяются одной транзакцией, не могут изменяться или считываться другой, пока первая не будет завершена. Если данные считываются одной транзакцией, они не могут быть изменены другой. По крайней мере, так по умолчанию ведет себя MS SQL Server.

На уровне таблицы существует пять разных типов блокировок:

- разделяемая (shared, S);
- монопольная (exclusive, X);
- разделяемая с намерением (intent shared, IS);
- монопольная с намерением (intent exclusive, IX);

• разделяемая с монопольным намерением (shared with intent exclusive, SIX).

Разделяемые (S) и монопольные (X) типы блокировок для таблицы соответствуют одноименным блокировкам для строк и страниц.

Блокировки с намерением (intent) не представляют собой особый режим. Они служат для оптимизации работы алгоритма установки разделяемых и монопольных блокировок, описанных выше.

В их основе лежит простая идея. Чтобы заблокировать в определенном режиме низкоуровневый ресурс (уровня строки или страницы), транзакция должна получить блокировку с намерением того же режима, но на более высоком уровне – на уровне таблицы. Если такой блокировки нет, то можно избежать проверки наличия уже существующих блокировок на интересующих ресурсах (строках, страницах) и сразу их установить. Если блокировка есть, то можно принять приемлемое решение о возможности установки определенной блокировки низкого уровня. Например, если одна транзакция блокирует строку, а другая запрашивает несовместимую блокировку для страницы или целой таблицы, в которых находится эта строка, MS SQL Server легко обнаруживает конфликт, поскольку он знает о намеренной блокировке первой транзакции, установленной для страницы и таблицы.

Совместимость разных типов блокировок на уровне таблиц базы данных приведена в табл. 4.5. Эта таблица интерпретируется точно таким же образом, как и предыдущая таблица.

Таблица 4.5

Возможность совмещения разных типов блокировок

| Тип запрашиваемой блокировки / Тип установленной блокировки | S | X | IS | SIX | IX |
|---|---|---|---|---|---|
| S | Да | Нет | Да | Нет | Нет |
| X | Нет | Нет | Нет | Нет | Нет |
| IS | Да | Нет | Да | Да | Да |
| SIX | Нет | Нет | Да | Нет | Нет |
| IX | Нет | Нет | Да | Нет | Да |

***Гранулярность блокировки.*** Определяет, какой ресурс блокируется в одной попытке блокировки.

Применительно к MS SQL Server могут блокироваться такие ресурсы, как:

- строка;
- индексный ключ;
- страница;
- экстент (группа страниц);
- таблица;
- база данных.

Система выбирает требуемую гранулярность блокировки автоматически.

Строка – наименьший ресурс, который может быть заблокирован. Поддержка блокировки на уровне строки включает в себя и данные строк, и записи индексов. Блокировка на уровне строки означает, что блокируется только та строка, к которой осуществляет доступ транзакция. Следовательно, все другие строки, принадлежащие той же странице, свободны и могут использоваться другими транзакциями.

Если задать блокировку с намерением на уровне страницы, то другим транзакциям будет запрещено получать монопольную блокировку для таблицы, содержащей эту страницу.

Обычно чем больше гранулярность блокировки, тем больше сокращается возможность одновременного доступа к данным. Это означает, что блокировка на уровне строки максимизирует конкурентный доступ, потому что она блокирует только одну строку, оставляя все другие свободными. Однако накладные расходы системы увеличиваются, потому что каждая заблокированная строка требует одной блокировки.

Блокировка на уровне страницы (и блокировка на уровне таблицы) ограничивает доступность данных, но уменьшает накладные расходы системы.

Если установлено много блокировок с одной и той же гранулярностью в процессе выполнения транзакции, сервер автоматически преобразует эти блокировки в блокировку на уровне таблицы.

***4.3.6.4. Взаимоблокировки.*** Блокировки могут стать причиной бесконечного ожидания и тупиковых ситуаций.

Тупиковые ситуации (deadlocks) возникают при взаимных блокировках транзакций. Чаще всего тупиковую ситуацию описывают примерно следующим образом.

Транзакция 1 блокирует ресурс А.

Транзакция 2 блокирует ресурс Б.

Транзакция 1 пытается получить доступ к ресурсу Б.

Транзакция 2 пытается получить доступ к ресурсу А.

Рассмотрим простой пример взаимного блокирования. Пусть имеется две транзакции, открытые разными пользователями, в которых выполнены следующие команды (рис. 4.34).

| **Транзакция 1** | **Транзакция 2** |
|---|---|
| BEGIN TRAN | BEGIN TRAN |
| UPDATE Поставщик | UPDATE Склад |
| SET Регион = ‘Москва’ | SET Цена += 10.00 |
| WHERE Код_постав = 17 | WHERE Ном_ном = 125 |
| SELECT Наименование, Цена, | SELECT Название, Рейтинг |
| Количество | FROM Поставщик |
| FROM Склад | WHERE Код_постав = 17 |
| WHERE Ном_ном = 125 | COMMIT TRAN |
| COMMIT TRAN | |

Рис. 4.34. Пример взаимной блокировки транзакций

Транзакция 1 обновляет в таблице ПОСТАВЩИК строку с поставщиком под номером 17. При этом транзакция 1 не закрывается. Транзакция 2 в это время обновляет строку с товаром под номенклатурным номером 125 в таблице СКЛАД и тоже остается открытой.

На данном этапе транзакция 1 удерживает монопольную блокировку для строки в таблице ПОСТАВЩИК, а транзакция 2 делает то же самое в таблице СКЛАД. Оба запроса завершились удачно, и ни одна транзакция не была заблокирована.

Далее транзакция 1 запрашивает из таблицы СКЛАД строку с товаром под номенклатурным номером 125, после чего подтверждает транзакцию.

Чтобы выполнить операцию чтения, транзакция 1 должна получить разделяемую блокировку. Но так как этот ресурс уже монопольно заблокирован транзакцией 2, транзакция 1 останавливает свою работу. На данном этапе происходит блокирование (пока что не взаим-

ное). Хотя все еще остается вероятность того, что транзакция 2 завершится, освободит все блокировки и позволит транзакции 1 получить доступ к ресурсу.

Но транзакция 2 не завершается и пытается запросить из таблицы ПОСТАВЩИК строку с поставщиком под номером 17. После чего подтверждает транзакцию.

Чтобы выполнить операцию чтения, транзакция 2 должна получить разделяемую блокировку для строки с поставщиком под номером 17, которая находится в таблице ПОСТАВЩИК. Это действие конфликтует с монопольной блокировкой, удерживаемой транзакцией 1. Транзакции блокируют друг друга, т. е. происходит взаимное блокирование.

Но это простейший вариант взаимной блокировки, в реальности приходится сталкиваться с более сложными случаями.

Понятно, что если возникла взаимоблокировка, то обе транзакции будут заблокированы навсегда, т. е. ни одна из них не будет выполнена. Поэтому данная ситуация требует вмешательства извне.

В MS SQL Server реализован так называемый монитор взаимоблокировок (Deadlock Monitor), который в автоматическом режиме периодически (где-то каждые 5 секунд) осуществляет поиск взаимоблокировок, и если он находит их, то вмешивается в данный процесс с целью устранения сложившейся ситуации.

А решение здесь только одно: выбрать в качестве жертвы одну из транзакций и принудительно завершить ее. Таким образом, одна из транзакций будет выполнена успешно, а вторая завершится с известной ошибкой.

В MS SQL Server этот механизм настроен таким образом, что в качестве жертвы выбирается та транзакция, которую легче откатить. Например, в одной транзакции до появления взаимоблокировки уже были выполнены какие-то изменения данных, а в другой транзакции – нет, там было только чтение данных, то, конечно же, в качестве жертвы будет выбрана последняя транзакция, так как там, по сути, и откатывать нечего.

Взаимное блокирование – довольно ресурсоемкое явление, так как оно требует отката всей выполненной транзакцией работы и повторного ее выполнения. Полностью избежать взаимоблокировок нельзя. Однако можно минимизировать возможность возникновения

подобных ситуаций в своей системе, если использовать несколько советов по написанию кода транзакций.

- Очевидно, что чем дольше длится транзакция, тем дольше удерживается блокировка и тем выше вероятность взаимного блокирования. Следует делать транзакции как можно более короткими, вынося за их рамки те операции, которые с логической точки зрения не являются частью единого целого, например взаимодействия с пользователем в транзакциях.

- Взаимное блокирование происходит в том случае, когда транзакции обращаются к одним и тем же ресурсам в разном порядке. В вышеприведенном примере транзакция 1 сначала запрашивает таблицу ПОСТАВЩИК, а потом – таблицу СКЛАД, тогда как транзакция 2 делала все наоборот. Взаимоблокировки не произошло бы, если бы обе транзакции запрашивали таблицы в одном порядке. Таким образом, можно решить данную проблему, поменяв местами вышеупомянутые таблицы в одной из транзакций. Конечно, если это не повлияет на логику работы информационной системы.

- Используйте по возможности самый низкий уровень изоляции (об уровнях изоляции будет рассказано в п. 4.3.6.5), поскольку чем меньше уровень изоляции, тем меньше блокируются ресурсы и вероятность появления взаимоблокировок уменьшается.

- По возможности используйте уровень изоляции строк, основанный на управлении версиями строк. Например, если вы установите уровень изоляции READ COMMITTED SNAPSHOT (см. п. 4.3.6.7), потребность в разделяемой блокировке отпадет и, следовательно, исчезает еще одна потенциальная причина возникновения взаимного блокирования.

- Грамотная структура индексов может минимизировать вероятность возникновения взаимного блокирования, если в логике запросов нет настоящего логического конфликта. Действительно, в рассмотренном примере можно наблюдать настоящий логический конфликт, так как обе стороны пытаются получить доступ к одним и тем же строкам. Взаимное блокирование может случиться и без этого, например из-за отсутствия подходящих индексов для поддержки фильтров в запросе. Представьте, что команда во второй транзакции пытается получить товар под номером 100. Таким образом, транзакции обращаются к разным ресурсам, поэтому теоретически конфлик-

тов быть не должно. Если в таблицах не содержатся индексы для столбца Ном_ном, СУБД придется сканировать (и, следовательно, блокировать) все имеющиеся строки, чтобы выполнить фильтрацию. Естественно, это может привести к взаимному блокированию.

***4.3.6.5. Уровни изоляции транзакций.*** Теоретически все транзакции должны быть изолированы друг от друга. Но в таком случае доступность данных значительно понижается, поскольку операции одной транзакции блокировали бы операции в других транзакциях, и наоборот.

Уровень изоляции определяет поведение транзакций, которые считывают или записывают данные в один и тот же момент времени.

Уровни изоляции принято описывать с точки зрения того, какие проблемы одновременного конкурентного доступа (см. п. 4.3.6.1) они исключают.

Система управления базами данных MS SQL Server поддерживает пять уровней изоляции:

READ UNCOMMITTED – чтение незафиксированных данных;

READ COMMITTED – чтение фиксированных данных (используется по умолчанию в MS SQL Server);

REPEATABLE READ – повторяемость чтения;

SERIALIZABLE – сериализуемость;

SNAPSHOT.

Первые четыре уровня изоляции доступны только при пессимистической модели одновременного конкурентного доступа. Такая модель основана на блокировке данных. Уровень SNAPSHOT доступен при оптимистической модели одновременного конкурентного доступа, в котором блокировка данных не используется.

Уровень изоляции можно установить с помощью команды SET. Она распространяет уровень изоляции на всю сессию и выглядит следующим образом:

```
SET TRANSACTION ISOLATION LEVEL
  READ UNCOMMITTED | READ COMMITTED
 | REPEATABLE READ | SNAPSHOT
 | SERIALIZABLE
```

Каждый последующий уровень запрашивает более строгие блокировки при чтении и тем самым увеличивает время ожидания их снятия. Более высокий уровень изоляции повышает согласованность данных, но при этом может снижаться количество параллельно выполняемых транзакций.

Более низкий уровень изоляции позволяет выполнять больше параллельных транзакций, но снижает согласованность данных.

MS SQL Server при записи данных устанавливает монопольную блокировку и удерживает ее до конца транзакции вне зависимости от того, какой уровень изоляции используется. Блокировки обновления ведут себя аналогичным образом. Ключевая разница между пессимистическими уровнями изоляции транзакций состоит в том, как MS SQL Server работает с разделяемыми блокировками.

В табл. 4.6 показано, как уровни изоляции транзакций влияют на разделяемые блокировки.

Таблица 4.6

Поведение разделяемых блокировок

| Уровень изоляции | Поведение разделяемой блокировки |
|---|---|
| Read uncommitted | Блокировки не устанавливаются |
| Read committed | Блокировки устанавливаются и сразу снимаются после выполнения команды |
| Repeatable read | Блокировки устанавливаются и сохраняются до конца транзакции |
| Serializable | Устанавливаются блокировки на диапазон и сохраняются до конца транзакции |

Ниже описаны эти четыре уровня изоляции.

***Уровень изоляции READ UNCOMMITTED.*** Это самый низкий уровень изоляции, который не требует получения разделяемой блокировки для чтения данных, и, следовательно, исключается конфликт с транзакциями, которые производят запись в монопольном режиме. В таком случае транзакция читает данные, которые были обновлены какой-либо другой транзакцией. А если для другой транзакции позже выполняется откат, то это значит, что первая транзакция прочитала данные, которые никогда по-настоящему не существовали (данный процесс называется «грязным» чтением). Кроме того, пока операция

чтения происходит на уровне изоляции READ UNCOMMITTED, транзакции могут свободно записывать данные.

Чтобы увидеть, как происходит чтение неподтвержденных данных, рассмотрим две транзакции, открытые одновременно в разных сессиях работы СУБД (рис. 4.35). Все действия происходят в контексте спроектированной базы данных.

| **Сессия 1** | **Сессия 2** |
|---|---|
| Уровень изоляции принят по умолчанию (READ COMMITTED) | SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED |
| BEGIN TRAN | BEGIN TRAN |
| SELECT Ном_ном, Цена<br>FROM Склад<br>WHERE Ном_ном =4 | |
| UPDATE Склад<br>SET Цена +=10<br>WHERE Ном_ном=4 | |
| | SELECT Ном_ном, Цена<br>FROM Склад<br>WHERE Ном_ном =4 |
| ROLLBACK TRAN | COMMIT TRAN |

Рис. 4.35. Пример чтения неподтвержденных данных

Уровень изоляции сессии 1 принят по умолчанию (READ COMMITTED). Программный код транзакции сессии 1 открывает транзакцию, запрашивает строку с номенклатурным номером 4 (результат этого запроса приведен на рис. 4.36) и затем увеличивает цену товара с номенклатурным номером 4 на 10.

| Ном_ном | Цена |
|---|---|
| 4 | 200,00 |

Рис. 4.36. Результат выполнения команды SELECT транзакции сессии 1

Обращаем внимание, что в данный момент времени транзакция сессии 1 осталась открытой, а строка с измененной ценой монопольно заблокирована этой транзакцией.

Перейдем теперь к транзакции сессии 2. Для этой транзакции устанавливается режим изоляции READ UNCOMMITTED. Транзакция сессии 2 запрашивает строку с товаром под номенклатурным номером 4. Поскольку этот запрос не пытается получить разделяемую блокировку, он не конфликтует с первой транзакцией. Строка будет получена в том виде, который она приняла после обновления, несмотря на то, что изменения не были подтверждены (рис. 4.37).

| Ном_ном | Цена |
|---|---|
| 4 | 210,00 |

Рис. 4.37. Результат выполнения
команды SELECT транзакции сессии 2

Далее транзакция сессии 1 откатывает транзакцию и отменяет обновление товара под номенклатурным номером 4, возвращая прежнюю цену (200,00). Значение 210,00, полученное при чтении в транзакции 2, так и не было подтверждено. Это пример «грязного» чтения.

Из четырех проблем одновременного конкурентного доступа к данным, описанных в п. 4.3.6.1, уровень изоляции READ UNCOMMITTED допускает три: «грязное» чтение, неповторяющееся чтение и фантомы. Однако при таком уровне нет блокировки чтения, так что издержки минимальны.

***Уровень изоляции READ COMMITTED.*** Как видно из названия, этот уровень допускает чтение только тех изменений, которые были подтверждены. Система управления базами данных сохраняет монопольные блокировки строк, которые изменяются в транзакции, до конца транзакции, а разделяемые блокировки чтения снимаются сразу же после выполнения команды SELECT. Это уровень компонента в MS SQL Server задан по умолчанию.

Рассмотрим предыдущий пример, сделав два изменения (рис. 4.38).

1. Для сессии 2 установим уровень изоляции READ COMMITTED.

2. Первая транзакция в сессии 1 заканчивается командой COMMIT TRAN, т. е. фиксацией всех сделанных изменений данных.

Так же как и в предыдущем примере, транзакция сессии 1 установит монопольную блокировку для строки товара с номенклатурным номером 4.

Команда SELECT транзакции сессии 2 будет заблокирована, поскольку для выполнения чтения ей нужно получить разделяемую блокировку, которая конфликтует с монопольной блокировкой в транзакции 1.

Как только транзакция сессии 1 зафиксирует сделанные изменения, т. е. выполнит команду COMMIT TRAN, будет получен результат выполнения транзакции сессии 2, который приведен на рис. 4.37.

| **Сессия 1** | **Сессия 2** |
|---|---|
| Уровень изоляции принят по умолчанию (READ OMMITTED) | Уровень изоляции принят по умолчанию (READ COMMITTED) |
| BEGIN TRAN | BEGIN TRAN |
| SELECT Ном_ном, Цена<br>FROM Склад<br>WHERE Ном_ном =4 | |
| UPDATE Склад<br>SET Цена +=10<br>WHERE Ном_ном =4 | |
| | SELECT Ном_ном, Цена<br>FROM Склад<br>WHERE Ном_ном =4 |
| COMMIT TRAN | COMMIT TRAN |

Рис. 4.38. Пример чтения подтвержденных данных

В отличие от READ UNCOMMITTED, уровень изоляции READ COMMITTED исключает «грязное» чтение. При нем разделяемая блокировка снимается сразу, как только заканчивается чтение ресурса, не дожидаясь закрытия транзакции. Это означает, что если одна транзакция производит две операции чтения одного и того же ресурса, ей не нужно ничего блокировать. Следовательно, ресурс может быть изменен между данными операциями, в результате чего получится два разных значения. Это так называемое неповторяющееся чтение. Для некоторых информационных систем подобное явление вполне приемлемо.

***Уровень изоляции REPEATABLE READ.*** Это уровень, при котором чтение одной и той же строки или строк в транзакции дает одинаковый результат. На этом уровне изоляции для чтения необходимо будет получить разделяемую блокировку, которая удерживается до завершения транзакции. Это сделает невозможным монопольное блокирование при записи и, как следствие, изменение ресурса. Тем самым обеспечивается повторяющееся чтение.

В следующем примере демонстрируется работа в режиме REPEATABLE READ (рис. 4.39).

| **Сессия 1** | **Сессия 2** |
|---|---|
| Уровень изоляции принят по умолчанию (READ COMMITTED) | SET TRANSACTION ISOLATION LEVEL REPEATABLE READ |
| BEGIN TRAN | BEGIN TRAN |
| | SELECT Ном_ном, Цена<br>FROM Склад<br>WHERE Ном_ном =4 |
| UPDATE Склад SET Цена +=10<br>WHERE Ном_ном =4 | |
| | SELECT Ном_ном, Цена<br>FROM Склад<br>WHERE Ном_ном =4 |
| COMMIT TRAN | COMMIT TRAN |

Рис. 4.39. Пример повторяющегося чтения данных

Результат выполнения первой команды SELECT в транзакции сессии 2 приведен на рис. 4.40.

| Ном_ном | Цена |
|---|---|
| 4 | 200,00 |

Рис. 4.40. Результат выполнения первой команды SELECT транзакции сессии 2

Поскольку транзакция сессии 2 работает в режиме REPEATABLE READ, она будет удерживать разделяемую блокировку для строки с номенклатурным номером 4 до окончания транзакции.

Команде UPDATE в транзакции сессии 1 не удастся в данный момент времени изменить эту строку, поскольку монопольное блокирование, которое запрашивает эта транзакция для записи, конфликтует с разделяемой блокировкой, которая была выдана для чтения. Если бы работа велась на уровнях изоляции READ UNCOMMITTED или READ COMMITTED, разделяемой блокировки не было бы или она выделялась бы только на время выполнения команды SELECT, попытка записи оказалась бы успешной.

Повторное считывание строки с номенклатурным номером 4 в транзакции сессии 2 дает ту же цену, как и при первом считывании (см. рис. 4.40).

После завершения транзакции сессии 2 разделяемая блокировка будет снята и транзакция сессии 1 может монопольно заблокировать и обновить строку.

Уровень изоляции REPEATABLE READ приводит к тому, что разделяемая блокировка, полученная для чтения данных, удерживается до конца транзакции. Таким образом, на протяжении всей транзакции вы можете считывать одни и те же значения. При этом строки, которых не было при запуске запроса, не блокируются. Следовательно, они могут появиться при повторном чтении. Такие строки называются фантомными, а операции, которые их возвращают, – фантомным чтением.

Подобная ситуация возникает, если между двумя обращениями к таблице происходит добавление новых строк в рамках другой транзакции; при этом добавленные строки попадают в результирующий набор одного из запросов первой транзакции.

Чтобы избежать фантомного чтения, необходимо перейти на уровень изоляции SERIALIZABLE.

***Уровень изоляции SERIALIZABLE.*** Это самый высокий уровень изоляции: транзакции полностью изолируются друг от друга. На этом уровне результаты параллельного выполнения транзакций для базы данных совпадают с результатом последовательного выполнения тех же транзакций в любом порядке. Поэтому если транзакция обращается к данным, то никакая другая транзакция не сможет добавить новые или удалить имеющиеся строки, которые могут быть считаны при выполнении транзакции.

Этот режим во многом похож на REPEATABLE READ. В частности, для выполнения операций чтения он требует установления разделяемой блокировки, которая удерживается до конца транзакции. Но есть и отличие: при чтении данных вместо разделяемых блокировок строк используются разделяемые блокировки диапазона ключей, соответствующих фильтру запроса. Это означает, что чтение блокирует все строки, с которыми будет происходить работа, как уже имеющиеся, так и те, что могут быть добавлены в будущем. Точнее говоря, данный режим не дает другим транзакциям добавлять строки, которые могут пройти фильтр запроса.

Рассмотрим в качестве примера параллельную работу двух транзакций (рис. 4.41), в которых отсутствует фантомное чтение.

| **Сессия 1** | **Сессия 2** |
|---|---|
| Уровень изоляции принят по умолчанию (READ COMMITTED)<br>BEGIN TRAN | SET TRANSACTION ISOLATION LEVEL SERIALIZABLE<br>BEGIN TRAN<br>SELECT Ном_ном, Наименование<br>FROM Склад<br>WHERE Ном_ном BETWEEN 1 AND 3 |
| INSERT INTO Склад(Ном_ном, Наименование, Цена, Количество)<br>VALUES (2, 'Монитор ASUS 28', 15000.00, 1) | |
| | SELECT Ном_ном, Наименование<br>FROM Склад<br>WHERE Ном_ном BETWEEN 1 AND 3 |
| COMMIT TRAN | COMMIT TRAN |

Рис. 4.41. Пример решения проблемы фантомного чтения

В транзакции сессии 2 из таблицы СКЛАД выбираются товары со значением номенклатурного номера (Ном_ном) между 1 и 3. Результат этого запроса представлен на рис. 4.42.

| Ном_ном | Наименование |
|---|---|
| 1 | Принтер XEROX |
| 3 | Процессор Intel |

Рис. 4.42. Результат выполнения первой команды SELECT транзакции сессии 2

Из рис. 4.42 следует, что в базе данных отсутствует товар с номенклатурным номером 2.

Далее первая транзакция пытается добавить новый товар с номенклатурным номером, равным 2.

На первых трех уровнях изоляции эта попытка была бы успешной, но в режиме SERIALIZABLE данная команда INSERT приведет к блокированию, поскольку транзакция сессии 2 для первой команды SELECT запросила разделяемую блокировку на диапазон ключей до конца транзакции. И эта блокировка будет запрещать как изменять строки, которые попадают в этот диапазон, так и вставлять новые.

Выполнив вторую команду SELECT транзакции сессии 2 получаем тот же результат, что и прежде, но без фантомного чтения (см. рис. 4.42).

Теперь, когда вторая транзакция подтверждена и разделяемая блокировка диапазона ключей снята, транзакция сессии 1 может установить монопольную блокировку, разрешение на которую она ожидала все это время, и начать добавление строки.

***4.3.6.6. Переопределение блокировок на уровне запросов.*** Выше приведена команда SET, с помощью которой можно установить уровень изоляции. Но есть другой способ задать тип блокировки – использование табличных подсказок.

Если команда SET распространяет уровень изоляции на всю сессию, то табличные подсказки ограничивают уровень изоляции только текущим запросом и вставляются в предложение FROM команды SELECT:

SELECT ... FROM <Имя таблицы> [WITH (<подсказка >)]

Подсказки блокировки используют для переопределения текущего значения уровня изоляции, установленного командой SET.

Использование табличных подсказок позволяет в одном запросе, объединяющем несколько таблиц, указывать свой уровень изоляции для каждой таблицы.

Использование подсказок блокировки не влияет на уровень изоляции для других транзакций.

Рассмотрим ситуацию, в которой может оказаться полезным использование подсказок блокировки. Предположим, что в системе для всех транзакций используется принятый по умолчанию уровень изоляции READ COMMITTED. Как известно, при этом уровне изоляции по данному ресурсу захватывается разделяемая блокировка только до

завершения чтения, и затем эта разделяемая блокировка освобождается. И если какая-либо транзакция читает одни и те же данные дважды, то результаты чтения могут не совпасть, поскольку другая транзакция могла захватить блокировку между первым и вторым чтениями и модифицировать эти данные.

Чтобы избежать проблемы повторяемости чтения, можно задать уровень изоляции REPEATABLE READ, но это вынудит MS SQL Server захватить все разделяемые блокировки, необходимые для команд SELECT во всех транзакциях, пока не будет завершена каждая транзакция. Иными словами, для целостности транзакций разделяемые блокировки будут захвачены по таблице, указанной в команде SELECT любой транзакции. Если не требуется поддерживать повторяющееся чтение для всех транзакций, то можно добавить какую-либо подсказку блокировки для определенного запроса. Например, подсказка блокировки REPEATABLEREAD в команде SELECT указывает MS SQL Server захват всех разделяемых блокировок по таблице, заданной в команде SELECT, вплоть до конца этой транзакции, независимо от уровня изоляции. Тем самым при повторном чтении будет наблюдаться согласованность данных (они не будут изменены другой транзакцией).

Далее приведем описание существующих подсказок блокировки на уровне таблиц.

HOLDLOCK – указывает, что накладывается разделяемая блокировка на ресурс до завершения транзакции. Равнозначна подсказке SERIALIZABLE.

NOLOCK – указывает, что не накладываются разделяемые блокировки, а монопольные блокировки других транзакций не мешают текущей транзакции считывать заблокированные данные. Эта подсказка позволяет читать незафиксированные данные. Равнозначна подсказке READUNCOMMITTED.

PAGLOCK – указывает, что используется блокировка страницы вместо блокировки отдельной таблицы. По умолчанию используется уровень блокировки, соответствующий операции (чтение или запись). При указании блокировок в транзакциях, выполняемых с уровнем изоляции SNAPSHOT, они применяются только в том случае, когда подсказка PAGLOCK используется в сочетании с другими табличными подсказками, требующими блокировки, например UPDLOCK или HOLDLOCK.

READCOMMITTED – указывает, что операции чтения выполняются по правилам для уровня изоляции READ COMMITTED путем использования блокировки или управления версиями строк.

READPAST – указывает, что СУБД не должно считывать строки и страницы, заблокированные другими транзакциями. Если указана подсказка READPAST, СУБД будет пропускать строки вместо блокировки текущей транзакции до тех пор, пока блокировки не будут сняты. Например, предположим, что в таблице Tabl1 есть один целочисленный столбец со значениями 1, 2, 3, 4, 5. Если транзакция А изменит значение 3 на 8, но еще не будет зафиксирована, то команда SELECT * FROM Tabl1 (READPAST) возвратит значения 1, 2, 4, 5. Параметр READPAST используется главным образом для устранения конфликта блокировок при реализации рабочей очереди, использующей таблицу MS SQL Server. Средство чтения очереди, использующее аргумент READPAST, пропускает прошлые записи очереди, заблокированные другими транзакциями, до следующей доступной записи очереди, не дожидаясь, пока другие транзакции снимут свои блокировки.

REPEATABLEREAD – указывает, что чтение выполняется по тем же правилам блокировки, что и у транзакций, использующих уровень изоляции REPEATABLE READ.

ROWLOCK – указывает, что используются блокировки на уровне строк вместо блокировки страниц или таблиц. При указании блокировок строк в транзакциях, выполняемых на уровне изоляции SNAPSHOT, они применяются только в случае, когда подсказка ROWLOCK используется в сочетании с другими табличными подсказками, требующими блокировки, например UPDLOCK или HOLDLOCK.

SNAPSHOT – указывает, что доступ к таблице выполняется с уровнем изоляции SNAPSHOT. Об уровне изоляции SNAPSHOT говорится в п. 4.3.6.7.

TABLOCK – указывает, что полученная блокировка применяется на уровне таблицы. При задании параметра TABLOCK на все время выполнения команды блокируется вся таблица, указанная в команде. MS SQL Server захватывает эту блокировку до завершения команды. Если также указана подсказка HOLDLOCK, то блокировка таблицы удерживается до конца транзакции.

TABLOCKX – указывает, что к таблице применяется монопольная блокировка. Эта подсказка препятствует доступу других транзакций к этой таблице на все время действия транзакции, запрещая всем остальным пользователям доступ к таблице до момента окончания транзакции.

UPDLOCK – указывает, что блокировки обновления применяются и удерживаются до завершения транзакции. UPDLOCK получает блокировки обновления для операций чтения только на уровне строк или страниц. Этот параметр блокирует изменение значений таблиц, но дает возможность другим пользователям считывать информацию. Это состояние помогает предотвратить задержки, мешающие пользователям просматривать данные таблиц в то время, когда их значения обновляются. При задании этого параметра пользователи могут просматривать содержимое базы данных, но не могут обновлять заблокированные строки. Если UPDLOCK используется в сочетании с TABLOCK или по какой-либо другой причине уже получена блокировка на уровне таблицы, то вместо них будет получена монопольная блокировка.

Например, использование подсказки UPDLOCK может быть полезно для обеспечения последовательного чтения некоторых данных, когда требуется выбрать некоторое значение из таблицы, выполнить обработку и обновить строку. В этом случае использование блокировки обновления или монопольной блокировки запретит чтение данных до тех пор, пока сессия не закончит транзакцию.

Транзакция, представленная на рис. 4.43, служит примером использования подсказки UPDLOCK в команде SELECT для изменения типа установленной в сессии блокировки.

**Сессия**

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRAN
  SELECT Ном_ном, Цена
  FROM Склад WITH (UPDLOCK)
  WHERE Ном_ном =10010
COMMIT TRAN
```

Рис. 4.43. Пример использования подсказки UPDLOCK

Приведем еще пример (рис. 4.44), в котором в одном запросе, соединяющем несколько таблиц, для каждой таблицы указывается свой уровень изоляции.

```
BEGIN TRAN
  SELECT p. Ном_ном, SUM(p. Количество) AS Количество,
      SUM(p. Количество * p. Цена) AS Сумма
  FROM Приход p WITH (READCOMMITTED) INNER JOIN Документ d
      WITH (SERIALIZABLE)
  ON p. Ном_док = d. Ном_док
  WHERE d. Дата BETWEEN ’02.03.2021’ AND ’24.04.2021’
  GROUP BY p. Ном_ном
  SELECT TOP 10 Ном_док, Дата
  FROM Документ
  ORDER BY Дата DESC
COMMIT TRAN
```

Рис. 4.44. Пример запроса с несколькими подсказками уровня изоляции

В первом запросе транзакции используется соединение наборов строк из двух таблиц – ПРИХОД и ДОКУМЕНТ. Для набора строк таблицы ПРИХОД используется уровень изоляции READ COMMITED, а для набора строк таблицы ДОКУМЕНТ – уровень изоляции SERIALIZABLE. Для набора строк таблицы ПРИХОД разделяемые блокировки снимутся сразу после чтения строк, но для таблицы ДОКУМЕНТ они останутся до конца транзакции. Другие сессии смогут изменять строки для таблицы ПРИХОД, но не смогут изменить данные таблицы ДОКУМЕНТ, которые были прочитаны при выполнении запроса.

Можно объединять совместимые подсказки блокировки, такие как TABLOCK и REPEATABLEREAD, но нельзя объединять конфликтующие подсказки, такие как REPEATABLEREAD и SERIALIZABLE.

*Примечание.* Оптимизатор запросов MS SQL Server автоматически определяет наиболее эффективный план исполнения и типы блокировок для запроса. И поскольку оптимизатор запросов автоматически выбирает подходящий тип (или типы) блокировок, подсказки

блокировки следует использовать, если только они хорошо изучены и когда это абсолютно необходимо, иначе они могут отрицательным образом повлиять на параллельно выполняемые задачи.

***4.3.6.7. Уровни изоляции, основанные на управлении версиями строк.*** В MS SQL Server поддерживаются еще два уровня изоляции – SNAPSHOT и READ COMMITTED SNAPSHOT, которые используются при оптимистической модели одновременного конкурентного доступа к данным.

SNAPSHOT – это полноценный уровень изоляции, в то время как READ COMMITTED SNAPSHOT – это параметр базы данных, который влияет на поведение операций чтения с уровнем изоляции READ COMMITTED.

Эти уровни – аналоги уровней SERIALIZABLE и READ COMMITTED соответственно. В их основе лежит механизм, который позволяет хранить предыдущие версии подтвержденных строк в БД tempdb. Эти уровни изоляции при чтении данных не запрашивают разделяемые блокировки. Теперь, когда операция чтения обратится к строке, для которой установлена монопольная блокировка, конфликта блокировки не произойдет, а будет считана старая версия строки из базы данных tempdb.

Нужно отметить, что при включении одного из этих двух уровней изоляции команды DELETE и UPDATE перед внесением изменений будут копировать содержимое строк в БД tempdb. Это не относится к команде INSERT, поскольку у строки, которая добавляется, не может быть предыдущих версий.

Нетрудно догадаться, что уровни изоляции, основанные на управлении версиями строк, позволяют увеличить производительность чтения данных, поскольку уменьшают влияние блокировок. Однако эти уровни изоляции имеют несколько неприятных нюансов. Наиболее значительные из них следующие.

1. Создается дополнительная нагрузка на базу данных tempdb. Применение этих уровней изоляции в системах с большим количеством изменений данных может привести к интенсивному использованию базы tempdb и значительному увеличению ее в объеме. Если база данных tempdb заполнена, операции обновления прекращают формирование версий и продолжаются до успешного выполнения, а операции считывания завершаются ошибкой, поскольку конкретной требуемой версии строки уже не существует.

2. Появляются дополнительные накладные расходы при изменении данных и их извлечении. MS SQL Server приходится копировать данные в базу tempdb и поддерживать связанный список версий записей, а при чтении данных нужно этот список обойти. Это добавляет дополнительную нагрузку на процессор и систему ввода-вывода.

Далее рассмотрим теоретические и практические стороны уровней изоляции, в основе которых лежит управление версиями строк.

***Уровень изоляции SNAPSHOT.*** SNAPSHOT – отдельный уровень изоляции. Разрешение уровня изоляции SNAPSHOT осуществляется в два шага. Так как по умолчанию использование уровня изоляции SNAPSHOT запрещено, то сначала необходимо активировать параметр ALLOW_SNAPSHOT_ISOLATION в контексте всей БД с помощью команды

```
ALTER DATABASE <Имя базы данных> SET ALLOW_SNAPSHOT_ISOLATION ON
```

После этого для каждого сеанса, который будет использовать этот уровень изоляции, он должен быть явно задан с помощью команды

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
```

или с помощью табличной подсказки WITH (SNAPSHOT).

Уровень изоляции SNAPSHOT гарантирует, что клиент, считывающий данные, получит последнюю версию строки, подтвержденную на момент начала транзакции. Это обеспечивает повторяющееся и исключает фантомное чтение (точно так же, как в режиме SERIALIZABLE). При этом вместо разделяемых блокировок используется разбиение строк на версии.

Транзакция уровня изоляции SNAPSHOT при изменении данных проверяет, не была ли строка данных изменена и подтверждены ли изменения данных этой строки в конкурирующей транзакции после того, как была запущена текущая транзакция. Если строка данных была изменена параллельной конкурирующей транзакцией, то возникает конфликт обновления и дальнейшее выполнение транзакции уровня изоляции SNAPSHOT завершается. Этот конфликт обновления обрабатывается СУБД.

Чтобы продемонстрировать поведение режима SNAPSHOT, рассмотрим конкретный пример выполнения двух транзакций, приведенных на рис. 4.45.

| Сессия 1 | Сессия 2 |
|---|---|
| Уровень изоляции принят по умолчанию (READ COMMITTED) | SET TRANSACTION ISOLATION LEVEL SNAPSHOT |
| BEGIN TRAN | BEGIN TRAN |
| UPDATE Склад<br>SET Цена +=10<br>WHERE Ном_ном =4 | |
| SELECT Ном_ном, Цена<br>FROM Склад<br>WHERE Ном_ном =4 | |
| | SELECT Ном_ном, Цена<br>FROM Склад<br>WHERE Ном_ном =4 |
| COMMIT TRAN | |
| | COMMIT TRAN |
| | BEGIN TRAN |
| | SELECT Ном_ном, Цена<br>FROM Склад<br>WHERE Ном_ном =4 |
| | COMMIT TRAN |

Рис. 4.45. Пример поведения режима SNAPSHOT

Транзакция сессии 1 увеличивает цену товара на складе с номенклатурным номером 4 на 10 и делает запрос к измененной строке с новой ценой. Если старая цена товара с номенклатурным номером 4 была равна 200, то результат этого запроса будет выглядеть следующим образом (рис. 4.46).

| Ном_ном | Цена |
|---|---|
| 4 | 210,00 |

Рис. 4.46. Результат выполнения
команды SELECT транзакции сессии 1

Стоит отметить, что перед изменением строки ее текущее состояние (с ценой 200,00) будет скопировано в БД tempdb.

Далее транзакция сессии 2 запрашивает строку с номенклатурным номером товара, равным 4. Если бы сессия 2 работала на уровне изоляции SERIALIZABLE, запрос был бы заблокирован. В нашем

случае установлен режим SNAPSHOT, поэтому получим последнюю версию строки, которая была доступна на момент запуска транзакции. Результат этого запроса представлен на рис. 4.47.

| Ном_ном | Цена |
|---|---|
| 4 | 200,00 |

Рис. 4.47. Результат выполнения
команды SELECT транзакции сессии 2

После фиксации изменений в транзакции сессии 1 (команда COMMIT TRAN) текущая версия строки (с ценой 210,00) уже подтверждена. Попытавшись прочитать данные в транзакции сессии 2, получим ту же версию строки, которая была актуальна на момент начала транзакции (с ценой 200,00).

Откроем в сессии 2 новую транзакцию, код которой открывает транзакцию, извлекает данные (команда SELECT) и фиксирует новую транзакцию.

На этот раз последняя подтвержденная версия строки (с ценой 210,00) доступна на момент запуска транзакции. Результат выполнения команды SELECT приведен на рис. 4.48.

| Ном_ном | Цена |
|---|---|
| 4 | 210,00 |

Рис. 4.48. Результат выполнения
команды SELECT в новой транзакции

Версия с предыдущей ценой потеряла актуальность. Она будет удалена специальным потоком, который ежеминутно проводит очистку БД tempdb.

В режиме SNAPSHOT при изменении данных MS SQL Server определяет, не была ли строка данных изменена ранее в конкурирующей транзакции. Такой конфликт обнаруживается посредством анализа БД tempdb. Уровень изоляции SNAPSHOT предотвращает некорректное обновление данных, но делает это не за счет блокировок, а путем прерывания транзакции, сообщая о наличии конфликта.

В рассмотренном выше примере только в одной транзакции изменялись данные. Поэтому никаких конфликтов между транзакциями не возникало. Ниже приводится пример, в котором возникает конфликт между двумя транзакциями при обновлении данных (рис. 4.49).

| **Сессия 1** | **Сессия 2** |
|---|---|
| SET TRANSACTION ISOLATION LEVEL SNAPSHOT | SET TRANSACTION ISOLATION LEVEL SNAPSHOT |
| BEGIN TRAN | BEGIN TRAN |
| SELECT Ном_ном, Цена<br>FROM Склад<br>WHERE Ном_ном =4 | |
| | UPDATE Склад<br>SET Цена =250<br>WHERE Ном_ном =4 |
| UPDATE Склад<br>SET Цена = 300<br>WHERE Ном_ном =4 | |
| COMMIT TRAN | COMMIT TRAN |

Рис. 4.49. Пример конфликта при обновлении данных

Сессия 1 содержит код, который устанавливает уровень изоляции SNAPSHOT, открывает транзакцию, считывает строку с товаром под номенклатурным номером 4 (предположим, что значение поля Цена равно 200) и пытается для товара с номенклатурным номером 4 установить цену, равную 300.

Одновременно с сессией 1 запускается сессия 2, в которой транзакция устанавливает для того же товара цену, равную 250. Причем изменение цены во второй сессии происходит раньше, чем в первой сессии. MS SQL Server увидит, что между операциями чтения и записи сработала другая транзакция, которая изменила данные. Следовательно, транзакция, инициированная сессией 1, завершится ошибкой и выдачей следующего сообщения:

*Сообщение 3960, уровень 16, состояние 2, строка 1*

*Транзакция в режиме изоляции моментального снимка прервана из-за конфликта обновлений. Невозможно использовать режим изоляции моментального снимка для прямого или косвенного доступа к таблице «Склад» в базе данных «Контроль_товара» для обновления, удаления или вставки строки, которая изменена или удалена другой транзакцией. Повторите транзакцию или измените уровень изоляции для инструкции обновления или удаления.*

При обнаружении конфликта можно обработать код ошибки и повторить всю транзакцию заново.

Таким образом, уровень изоляции SNAPSHOT предоставляет аналогичную согласованность данных, что и уровень SERIALIZABLE, но без установки блокировок, хотя может генерировать огромное количество данных в базе tempdb.

***Уровень изоляции READ COMMITTED SNAPSHOT.*** Включается командой

ALTER DATABASE <Имя базы данных> SET READ_COMMITTED_SNAPSHOT ON

Обращаем внимание, что любая команда, для которой указан уровень изоляции READ COMMITTED, теперь будет выполняться на уровне READ COMMITTED SNAPSHOT.

Уровень изоляции READ COMMITTED SNAPSHOT тоже основан на управлении версиями строк. В отличие от SNAPSHOT, он предоставляет последний снимок данных, подтвержденный на момент запуска команды, а не транзакции. Кроме того, он не следит за возникновением конфликтов при операциях обновления. По принципу своей работы этот уровень похож на уровень READ COMMITTED, если не считать того факта, что при нем чтение может производиться без получения разделяемых блокировок и без ожидания, связанного с монопольным блокированием.

Основное преимущество уровня изоляции READ COMMITTED SNAPSHOT состоит в том, что операции чтения не блокируют обновлений, а обновления не блокируют операций чтения. Однако обновления блокируют другие обновления, поскольку для выполнения операций обновления устанавливаются монопольные блокировки.

Рассмотрим пример, в котором две сессии используют уровень изоляции READ COMMITTED SNAPSHOT (рис. 4.50).

| **Сессия 1** | **Сессия 2** |
|---|---|
| BEGIN TRAN | BEGIN TRAN |
| UPDATE Склад<br>SET Цена += 10<br>WHERE Ном_ном =4 | |
| SELECT Ном_ном, Цена<br>FROM Склад<br>WHERE Ном_ном =4 | |
| | SELECT Ном_ном, Цена<br>FROM Склад<br>WHERE Ном_ном =4 |
| COMMIT TRAN | |
| | SELECT Ном_ном, Цена<br>FROM Склад<br>WHERE Ном_ном =4 |
| | COMMIT TRAN |

Рис. 4.50. Пример действий на уровне изоляции READ COMMITTED SNAPSHOT

Транзакция сессии 1 увеличивает цену товара с номенклатурным номером 4 на 10, а затем запрашивает измененную строку. Результат такого запроса, представленный на рис. 4.51, говорит о том, что цена была повышена до 210,00.

| Ном_ном | Цена |
|---|---|
| 4 | 210,00 |

Рис. 4.51. Результат выполнения команды SELECT в сессии 1

Далее транзакция во второй сессии читает строку с товаром под номенклатурным номером 4. В результате получаем последнюю подтвержденную версию строки с ценой, равной 200, которая была доступна на момент запуска команды.

После этого фиксируем транзакцию в сессии 1 и повторно читаем строку с товаром под номенклатурным номером 4.

Если транзакция работала в режиме SNAPSHOT, была бы получена цена 200,00, но поскольку определен уровень изоляции READ COMMITTED SNAPSHOT, вернулась последняя подтвержденная версия строки, доступная на момент запуска команды, а не транзакции (рис. 4.52).

| Ном_ном | Цена |
|---|---|
| 4 | 210,00 |

Рис. 4.52. Результат повторного чтения строки в сессии 2

Как мы уже знаем, это явление называется неповторяющимся чтением.

Понимая смысл проблем работы в многопользовательском режиме (см. п. 4.3.6.1), очень просто разобраться в назначении каждого уровня изоляции, так как они различаются между собой количеством проблем работы в многопользовательском режиме. В табл. 4.7 приведено поведение СУБД MS SQL Server при различных уровнях изоляции транзакций.

Таблица 4.7

Поведение MS SQL Server при различных уровнях изоляции

| Уровень изоляции | Проблема | | | |
|---|---|---|---|---|
| | Потерянное обновление | «Грязное» чтение | Неповторяющееся чтение | Фантомное чтение |
| Read uncommitted | Может произойти | Может произойти | Может произойти | Может произойти |
| Read Committed или Read committed snapshot | СУБД предотвращает | СУБД предотвращает | Может произойти | Может произойти |
| Repeatable read | СУБД предотвращает | СУБД предотвращает | СУБД предотвращает | Может произойти |
| Serializable или Snapshot | СУБД предотвращает | СУБД предотвращает | СУБД предотвращает | СУБД предотвращает |

## Вопросы для самопроверки

1. Какие модели независимой параллельной работы нескольких транзакций вы знаете?

2. Какова особенность пессимистической модели одновременного конкурентного доступа?

3. Какова особенность оптимистической модели одновременного конкурентного доступа?

4. Что такое блокировка?

5. Что определяет режим блокировки?

6. Что определяет гранулярность блокировки?

7. Что такое монопольная блокировка?

8. Что такое разделяемая блокировка?

9. В каком случае запрашивается монопольная блокировка?

10. В каком случае запрашивается разделяемая блокировка?

11. Какие типы блокировок совместимы с монопольной блокировкой?

12. Как взаимодействуют несколько транзакций, запрашивая одновременно различные режимы блокировки?

13. Что такое блокировка с намерением?

14. В чем состоит разница между основными типами блокировки (разделяемой и монопольной) и блокировкой с намерением?

15. Какая иерархия ресурсов используется при определении гранулярности блокировки?

16. Что такое повышение гранулярности блокировки?

17. Для чего применяются различные уровни изоляции?

18. Какие уровни изоляции поддерживает MS SQL Server?

19. Что такое запрещение «грязного» чтения?

20. Что такое запрещение неповторяющегося чтения?

21. Что такое запрещение фантомов?

22. Какая модель одновременного конкурентного доступа применяется с уровнем изоляции SNAPSHOT?

23. Как задается уровень изоляции?

24. Что такое взаимоблокировка?

### *4.3.7. Журнал транзакций*

Выше были рассмотрены два аспекта баз данных, в которых транзакции играют важную роль, – это сохранение целостности данных и обеспечение параллельной работы пользователей с базой данных. Выясним, какое значение имеют транзакции для восстановления данных при откатах и сбоях.

Реляционные СУБД позволяют фиксировать или откатывать транзакции. Очевидно, что для этого необходима некоторая дополни-

тельная информация. В подавляющем большинстве современных реляционных СУБД такая избыточная дополнительная информация поддерживается в виде журнала изменений базы данных, чаще всего называемого *журналом транзакций (transaction log).*. Для каждой базы данных ведется ее собственный журнал транзакций.

Общая структура журнала условно может быть представлена в виде некоторого последовательного файла, в котором фиксируется каждое изменение БД, происходящее в ходе выполнения транзакции. Все транзакции имеют свои внутренние номера, поэтому в едином журнале транзакций фиксируются все изменения, проводимые всеми транзакциями.

Каждая запись в журнале транзакций содержит номер транзакции, к которой она относится, и значения атрибутов, которые она меняет. Кроме того, для каждой транзакции в журнале фиксируется команда начала и завершения транзакции.

Что же в действительности происходит в процессе регистрации транзакции?

Когда пользователь выполняет запрос на изменение базы данных, СУБД автоматически вносит в журнал транзакций одну запись для каждой строки, измененной в процессе выполнения запроса. Эта запись содержит две копии строки. Одна копия представляет собой строку до изменения, а другая – после изменения. Только когда в журнале будет сделана запись, СУБД изменит физическую строку на диске. Затем, если пользователь выполняет команду COMMIT, в журнале отмечается конец транзакции. Если пользователь выполняет команду ROLLBACK, СУБД обращается к журналу и извлекает из него «исходные» копии строк, измененных во время транзакции. Используя эти копии, СУБД возвращает строки в прежнее состояние и таким образом отменяет изменения, внесенные в базу данных в ходе транзакции.

Однако назначение журнала транзакций гораздо шире. Он предназначен для обеспечения надежного хранения данных в БД. А это требование предполагает, в частности, возможность восстановления согласованного состояния базы данных после любого рода аппаратных и программных сбоев.

Возможны следующие ситуации, при которых требуется восстановление состояния базы данных.

1. Индивидуальный откат транзакции. Этот откат должен быть применен в следующих случаях:

- стандартная ситуация отката транзакции – ее явное завершение командой ROLLBACK;
- аварийное завершение работы прикладной программы, которое логически эквивалентно выполнению оператора ROLLBACK, но физически имеет иной механизм выполнения;
- принудительный откат транзакции в случае взаимной блокировки при параллельном выполнении транзакций. В подобном случае для выхода из тупика данная транзакция может быть выбрана в качестве жертвы и ее выполнение будет принудительно прекращено ядром СУБД.

2. Восстановление после внезапной потери содержимого оперативной памяти (мягкий сбой). Такая ситуация может произойти в следующих случаях:

- при аварийном выключении электрического питания;
- возникновении неустранимого сбоя процессора (например, при срабатывании контроля оперативной памяти). Ситуация характеризуется потерей той части базы данных, которая к моменту сбоя содержалась в буферах оперативной памяти.

3. Восстановление после поломки основного внешнего носителя базы данных (жесткий сбой). Эта ситуация при достаточно высокой надежности современных устройств внешней памяти может возникать сравнительно редко, но тем не менее СУБД должна быть в состоянии восстановить базу данных даже в этом случае. Основа восстановления – архивная копия и журнал транзакций.

Во всех трех случаях основа восстановления – данные, хранящиеся в журнале транзакций.

Для восстановления согласованного состояния базы данных при индивидуальном откате транзакции нужно устранить последствия команд модификации базы данных, которые выполнялись в этой транзакции. Для восстановления непротиворечивого состояния БД при мягком сбое необходимо восстановить содержимое БД по содержимому журналов транзакций, хранящихся на дисках. Для восстановления согласованного состояния БД при жестком сбое следует восстановить содержимое БД по архивным копиям и журналам транзакций, которые хранятся на неповрежденных внешних носителях.

Следует отметить, что журнал транзакций используется при старте СУБД для того, чтобы отменить сделанные изменения и установить состояние базы данных на момент, предшествующий началу изменений.

Например, при запуске MS SQL Server для каждой БД начинается процесс регенерации (recovery). MS SQL Server определяет те транзакции, которые необходимо откатить. Это происходит в том случае, когда неизвестно, все ли изменения из кэша записаны на диск. Поскольку при выполнении контрольной точки все изменения сбрасываются на диск, то с нее и стартует процесс регенерации, который проводит фиксацию транзакций на диск. Все изменения на страницах, сделанные до контрольной точки, уже записаны на диск, поэтому нет смысла для сброса их на диск еще раз и они не берутся к рассмотрению.

### Вопросы для самопроверки

1. Для чего используется журнал транзакций?
2. Какова общая структура журнала транзакций?
3. Что такое мягкий сбой?
4. Что такое жесткий сбой?

## *4.3.8. Индексирование таблиц*

***4.3.8.1. Способы размещения данных и доступа к данным в реляционных базах данных.*** На этапе физического проектирования решаются вопросы размещения хранимых данных в пространстве памяти и выбора эффективных методов доступа к ним.

Понимание того, как хранятся данные, – основа понимания того, как получить к ним доступ.

В реляционных БД данные хранятся в таблицах. Система управления базами данных MS SQL Server поддерживает два типа таблиц: таблицы-кучи, или просто кучи, и кластеризованные таблицы, имеющие кластеризованный индекс.

Для начала разберемся с таким понятием, как куча. Как только таблица создана, и в ней еще нет индексов, она выглядит как куча данных (heap). Куча – это набор страниц данных, содержащих строки

таблицы. Каждая страница данных содержит 8 Кб информации. Строки данных хранятся хаотично, без определенного порядка. Система управления базами данных вставляет в таблицу-кучу строки данных в той последовательности, в которой они поступают, добавляя их в конец страницы. По сути, это и есть нагромождение (куча) данных. Нет определенного порядка и для последовательности страниц. Когда страница заполнится данными, то они добавляются на новую страницу.

Для получения ответа на любой запрос к такой таблице-куче СУБД использует последовательный метод доступа к данным, который заключается в сканировании таблицы. Выражение «сканирование таблицы» означает, что система последовательно считывает в оперативную память компьютера каждую страницу таблицы, исследует каждую строку таблицы (от первой до последней) и помещает строку в результирующий набор запроса, если для нее удовлетворяется условие поиска в предложении WHERE. Таким образом, все строки извлекаются в соответствии с их физическим расположением в памяти. Просканировать таблицу, содержащую небольшое количество строк, проблем не составляет. Однако в случае больших таблиц такой процесс приводит к быстрому падению производительности.

Один из важнейших путей достижения высокой производительности баз данных – использование индексов.

Суть использования индексов легко пояснить на примере с поиском нужного материала в книге. Вряд ли вы станете поочередно перелистывать сотни страниц, чтобы добраться до главы с интересующим вас материалом. Вы откроете страницу с содержанием книги, найдете нужную главу и перейдете на указанную там страницу.

Индексация позволяет задействовать данный метод и в работе БД, которая с помощью созданного индекса быстро находит данные по запросу. Роль содержания в базе данных выполняют индексы.

*Индекс* – это объект базы данных, который определяет соответствие значения ключа строки и местоположения этой строки на внешнем запоминающем устройстве.

Столбцы, входящие в индекс, принято называть *ключевыми полями (key fields), или ключами*. Таблицу, для которой используется индекс, называют *индексированной.*

Индекс представляет собой файл особого типа, в котором каждая запись состоит из двух значений: данных и указателя. Данным является значение поля, по которому производится индексирование, т. е. значение ключа, а указатель связывает ключ с соответствующей строкой индексированного файла, т. е. файла, в котором хранится индексируемая таблица. Основное преимущество использования индексов заключается в значительном ускорении процесса выборки данных за счет упорядочивания значений индексируемого столбца. Строка индекса значительно короче, чем строка таблицы, на которую он указывает. Поэтому индекс занимает меньше страниц памяти, чем сама таблица, и, следовательно, система тратит меньше времени на чтение индекса, чем на чтение таблицы.

Однако индексы имеют недостаток, который состоит в замедлении процесса обновления данных в индексируемой таблице. Действительно, после обновления таблицы – добавления или удаления строк, а также изменения значений индексируемых полей – индекс приводится в соответствие с последней версией данных таблицы. Обновление индекса, естественно, занимает некоторое время (иногда очень большое).

Индексы создаются не только для таблиц, но и для представлений. Каждая таблица или представление БД могут иметь один или несколько индексов. Индексы создаются как по одному столбцу, так и по нескольким столбцам таблицы.

Обращение к строке таблицы через индексы выполняется в два этапа: сначала СУБД считывает индекс в оперативную память (ОП) и находит в нём требуемое значение ключа и соответствующий адрес строки, а затем по этому адресу происходит обращение к внешнему запоминающему устройству.

Рассмотренные выше методы доступа к данным реализуются СУБД и не требуют специального программирования. Задача разработчика – определить столбцы, по которым следует индексировать таблицу, и выбрать тип каждого индекса.

***4.3.8.2. Кластеризованные и некластеризованные индексы.*** В существующих реляционных СУБД используют два типа индексов: кластеризованные и некластеризованные.

Для того чтобы правильно выбрать типы индексов, которые будут использоваться в проектируемой базе данных, необходимо разобраться, в чем различие этих индексов.

Наиболее популярный подход к организации индексов в базах данных – использование техники B-деревьев. С точки зрения логического представления B-дерево – это сбалансированное дерево поиска, в котором каждый узел содержит множество ключей и имеет большое число узлов-потомков.

Ключи хранятся в некотором сортированном порядке, чтобы допускать двоичный поиск.

Узлы дерева логически разбиваются на три группы.

1. *Корневой узел.* Этот узел не имеет предков и является вершиной дерева.

2. *Внутренние узлы.* Это узлы, соединяющие корень с листьями. Обычно имеется более одного уровня внутренних узлов.

3. *Узлы листьев.* Это узлы самого нижнего уровня, которые не имеют дочерних узлов.

Сбалансированность означает, что все самые нижние узлы (листья) дерева находятся на расстоянии одинакового количества уровней от вершины (корневого узла) дерева. Это свойство поддерживается даже тогда, когда в индексированный столбец добавляются или удаляются данные.

Параметры B-дерева – порядок $n$ и количество уровней. Порядок $n$ – это количество ссылок из узла $i$-го уровня на узлы ($i$+1) уровня. Каждый узел дерева должен иметь не более $n$ – 1 ключей.

С точки зрения физической организации B-дерево представляется как мультисписочная структура страниц внешней памяти, т. е. каждому узлу дерева соответствует блок внешней памяти (страница). Поэтому в литературе очень часто термины «узел» и «страница» используют как синонимы. B-деревья имеют очень высокий порядок (обычно значение $n \geq 100$), что снижает количество операций ввода-вывода, требующих поиска элемента в дереве.

Не вдаваясь в подробности теории B-деревьев, схематично структуру сбалансированного дерева можно изобразить так, как на рис. 4.53. На данном рисунке предполагается, что значения поля, по которому была проиндексирована таблица, находятся в диапазоне от 1 до 100.

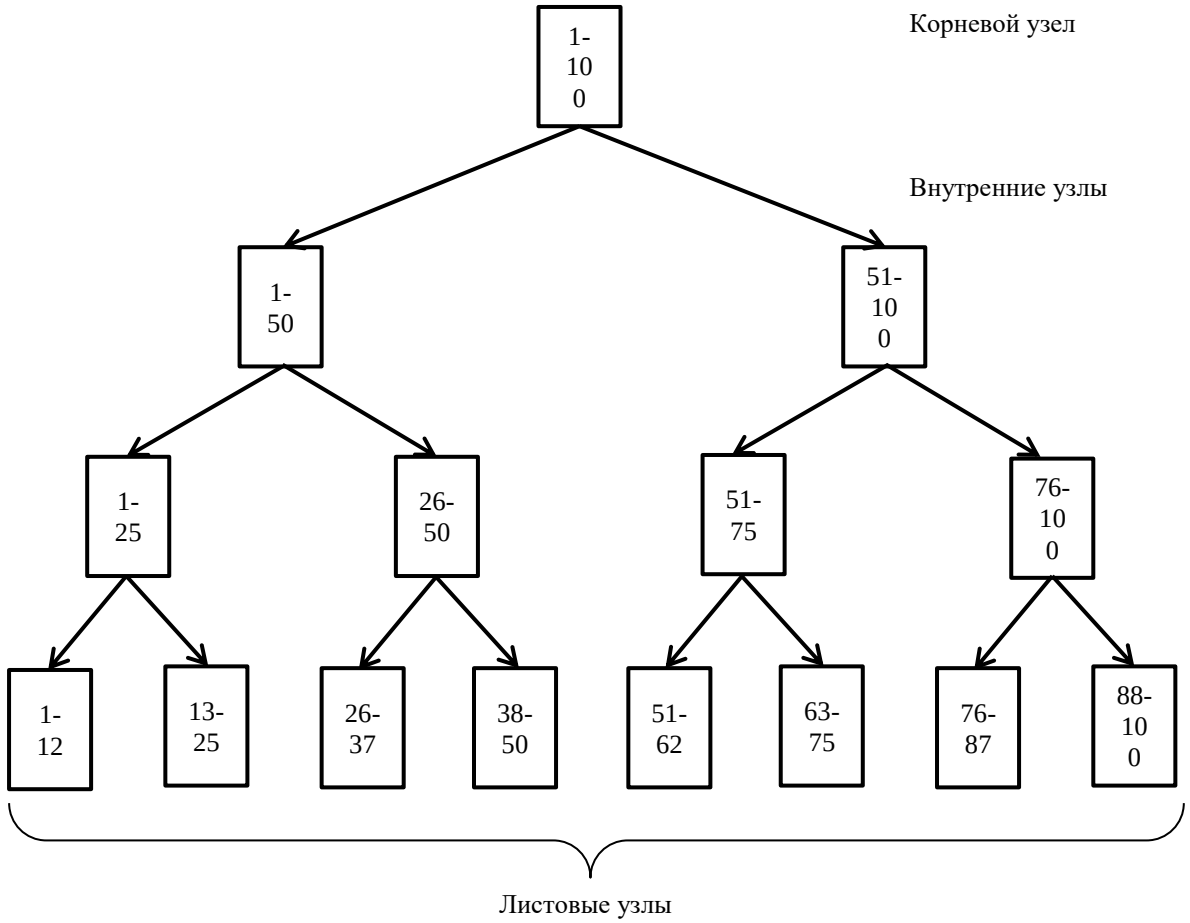



Рис. 4.53. Структура сбалансированного дерева

Когда формируется запрос, в котором используется индексированный столбец, СУБД начинает идти сверху от корневого узла и постепенно двигается вниз через внутренние узлы, при этом каждый слой внутреннего уровня содержит более детальную информацию о данных. Система управления базами данных продолжает двигаться по узлам индекса до тех пор, пока не достигнет нижнего уровня с листьями индекса.

Например, если надо найти значение 43 в индексированном столбце, то СУБД сначала на корневом уровне определит страницу на первом внутреннем уровне. В данном случае первая страница указывает на значения от 1 до 50, а вторая – от 51 до 100. Таким образом, СУБД обратится к первой странице этого внутреннего уровня. Далее будет выяснено, что следует обратиться ко второй странице следующего внутреннего уровня. Отсюда СУБД прочитает на нижнем уровне значение самого индекса. В зависимости от типа индекса листья индекса могут содержать как сами данные таблицы, так и указатель на

строки с данными в индексированной таблице. В первом случае индекс кластеризованный, а во втором – некластеризованный.

*Кластеризованный индекс* – это индекс, в котором логический порядок значений ключа индекса определяет физический порядок соответствующих строк в таблице, т. е. кластеризованный индекс физически *переупорядочивает данные таблиц* по ключу индекса. На корневом и внутренних узлах такого индекса находятся страницы индекса. Каждая строка индекса содержит ключевое значение и указатель либо на страницу внутреннего уровня сбалансированного дерева, либо на строку данных на конечном уровне индекса. Важное свойство кластеризованного индекса – та особенность, что его листья дерева (узлы-листья) содержат страницы данных. Таким образом, кластеризованный индекс – это, по сути, таблица данных, все строки которой отсортированы в порядке возрастания или убывания значений ключа индекса. Поэтому для таблицы или представления в каждый момент времени может существовать только один кластеризованный индекс. Не создавать же для другого индекса свою копию таблицы.

*Некластеризованный индекс,* в отличие от кластеризованных индексов, не меняет физический порядок расположения данных в таблице. Если для таблицы определить один или более некластеризованный индекс, физический порядок строк этой таблицы не будет изменен. Для каждого некластеризованного индекса СУБД создает дополнительную индексную структуру, которая сохраняется в индексных страницах и отделена от данных в таблице.

Второе отличие от кластеризованных индексов заключается в том, что страницы листьев некластеризованного индекса состоят из ключей индекса и закладок.

Закладка в некластеризованном индексе указывает, где находится строка, соответствующая ключу индекса. Составляющая закладки индекса может быть двух видов в зависимости от того, является ли таблица кластеризованной или кучей. Если в таблице нет кластеризованного индекса и индексируемая таблица является кучей, то закладка идентична идентификатору строки (RID – Row Identifier), состоящему из трех частей: адреса файла, в котором хранится таблица, адреса физического блока (страницы), в котором хранится строка, и смещения строки в странице. Если существует кластеризованный индекс,

то закладка некластеризованного индекса показывает сбалансированное дерево кластеризованного индекса таблицы.

Для поиска данных СУБД использует некластеризованный индекс двумя способами.

1. Если имеется куча (т. е. таблица без кластеризованного индекса), то система сначала выполняет обход некластеризованного индекса, а затем извлекает строку, используя идентификатор строки.

2. В случае кластеризованной таблицы после обхода структуры некластеризованного индекса следует обход структуры кластеризованного индекса таблицы.

В обоих случаях количество операций ввода-вывода довольно велико, поэтому следует подходить к проектированию некластеризованного индекса с осторожностью и применять его только в том случае, если есть уверенность, что его использование существенно повысит производительность.

***4.3.8.3. Создание индексов.*** Средства языка SQL предлагают несколько способов определения индекса:

- автоматическое создание кластеризованного индекса при создании ограничения PRIMARY KEY (первичный ключ) при условии, что в таблице нет кластеризованного индекса и не указан уникальный некластеризованный индекс;
- автоматическое создание уникального некластеризованного индекса при определении ограничения целостности UNIQUE;
- создание индекса с помощью команды CREATE INDEX.

В двух первых способах определения индекса, после того как индекс создан как часть ограничения, он автоматически получает то же имя, что и ограничение. Если у ограничения нет имени, то индекс получает имя по умолчанию в соответствии с правилом, существующим в СУБД.

Команда CREATE INDEX имеет следующий синтаксис:

```
CREATE [UNIQUE]
        [CLUSTERED/NONCLUSTERED]
    INDEX <имя_индекса>
        ON [<владелец.>] <имя_таблицы> / <имя представления>
        (<имя_столбца> [ASC/DESC] )
    [WITH [PAD_INDEX]
```

[ [,] FILLFACTOR = x]
[ [,] IGNORE_DUP_KEY]
[ [,] DROP_EXISTING]
[ [,] STATISTICS_NORECOMPUTE] ]
[ON <имя_группы_ файлов>]

Параметры, используемые в этом синтаксисе, имеют следующие значения.

Параметр UNIQUE указывает, что повторяющиеся значения ключевого поля индекса не допускаются. В уникальном составном индексе уникальной должна быть комбинация значений всех столбцов каждой строки. Если ключевое слово UNIQUE не указывается, то повторяющиеся значения в проиндексированном столбце (столбцах) разрешаются. В индексируемом столбце желательно запретить хранение значений NULL, чтобы избежать проблем, связанных с уникальностью значений. После того как для столбца появится уникальный индекс, СУБД не разрешит выполнение команд INSERT и UPDATE, которые приведут к появлению дублирующих значений.

Параметр CLUSTERED определяет кластеризованный индекс, который заставляет СУБД физически хранить данные в таблицах в том порядке, который задает этот индекс, а вся добавляемая информация станет приводить к изменению физического порядка данных. Кластеризованным может быть только один индекс в таблице. Если при создании кластеризованного индекса не указать условие уникальности, то MS SQL Server автоматически создает скрытое дополнительное значение ключа, обеспечивающее уникальность индекса. В то же время, если задать условие уникальности явно, то для MS SQL Server отпадет необходимость в создании этого ключа.

Параметр NONCLUSTERED определяет создание некластеризованного индекса как полностью отдельного от таблицы (данных таблицы) объекта. Этот параметр установлен по умолчанию.

Параметр имя_индекса должен быть уникальным в таблице. Это означает, что одно и то же имя индекса может быть дано двум индексам при условии, что они созданы в разных таблицах базы данных.

Параметр имя_таблицы определяет имя таблицы, которая индексируется. Оно может включать в себя полную спецификацию имени таблицы:

[[база данных] . владелец].

Параметр имя_представления – имя представления, содержащего индексируемые столбцы.

Параметр имя столбца – имя индексируемого столбца (или столбцов). Если указано более одного столбца, создается *составной индекс*. Имена столбцов в составном индексе должны быть отделены друг от друга запятыми. Для составного индекса существуют определенные ограничения, связанные с его размером и количеством столбцов: индекс может иметь размер максимум 900 байт и не более 16 столбцов.

Параметр ASC|DESC определяет направление упорядочения значений ключа индекса. Если указано ASC, то создается восходящий индекс (например, упорядоченный от A до Z). При задании DESC создается нисходящий индекс (упорядоченный от Z до A). По умолчанию создается восходящий индекс.

Рассмотрим простейший пример создания индекса для таблицы СКЛАД:

```
CREATE UNIQUE INDEX Name_Product_Ind
ON Склад (Наименование)
```

В результате выполнения этой команды для таблицы СКЛАД будет создан некластеризованный индекс с именем Name_Product_Ind, ключом которого является столбец «Наименование». Повторяющиеся значения ключевого поля индекса не допускаются, а значения ключа индекса отсортированы по возрастанию.

Параметр FILLFACTOR = x определяет в процентах полноту заполнения каждой страницы индекса во время его создания. Значение параметра x можно устанавливать в диапазоне от 1 до 100.

Если данные в таблице изменяются крайне редко или вообще не изменяются, то можно установить значение x = 100, что заставит СУБД заполнять страницы индекса на 100 %, т. е. существующая страница листа, так же как страница, не относящаяся к листу, не будет иметь свободного места для вставки новых строк.

Если данные таблицы пополняются часто, то лучше оставить больше свободного места. При значении параметра x от 1 и до 99 страницы листьев создаваемой структуры индекса будут содержать свободное место. Чем больше значение x, тем меньше свободного ме-

ста в страницах листьев индекса. Например, при значении x = 60 каждая страница листьев индекса будет иметь 40 % свободного места для вставки строк индекса в дальнейшем.

Использование параметра FILLFACTOR заметно оптимизирует работу СУБД, так как освобождает ее от необходимости распределять данные по разным страницам, если индексная информация слишком велика и не помещается на одну страницу (размер страницы – 8 Кб). Разбивка страниц при выполнении операций ввода-вывода – дорогостоящая операция, и по возможности ее следует избегать. Поэтому параметр FILLFACTOR рекомендуется использовать для таблиц, в которые часто вставляются данные или в которых часто изменяется индексное значение.

Удачно задав фактор заполнения, можно значительно ускорить выполнение запросов. Однако для этого требуется практический опыт работы с базами данных.

Следует учитывать, что использование параметра FILLFACTOR существенно влияет на требуемый объем памяти. Например, при задании значения x = 50 объем памяти, занимаемый таблицей, увеличится примерно вдвое.

Параметр FILLFACTOR особенно полезен при использовании кластеризованного индекса, поскольку страницы его нижнего уровня фактически идентичны страницам данных.

Параметр PAD_INDEX указывает, что значение параметра FILLFACTOR применяется как к страницам индекса, так и к страницам данных в индексе. Поэтому параметр PAD_INDEX применяется совместно с параметром FILLFACTOR, который в основном задает объем свободного пространства в процентах от общего объема страниц листьев индекса.

Параметр IGNORE_DUP_ KEY контролирует поведение СУБД, когда в таблице с уникальным индексом появляются повторяющиеся строки. По умолчанию СУБД отказывается добавлять повторяющиеся строки и возвращает ошибку. Опция IGNORE_DUP_ KEY заставляет СУБД продолжать обработку так, как будто ошибочного состояния не было.

Параметр DROP_EXISTING указывает, что существующий кластеризованный или некластеризованный индекс нужно удалить и со-

здать заново указанный индекс. Фактически это перестройка (rebuild) индекса, которая необходима при фрагментации данных и удалении кластеризованного индекса.

Фрагментация данных возможна при любых изменениях данных в таблице. Если эти данные проиндексированы, то также возможна фрагментация индекса, когда информация индекса оказывается разбросанной по разным физическим страницам. В результате фрагментации данных индекса СУБД может быть вынуждена выполнять дополнительные операции чтения данных, что понижает общую производительность системы. В таком случае требуется перестроить все фрагментированные индексы. Параметр DROP_EXISTING позволяет повысить производительность при пересоздании кластеризованного индекса таблицы, которая также имеет некластеризованные индексы.

Кроме того, каждый некластеризованный индекс в кластеризованной таблице содержит в своих листьях дерева соответствующие значения кластеризованного индекса таблицы. По этой причине при удалении кластеризованного индекса таблицы требуется создать вновь все ее некластеризованные индексы. Таким образом, использование параметра DROP_EXISTING позволяет избежать повторного пересоздания некластеризованных индексов.

Параметр STATISTICS_NORECOMPUTE определяет функции автоматического обновления статистики для таблицы. Система управления базами данных автоматически вычисляет и хранит статистические данные о составе индексов. При желании средство статистического учета можно отключить с помощью опции STATISTICS_NORECOMRUTE. В общем случае отключать средство автоматического обновления статистики индексов не рекомендуется, поскольку программа оптимизации СУБД использует ее для повышения эффективности выполнения запросов. Для его повторного включения требуется выполнить команду UPDATE STATISTICS без ключевого слова NORECOMPUTE.

Параметр имя_группы_файлов позволяет осуществить выбор файловой группы, в которой будет находиться создаваемый индекс. Естественно, что файловая группа к этому моменту уже должна существовать. Использование индекса из другой файловой группы повышает производительность кластеризованных индексов в связи с па-

раллельностью выполнения процессов ввода-вывода и работы с самим индексом. Если группа файлов не указана, то индекс создается для группы файлов, заданной по умолчанию.

Приведем еще пример создания индексов. Создадим уникальный кластеризованный индекс с именем Name_RegionInd для таблицы ПОСТАВЩИК по столбцам «Название» и «Регион». Кроме того, значения ключа индекса будут упорядочены по убыванию. Также запретим автоматическое обновление статистики при изменении данных в таблице и установим фактор заполнения индексных страниц на уровне 60 %.

```
CREATE UNIQUE NONCLUSTERED
  INDEX Name_RegionInd
  ON Поставщик (Название DESC, Регион DESC)
  WITH
     FILLFACTOR=60,
     STATISTICS_NORECOMPUTE
```

***4.3.8.4. Выбор способа индексации.*** Теперь, когда рассмотрены способы создания индексов и их доступные параметры, возникает закономерный вопрос: как проектировать индексы, т. е. какие индексы и для каких столбцов следует использовать в каждой конкретной ситуации.

Приступая к проектированию индексов, важно помнить следующее:

- каждый индекс занимает определенный объем дискового пространства, следовательно, существует вероятность того, что общее количество страниц индекса базы данных может превысить количество страниц данных в базе;
- в отличие от получения выгоды при использовании индекса для выборки данных, вставка и удаление данных такой выгоды не предоставляют по причине необходимости обслуживания индекса. Чем больше индексов имеет таблица, тем больший требуется объем работы по их реорганизации.

Надо четко понимать, что проектирование индексов – очень сложная и ответственная работа, требующая достаточного опыта и профессионализма. Дело в том, что созданные тщательным образом

индексы способны улучшить производительность системы, а созданные непрофессионально – понизить производительность системы.

Потому при индексировании таблиц следует придерживаться следующих эмпирических рекомендаций, которые относятся как к базам данных в целом, так и к запросам, направленным к ним.

1. При модификации данных в БД основная часть работы системы связана с обновлением индексов. Поэтому в базе данных, в которой часто добавляются или удаляются данные, лучше создавать только самые необходимые индексы, ограничивая количество индексов в таблице данных. Как правило, в таких системах «магическое число» – пять индексов. Если оно превышено, производительность резко падает.

2. В базе данных, в которой большей частью выполняется считывание данных на фоне одного или нескольких случайных добавлений, а основная задача – анализ существующей информации, можно реализовать все индексы, необходимые для повышения производительности запросов. Ограничивающий фактор в этом случае – дисковое пространство, поскольку каждый индекс требует дополнительного пространства в базе данных.

3. Для таблиц небольшого объема не всегда целесообразно использовать индексы. Такой поиск может выполняться дольше, чем обычное сканирование таблицы.

4. Подходящие кандидаты для ключей индекса следующие:

- столбцы первичных ключей. При создании первичного ключа автоматически создается кластеризованный индекс;
- столбцы, входящие в условие соединения таблиц (внешние ключи). В случае операции соединения рекомендуется создавать индекс для каждого соединяемого столбца;
- столбцы, непосредственно указанные в предложении WHERE. Причем, если условие поиска часто используемого запроса содержит операторы AND, лучше всего создать составной индекс по всем столбцам таблицы, указанным в предложении WHERE команды SELECT.

Рассмотрим следующий пример. Предположим, что из спроектированной базы данных (см. приложение) требуется получить список поставщиков и номера документов, по которым они поставляли

товар после 1 января 2022 г. Это можно сделать с помощью следующей команды

```
SELECT Название, Ном_док
FROM Поставщик INNER JOIN Документ
ON Поставщик. Код_постав = Документ. Код_постав
WHERE Дата > '01.01.2022'
```

Запрос содержит операцию соединения, в которой участвуют столбцы родительской таблицы ПОСТАВЩИК и дочерней таблицы ДОКУМЕНТ. Соединяемые столбцы представляют собой первичный ключ Код_постав таблицы ПОСТАВЩИК и соответствующий внешний ключ Код_постав таблицы ДОКУМЕНТ.

При создании первичного ключа СУБД неявно создает кластеризованный индекс. Следовательно, кластеризованный индекс для столбца первичного ключа Код_постав таблицы ПОСТАВЩИК уже существует. Поэтому в рассматриваемом примере надо создать некластеризованный индекс для столбца внешнего ключа Код_постав таблицы ДОКУМЕНТ:

```
CREATE INDEX Id_supp _ Ind
ON Документ (Код_постав)
```

Кроме этого, для столбца «Дата» таблицы ДОКУМЕНТ, который присутствует в условии предложения WHERE, следует создать дополнительный некластеризованный индекс:

```
CREATE INDEX Date _ Ind
ON Документ (Дата)
```

*Примечание.* Иногда вместо первичного ключа в соединении используется столбец, который имеет ограничение для обеспечения целостности UNIQUE. В этом случае для кластеризованной родительской таблицы следует создать некластеризованный индекс для столбца со свойством UNIQUE и некластеризованный индекс для столбца внешнего ключа.

5. Индексированию не подлежат следующие типы столбцов.

- Столбцы, входящие в условия предложения WHERE с низким уровнем селективности (более 80 %). Под селективностью условия понимается соотношение количества строк, удовлетворяющих усло-

вию, к общему количеству строк в таблице. Высокая селективность соответствует меньшему значению этого соотношения. Например, если в таблице ПОСТАВЩИК 85 % составляют поставщики из Владимирской области, то создавать индекс по столбцу «Регион» не имеет смысла. Простое сканирование таблицы – самый эффективный план выполнения запроса. Фраза «самый эффективный план» означает здесь наибольшую эффективность с точки зрения минимизации количества прочитанных страниц.

• Совершенно неуникальные столбцы, например содержащие пол служащего (мужской или женский), или столбцы типа BIT.

• Слишком широкие столбцы. Например, если столбец имеет ширину 30 символов, то вряд ли его индексация окажется полезной. По этой причине не подлежат индексации столбцы типа TEXT, IMAGE.

6. Производительность индекса напрямую зависит от того, насколько уникальны значения в столбце. Она снижается с увеличением дубликатов значений в столбце и растет – с уменьшением. Потому по возможности следует использовать уникальный индекс.

Эти рекомендации будет разумно применять при проектировании индексов для частых запросов, а затем оценивать их использование. Поэтому можно сделать вывод, что работа с индексами – это эксперимент и еще раз эксперимент.

***4.3.8.5. Удаление индексов.*** Удалять можно только созданные вами индексы. Для этого используется команда DROP INDEX. В общем виде команда удаления индекса выглядит следующим образом:

DROP INDEX <имя_индекса > ON <имя_таблицы >]

В следующем примере удаляется созданный ранее индекс:

DROP INDEX Id_supp _ Ind

Команду DROP INDEX нельзя использовать для удаления индекса, который был автоматически создан на ограничения PRIMARY KEY или UNIQUE. Для удаления ограничения и соответствующего индекса используется команда ALTER TABLE с предложением DROP CONSTRAINT:

ALTER TABLE <имя_таблицы> DROP CONSTRAINT <имя_индекса>

Обращаем внимание, что если удалить кластеризованный индекс, то все некластеризованные индексы должны быть перестроены, так как все они используют ключ кластеризованного индекса как указатель в своих страницах узлов.

## Вопросы для самопроверки

1. Что такое индекс?
2. Для чего используются индексы?
3. Какова структура индекса?
4. Какие типы индексов вы знаете?
5. Каковы особенности кластеризованных индексов?
6. Каковы особенности некластеризованных индексов?
7. Какие существуют способы создания индексов?
8. Для чего используется параметр FILLFACTOR = x команды CREATE INDEX?
9. С какой целью используется параметр DROP_EXISTING команды CREATE INDEX?
10. Какие правила создания индексов вы знаете?
11. Каким образом можно удалить индекс?
12. Как удалить индекс, который был неявно создан для первичного ключа таблицы?

## Практические задания

1. Напишите команду CREATE INDEX для создания некластеризованного индекса для столбца «Название» таблицы ПОСТАВЩИК с заполнением пространства каждой страницы листьев индекса на 70 %.

2. Напишите команду CREATE INDEX для создания некластеризованного составного индекса для столбцов «Регион» и «Рейтинг» таблицы ПОСТАВЩИК.

3. Напишите команду CREATE INDEX для создания индексов, которые повысят производительность запросов:

а) SELECT Название, Регион
FROM Поставщик
WHERE Рейтинг > 50

б) SELECT Наименование, Цена, Количество
FROM Склад
WHERE Наименование = ‘Принтер Canon’
AND Цена < 5000

в) SELECT Дата
FROM Документ INNER JOIN Приход
ON Документ. Ном_док = Приход. Ном_док

г) SELECT Наименование, Ном_док
FROM Приход INNER JOIN Склад
ON Приход. Ном_ном = Склад. Ном_ном
WHERE Тип = ‘Монитор’

# ЗАКЛЮЧЕНИЕ

В пособии рассмотрены методология проектирования баз данных и основные задачи, решаемые на этапах инфологического, логического и физического проектирования БД.

Авторы ставили перед собой задачу не только изложить материал по теории проектирования баз данных, но и дать примеры для грамотного использования полученных знаний. Рассмотренные в пособии примеры позволят студентам проектировать и создавать базы данных.

Надеемся, что представленный в пособии учебный материал сыграет важную роль в освоении технологий баз данных, позволит грамотно проектировать БД, что является одним из ключевых требований к любому специалисту в области компьютерных технологий.

Пособие основано на материалах лекций, которые читаются авторами обучающимся в Институте информационных технологий и электроники Владимирского государственного университета.

# БИБЛИОГРАФИЧЕСКИЙ СПИСОК


1. *Дейт, К. Дж.* Введение в системы баз данных / К. Дж. Дейт. – М. : Вильямс, 2005. – 1328 с. – ISBN 5-8459-0788-8.

2. *Хомоненко, А. Д.* Базы данных : учеб. для вузов / А. Д. Хомоненко, В. М. Цыганков, М. Г. Мальцев ; под ред. А. Д. Хомоненко. – 6-е изд., доп. – СПб. : КОРОНА – Век, 2009. – 736 с. – ISBN 5-7931-0284-1.

3. *Коннолли, Т.* Базы данных: проектирование, реализация и сопровождение. Теория и практика / Т. Коннолли, К. Бегг. – М. : Вильямс, 2018. – 1439 с. – ISBN 978-5-8459-2020-1.

4. *Марка, Д. А.* Методология структурного анализа и проектирования / Д. А. Марка, К. МакГоуэн. – М., 1993. – 242 с. – ISBN 5-7395-0007-9.

5. *Фаулер, М.* UML в кратком изложении. Применение стандартного языка объектного моделирования / М. Фаулер, К. Скотт. – М. : Мир, 1999. – 191 с. – ISBN 5-03-003331-9.

6. *Крёнке, Д.* Теория и практика построения баз данных / Д. Крёнке. – СПб. : Питер, 2005. – 800 с. – ISBN 5-94723-583-8.

7. *Чен, П.* Модель «сущность – связь» – шаг к единому представлению данных / П. Чен // СУБД. – 1995. – № 3. – С. 2 – 31.

8. *Кара-Ушанов, В. Ю.* Модель «сущность – связь» [Электронный ресурс] / В. Ю. Кара-Ушанов. – URL: https://study.urfu.ru/Aid/Publication/13604/1/Kara-Ushanov.pdf (дата обращения: 30.05.2022).

9. Подтипы сущностей. Супертип. Пример. Преимущества и недостатки применения подтипов сущностей [Электронный ресурс]. – URL: https://www.bestprog.net/ru/2019/01/27/entity-subtypes-supertype-example-advantages-and-disadvantages-of-using-subtypes-of-entities-ru/ (дата обращения: 30.05.2022).

10. Нормальная форма Бойса – Кодда (BCNF) [Электронный ресурс]. – URL: https://info-comp.ru/boyes-codd-normal-form (дата обращения: 02.06.2022).

11. *Гарсиа-Молина, Г.* Системы баз данных : Полный курс / Г. Гарсиа-Молина, Дж. Д. Ульман, Дж. Уидом ; [пер. с англ. и ред. А. С. Варакина]. – М. : Вильямс, 2003. – 1083 с. – ISBN 5-8459-0384-X.

12. *Градусов, А. Б.* Базы данных : Введение в технологию баз данных : учеб.-практ. пособие / А. Б. Градусов. – Владимир : Изд-во ВлГУ, 2021. – 208 с. – ISBN 978-5-9984-1226-4.

# ПРИЛОЖЕНИЕ

## SQL-скрипт спроектированной базы данных «Контроль_товара»

```
CREATE TABLE Поставщик (
    Код_постав INT IDENTITY(1,1) NOT NULL PRIMARY KEY,
    Название VarChar(20) NOT NULL,
    Регион VarChar (15),
    Рейтинг INT NOT NULL CHECK (Рейтинг BETWEEN 0
    AND 100) DEFAULT (Рейтинг =0)
)
CREATE TABLE Документ (
    Ном_док INT NOT NULL PRIMARY KEY,
    Дата DateTime NOT NULL DEFAULT (Дата = GetDate()),
    Код_постав INT NOT NULL,
    CONSTRAINT FK_ Doc FOREIGN KEY (Код_постав)
    REFERENCES Поставщик (Код_постав)
    ON UPDATE CASCADE
 )
CREATE TABLE Склад (
    Ном_ном INT NOT NULL PRIMARY KEY,
    Наименование VarChar(20) NOT NULL,
    Цена MONEY NOT NULL,
    Количество INT NOT NULL
 )
CREATE TABLE Приход (
    Ном_док INT NOT NULL,
    Ном_ном INT NOT NULL,
    Цена MONEY NOT NULL,
    Количество INT NOT NULL,
    Тип VarChar(20) NOT NULL,
    CONSTRAINT PK_ Com PRIMARY KEY (Ном_док,
    Ном_ном),
    CONSTRAINT FK_ Com_1 FOREIGN KEY (Ном_док)
```

```
REFERENCES Документ (Ном_док)
    ON DELETE CASCADE,
    CONSTRAINT FK_ Com_2 FOREIGN KEY (Ном_ном)
 REFERENCES Склад (Ном_ном)
 )
CREATE TABLE Зарубежный_поставщик (
    Код_постав INT NOT NULL PRIMARY KEY,
    Валюта VarChar(10) NOT NULL,
    Язык VarChar(15) NOT NULL,
    CONSTRAINT FK_ For_ sup FOREIGN KEY (Код_постав)
REFERENCES Поставщик (Код_постав)
ON UPDATE CASCADE
ON UPDATE CASCADE
)
CREATE TABLE Отечественный_поставщик (
    Код_постав INT NOT NULL PRIMARY KEY,
    Форма_ собственности VarChar(20) NOT NULL,
    CONSTRAINT FK_ Dom_ sup FOREIGN KEY (Код_постав)
REFERENCES Поставщик (Код_постав)
ON UPDATE CASCADE
ON UPDATE CASCADE
)
```

*Учебное электронное издание*

ГРАДУСОВ Александр Борисович
ШУТОВ Антон Владимирович

БАЗЫ ДАННЫХ

Проектирование баз данных

Учебно-практическое пособие

Редактор Т. В. Евстюничева

Технический редактор Ш. Ш. Амирсейидов

Корректор Н. В. Пустовойтова

Компьютерная верстка Л. В. Макаровой

Выпускающий редактор А. А. Амирсейидова

***Системные требования:*** Intel от 1,3 ГГц; Windows XP/7/8/10; Adobe Reader; дисковод CD-ROM.

**Тираж 9 экз.**

Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых
Изд-во ВлГУ
rio.vlgu@yandex.ru

Институт информационных технологий и электроники
кафедра вычислительной техники и систем управления
a1981@mail.ru