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

# БАЗЫ ДАННЫХ

## Введение в технологию баз данных

Учебное-практическое пособие

Владимир 2021

УДК 004.65
ББК 32.973
Г75

Рецензенты:
Доктор технических наук, профессор
профессор кафедры информационных систем и программной инженерии
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых
*Е. Р. Хорошева*

Генеральный директор ООО «АйТим»
*Е. А. Уланов*

**Градусов, А. Б.**
Г75 Базы данных : Введение в технологию баз данных : учеб.-практ. пособие / А. Б. Градусов ; Владим. гос. ун-т. им. А. Г. и Н. Г. Столетовых. – Владимир : Изд-во ВлГУ, 2021. – 208 с.
ISBN 978-5-9984-1226-4

Содержится обоснование концепции баз данных, описываются назначение и функции системы управления базами данных, рассматриваются особенности классических структур данных, дается описание основ языка запросов к реляционным базам данных SQL.

Предназначено для студентов бакалавриата очной и заочной форм обучения по направлениям 09.03.03 – Прикладная информатика и 27.03.04 – Управление в технических системах.

Рекомендовано для формирования общепрофессиональных компетенций в соответствии с ФГОС ВО.

Табл. 6. Ил. 105. Библиогр.: 8 назв.

УДК 004.65
ББК 32.973

ISBN 978-5-9984-1226-4

© ВлГУ, 2021
© Градусов А. Б., 2021

## ПРЕДИСЛОВИЕ

Технология баз данных одна из наиболее востребованных информационных технологий в практике информационных систем (ИС). В любой сфере деятельности человеку необходимо располагать какими-либо сведениями, документами, фактами, т.е. иметь определенную информацию. Другими словами, любая деятельность человека представляет собой процесс сбора и переработки информации, принятия на ее основе решений и их выполнения. А для принятия любых решений требуется четкая и точная оценка текущей ситуации и возможных перспектив ее изменений.

Хранить и перерабатывать большой объем информации в заданные сроки практически невозможно без специальных средств обработки информации. В качестве таких средств используют информационные системы.

В широком понимании под определение ИС подпадает любая система обработки информации. Эти системы составляют фундамент информационной деятельности во всех сферах, начиная с производства, управления финансами, телекоммуникациями и заканчивая управлением семейным бюджетом.

Массивы данных, хранящиеся в ИС, должны быть оптимальным образом организованы для их компьютерного хранения и обработки, обеспечивать коллективное использование данных многими пользователями.

Массив данных общего пользования в информационных системах называется *базой данных*. База данных (БД) является моделью предметной области информационной системы.

Цель учебного пособия состоит в формировании концептуальных представлений об основных принципах построения баз данных,

систем управления базами данных, моделях, описывающих базу данных, а также о манипулировании данными средствами языка SQL.

Рассмотрение изучаемых вопросов иллюстрируется на примере системы управления базами данных MS SQL-Server.

Разумеется, в пособии невозможно охватить все аспекты технологии баз данных. Издание отражает опыт автора в преподавании дисциплины «Базы данных». Приведенный список литературы поможет значительно расширить кругозор знаний о базах данных. Стоит отметить в этом списке фундаментальный труд одного из основоположников реляционной модели К. Дейта [1], а также используемый во многих зарубежных университетах курс Г. Гарсиа-Молины [2]. Из отечественных изданий необходимо обратить внимание на учебники И. П. Карповой [3], А. В. Кузина [4] и А. Д. Хомоненко [5]. Более глубоко изучить язык SQL поможет книга Дж. Гроффа [6]. И конечно же, большое количество информации содержится на бескрайних просторах Интернета.

# Глава 1. ВВЕДЕНИЕ В БАЗЫ ДАННЫХ

## 1.1. Принципы обработки данных

История возникновения и развития систем обработки данных органически связана с историей развития вычислительной техники.

На первом этапе (50-е годы прошлого века) компьютеры использовались для решения задач вычислительного характера, которые характеризует следующее:

- небольшой объем исходных данных;
- сложные алгоритмы обработки данных;
- исходные данные – числа;
- описание данных располагается в программе;
- исходные данные в программе используются однократно;
- исходные данные размещаются в оперативной памяти компьютера вместе с программой в так называемом сегменте данных (рис. 1.1).

Рис. 1.1. Представление программы и данных в памяти компьютера

Типичным примером такого класса задач является задача решения системы алгебраических уравнений одним из численных методов. Если система состоит из десяти уравнений, то число исходных данных будет составлять 110 чисел, а в результате решения получим 10 чисел. Естественно предположить, что повторно решать систему уравнений с теми же коэффициентами нет смысла.

Второй этап применения компьютеров (60-е годы прошлого века) характеризуется переходом к новому поколению ЭВМ и появлением нового класса задач, который относится к использованию средств вычислительной техники в автоматизированных информационных системах.

Эти задачи характеризуются:

- большим объемом исходных данных;
- использованием сложных структур данных;

- исходные данные хранятся в виде файлов на внешних устройствах памяти автономно от программы (рис. 1.2);
- программа содержит описание структуры файла;
- данные файлов используются многократно.

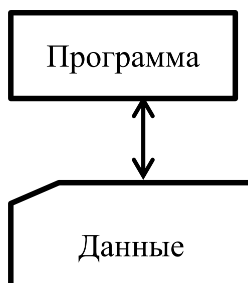



Рис. 1.2. Использование файлов для обработки данных

В ЭВМ этого периода времени в качестве внешних запоминающихся устройств, главным образом, использовались магнитные ленты, т.е. устройства последовательного доступа к данным. Это привело к тому, что при достаточно высоком быстродействии вычислений данные информационных задач хранились на "медленных" внешних запоминающих устройствах.

Можно предположить, что именно требования информационных задач вызвали появление съемных магнитных дисков с подвижными головками, что явилось революцией в истории вычислительной техники. Эти устройства внешней памяти обладали большой емкостью, обеспечивали удовлетворительную скорость доступа к данным в режиме произвольной выборки, а возможность смены дискового пакета на устройстве позволяла иметь практически неограниченный архив данных. С появлением магнитных дисков началась история систем управления данными во внешней памяти.

Однако файлы с произвольным доступом решили проблемы лишь частично. Файловые системы обладают рядом недостатков:

- **Зависимость программ от данных**. Сведения о структуре данных включались в код программы. При изменении структуры данных необходимо было вносить изменения в программу.
- **Избыточность данных**. Практика применения такого подхода показала, что при наличии нескольких программ, обрабатывающих файлы, возникают трудно преодолимые проблемы с обеспечением достоверности исходных данных. Предположим, что на предприятии отдельные службы решили автоматизировать часть своих функ-

ций с помощью ЭВМ. Тогда бухгалтерия для собственных целей создаст набор данных, содержащий сведения о рабочих и служащих предприятия, и использует этот набор для решения своих задач. Отдел кадров для своих задач создаст набор данных, который также содержит сведения о сотрудниках предприятия, причем часть данных этого набора отражает ту же информацию, что данные первого набора. В результате многие данные, хранящиеся в памяти ЭВМ, дублируются, что ведет к неоправданному расходу памяти. Избыточность данных также порождает риск противоречий между разными версиями общих данных. Например, если сотрудник изменил фамилию, а изменения внесены в разные файлы не одновременно, то через некоторое время такие расхождения могут существенно снизить качество информации, содержащейся в файлах данных.

- **Неэффективность параллельной работы нескольких пользователей.** Если операционная система поддерживает многопользовательский режим, вполне реальна ситуация, когда два или более пользователя одновременно пытаются работать с одним и тем же файлом. Если все пользователи собираются только читать файл, ничего страшного не произойдет. Но если хотя бы один из них будет изменять файл, для корректной работы этих пользователей требуется взаимная синхронизация их действий по отношению к файлу. В операционных системах обычно применялся следующий подход. В операции открытия файла среди прочих параметров указывался режим работы (чтение или запись). Если к моменту выполнения этой операции некоторым пользовательским процессом P2 файл был уже открыт другим процессом P1 в режиме записи, то в зависимости от особенностей системы процессу P2 либо сообщалось о невозможности открытия файла, либо он блокировался до тех пор, пока в процессе P1 не выполнялась операция закрытия файла. При подобном способе организации одновременная работа нескольких пользователей, связанная с модификацией данных в файле, либо вообще не реализовывалась, либо была очень замедлена.

Эти недостатки послужили тем толчком, который заставил разработчиков информационных систем предложить новый подход к организации данных. Этот подход предполагает хранение данных и их описания в одном месте (рис. 1.3).

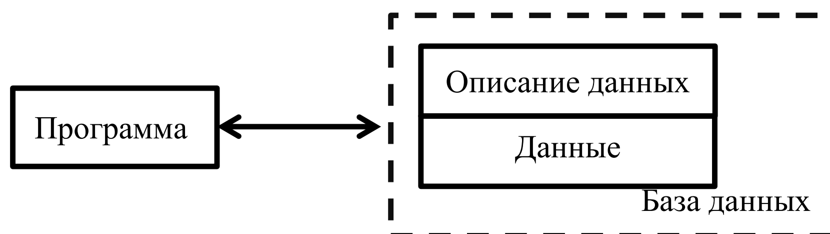



Рис. 1.3. Схема обработки данных с использованием баз данных

При таком подходе к обработке данных используется единая система взаимосвязанных файлов с минимальной избыточностью, получившая название база данных (БД).

Описание данных называют метаданными. Метаданные хранятся в части базы данных, которая называется каталогом или словарём-справочником данных (ССД). Зная формат метаданных, можно запрашивать и изменять данные без написания дополнительных программ.

Одна и та же база данных может быть использована для решения многих прикладных задач. Наличие метаданных и возможность информационной поддержки решения многих задач – это принципиальные отличия базы данных от любой другой совокупности данных, расположенных во внешней памяти ЭВМ.

## 1.2. Основные понятия баз данных

Прежде всего, разберемся с понятиями “данные” и “информация”. Данные представляют собой набор значений, характеризующих объект, процесс, явление и т.д. Сами по себе данные не несут никакой информации. Например, числовое данное равное 5 может быть количеством проданного товара, номером месяца, экзаменационной оценкой, ценой товара и т.п. Данные становятся информацией тогда, когда пользователь задает им определенную структуру, т. е. осознает их смысловое содержание. Например, в информационной системе торговой фирмы регистрируются продажи товаров. Для этого необходимы следующие данные: код товара, наименование товара, количество, цена, покупатель, дата.

Для каждой продажи соответствующие данные имеют конкретное значение. Например, факт, что фирма «НПП Автоматика» купила 21 марта 2020 года 500 транзисторов КТ209А по цене 7 рублей за

штуку, в информационной системе будет представлен в следующем виде (табл. 1.1)

Таблица 1.1. Факт конкретной продажи

| Код товара | Наименование товара | Количество | Цена | Покупатель | Дата |
|---|---|---|---|---|---|
| 1010 | транзистор КТ209А | 500 | 7 | НПП Автоматика | 21.03.2020 |

Значения, приведенные в табл. 1.1, имеют смысл только во взаимосвязи друг с другом. Отдельно выбранное число 500 теряет свой содержательный смысл.

Для описания аналогичных представлений данных в предметной области задач вводится ряд новых понятий.

***Элемент данных (поле) – наименьшая единица поименованных данных.***

Для данного примера элементами данных являются код товара, наименование товара, количество, цена, покупатель, дата.

Для описания продажи товара используется понятие «Запись».

***Запись –совокупность элементов данных (полей).***

Для представления всего набора продаж используется понятие «файл»

***Файл - поименованная совокупность всех экземпляров записей заданного типа.***

Каждый факт продажи товара содержится в одной записи файла. Файлы системы могут содержать огромное количество таких фактов. Таким образом, такой файл содержит данные. Информация же – это организованные данные или выводы из них. Например, информацией будут являться ответы на вопросы:

- Сколько было продано транзисторов КТ209А в январе 2020 года?
- Какие клиенты фирмы, и в каком количестве ежемесячно покупают товар данного наименования?

Обычно такая информация получается в результате обработки большого количества данных и, следовательно, информация отличается от данных.

Таким образом, можно сказать, что

**данные** – это информация, зафиксированная в некоторой форме, пригодной для последующей обработки, передачи и хранения, напри-

мер, находящаяся в памяти ЭВМ или подготовленная для ввода в ЭВМ;

**информация** – любые сведения о каком-либо событии, явлении, процессе и т.п., являющиеся объектом некоторых операций: восприятия, передачи, преобразования, хранения или использования.

В теории и практики баз данных используется следующая терминология [3]:

**Подготовка информации** состоит в её формализации, сборе и переносе на машинные носители.

**Обработка данных** – это совокупность задач, осуществляющих преобразование данных. Обработка данных включает в себя ввод данных в ЭВМ, отбор данных по каким-либо критериям, преобразование структуры данных, перемещение данных на внешней памяти ЭВМ, вывод данных, являющихся результатом решения задач, в табличном или в каком-либо ином удобном для пользователя виде.

### 1.3 Информационные системы, использующие концепцию баз данных

Под информационной системой (ИС) будем понимать совокупность программно-аппаратных средств, предназначенных для автоматизации деятельности, связанной с хранением, передачей и обработкой информации.

ИС в общем случае состоит из следующих компонентов: базы (несколько баз) данных, системы управления базами данных (СУБД), описания базы данных.

Архитектура таких систем обработки данных представлена на рис. 1.4.

Рассмотрим компоненты этой структуры.

**База данных** (БД) – совокупность данных, организованных по определённым правилам, предусматривающим общие принципы описания, хранения и манипулирования данными, независимая от прикладных программ [8]. Эти данные относятся к определённой предметной области и организованы таким образом, что могут быть использованы для решения многих задач многими пользователями.

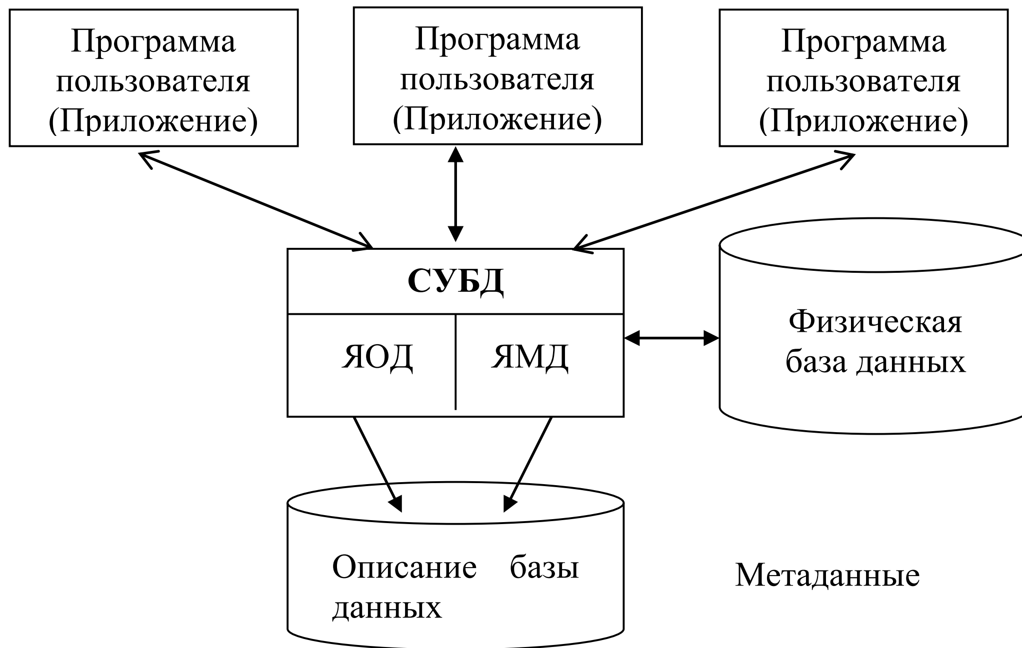



Рис. 1.4. Архитектура информационных систем, использующих концепцию баз данных

На данных, хранящихся в базе данных, основана вся информация, необходимая в работе любой организации. Но следует заметить, что данные, из которых состоит база данных, должны быть тщательно и логично организованы. Поэтому БД строится по определенным правилам и должна удовлетворять ряду требований, из которых основные:

- Минимальная избыточность. Каждый элемент данных вводится в БД один раз и хранится там в единственном экземпляре. При вводе данных СУБД осуществляет проверку на дублирование.
- Возможность актуализации. Данные, хранящиеся в БД, могут устаревать, при этом возникает необходимость ввести новые данные. Структура данных должна позволять включать новые и удалять устаревшие данные, а также вносить изменения в хранящиеся данные. При этом не должны меняться общая схема БД и программы пользователей.
- Обеспечение целостности данных. В системе возможно возникновение случайных ошибок в результате неосторожных действий пользователей, ошибок в программах и сбоев оборудования. СУБД должна обеспечивать защиту данных от разрушений и возможность восстановления искаженных данных.

- Безопасность и секретность. Пользователи должны работать только с теми данными, которые им необходимы. Данными, хранящимися в системе, не должны пользоваться лица, не имеющие на это права.
- Возможность обеспечения разнообразных запросов пользователей. Это требование является основным для БД.

**Система управления базами данных** (СУБД) – это совокупность программ и языковых средств, предназначенных для управления данными в базе данных, ведения базы данных и обеспечения взаимодействия её с прикладными программами [8].

Ведение базы данных – деятельность по обновлению, восстановлению и изменению структуры базы данных с целью обеспечения её целостности, сохранности и эффективности использования [8].

С помощью СУБД производятся запись данных в базу данных, их выборка по запросам пользователей и прикладных программ, обеспечивается защита данных от искажений и от несанкционированного доступа и т.п.

Каждой прикладной программе СУБД предоставляет интерфейс с базой данных и располагает средствами доступа к ней. Таким образом, СУБД играет центральную роль в функционировании информационной системы, так как обращение к базе данных возможно только через СУБД.

Концепция баз данных подразумевает рост совместного использования данных различными приложениями (прикладными программами) и сокращение избыточности хранения одних и тех же данных. Для организации управления этими процессами используется **описание базы данных**.

В описание базы данных хранится структура всех таблиц базы данных, информация об индексах, служащих для быстрого обращения к данным, правила проверки данных и много другой специальной информации о данных. Описание базы данных является частью современной базы данных.

Программы, с помощью которых пользователи работают с базой данных, называются **приложениями**. В общем случае с одной базой данных могут работать множество различных приложений. Например, если база данных моделирует некоторое предприятие, то для работы с ней может быть создано приложение, которое обслуживает

подсистему учета кадров, другое приложение может быть посвящено работе подсистемы расчета заработной платы сотрудников, третье приложение работает как подсистемы складского учета, четвертое приложение посвящено планированию производственного процесса. При рассмотрении приложений, работающих с одной базой данных, предполагается, что они могут работать параллельно и независимо друг от друга, и именно СУБД призвана обеспечить работу множества приложений с единой базой данных таким образом, чтобы каждое из них выполнялось корректно, но учитывало все изменения в базе данных, вносимые другими приложениями.

### 1.4. Функции СУБД

Как уже отмечалось выше, СУБД выполняет роль интерфейса между прикладными программами и базой данных, обеспечивающего их независимость.

С точки зрения пользователя, СУБД реализует функции:

1. Определение структуры создаваемой базы данных. Как правило, создание структуры базы данных происходит в режиме диалога. СУБД последовательно запрашивает у пользователя необходимые данные. В большинстве современных СУБД база данных представляется в виде совокупности таблиц. Рассматриваемая функция позволяет описать и создать в памяти компьютера структуру таблиц.

2. Предоставление пользователям возможности манипулирования данными (выборка необходимых данных, выполнение вычислений).

Реализация этих обеих функций в СУБД осуществляется на основе использования специального языка программирования, входящего в состав СУБД. В языках программирования СУБД принято выделять *язык описания данных* (ЯОД) и *язык манипулирования данными* (ЯМД).

ЯОД – это язык высокого уровня, предназначенный для описания структуры базы данных. С его помощью описываются типы данных, их размер и связи между собой. Это язык декларативного типа. В соответствии с полученным описанием СУБД сможет найти в базе требуемые данные, правильно преобразовать их и передать, например, в прикладную программу, которой они потребовались. При записи данных в базу данных СУБД определяет место в памяти ЭВМ, куда

их требуется поместить, преобразует к заданному виду и устанавливает необходимые связи.

Язык манипулирования данными представляет собой систему команд манипулирования данными и используется в прикладных программах для выполнения операций с базой данных (поиск, вставка, удаление  и обновление). При этом фактическая структура физической базы данных известна только СУБД.

Кроме, указанных выше основных функций, СУБД выполняют и другие функции:

- обеспечение логической целостности базы данных;
- обеспечение физической целостности базы данных;
- управление полномочиями пользователей на доступ к базе данных;
- обеспечение одновременного доступа к данным для нескольких пользователей.

*Обеспечение логической целостности базы данных.*

Основной целью реализации этой функции является повышение достоверности данных в базе данных.

*Целостность* – свойство базы данных, означающее, что она содержит полную и непротиворечивую информацию, адекватно отражающую предметную область.

Целостность базы данных обеспечивается ограничениями на значения элементов данных, поддержкой заданных принципов взаимосвязи данных (ссылочная целостность), управлением транзакциями.

Достоверность данных может быть нарушена при их вводе в БД или при неправомерных действиях процедур обработки данных, получающих и заносящих в БД неправильные данные. Для повышения достоверности данных в системе используются ограничения целостности, которые служат для определения неверных данных. Так, во всех современных СУБД проверяется соответствие вводимых данных их типу, описанному при создании структуры. Система не позволит ввести символ в поле числового типа, не позволит ввести недопустимую дату и т.п. В развитых системах ограничения целостности описывает программист, исходя из содержательного смысла задачи, и их проверка осуществляется при каждом обновлении данных. Более по-

дробно разные аспекты логической целостности базы данных будут рассматриваться в последующих разделах.

Данные в БД хранятся в разных таблицах, которые связаны между собой. Для поддержки заданных принципов взаимосвязи данных в СУБД существуют правила ссылочной целостности. Например, между двумя связанными таблицами может быть установлено правило: при удалении записей в родительской таблице автоматически осуществляется каскадное удаление всех записей из дочерней таблицы, связанных с удаляемой записью.

*Транзакцией* называется некоторая неделимая последовательность операций над данными БД, которая отслеживается СУБД. В состав транзакции может входить несколько команд изменения базы данных, но либо выполняются все эти команды, либо не выполняется ни одна. Если по каким-либо причинам (сбои и отказы оборудования, ошибки в программном обеспечении) транзакция остается незавершенной, то она отменяется, т.е. происходит откат транзакции.

*Обеспечение физической целостности базы данных*

СУБД, кроме ведения собственно базы данных, ведет также журнал транзакций. Ведение журнала изменений в БД (журнализация изменений) и создание резервных копий, дают возможность восстановить данные при наличии аппаратных сбоев, а также ошибок программного обеспечения.

*Управление полномочиями пользователей на доступ к базе данных.*

Разные пользователи могут иметь разные полномочия по работе с данными (некоторые данные должны быть недоступны; определенным пользователям не разрешается обновлять данные и т.п.). В СУБД предусматриваются механизмы разграничения полномочий доступа, основанные либо на принципах паролей, либо на описании полномочий.

*Обеспечение одновременного доступа к данным для нескольких пользователей.*

Достаточно часто может иметь место ситуация, когда несколько пользователей одновременно выполняют операцию обновления одних и тех же данных. Это может привести к нарушению логической целостности данных. Для повышения эффективности работы с базой

данных следует «изолировать» пользователей друг от друга. Для этого СУБД используют блокировки.

*Блокировкой* называется временное ограничение на выполнение некоторых операций обработки данных.

Существуют разные типы блокировок. *Блокировка* может быть наложена как на всю базу данных, так и на отдельную строку таблицы.

### 1.5. Виды архитектуры информационной системы

Архитектурой информационной системы называется концепция, согласно которой взаимодействуют компоненты информационной системы.

Любая информационная система включает в себя три компонента:

управление данными;

бизнес-логику;

пользовательский интерфейс.

Данные хранятся в базах данных, а управление ими осуществляется с помощью системы управления базами данных. Бизнес-логика определяет правила, по которым обрабатываются данные. Она реализуется набором процедур, написанных на различных языках программирования. Пользователь работает с интерфейсом, где логика работы ИС представлена в виде элементов управления – полей, кнопок, списков, таблиц и т.д.

Однако, эти три компонента в разных ИС взаимодействуют друг с другом различными способами.

Существуют следующие виды архитектур ИС:

централизованная;

файл-серверная;

клиент-серверная;

трехслойная.

#### 1.5.1. Централизованная архитектура

При использовании этой архитектуры база данных, СУБД и прикладная программа (приложение) располагаются на одном компьютере (рис. 1.5). Для такого способа организации не требуется поддержки сети и все сводится к автономной работе.

Многопользовательская технология работы обеспечивалась либо режимом мультипрограммирования (одновременно могли работать

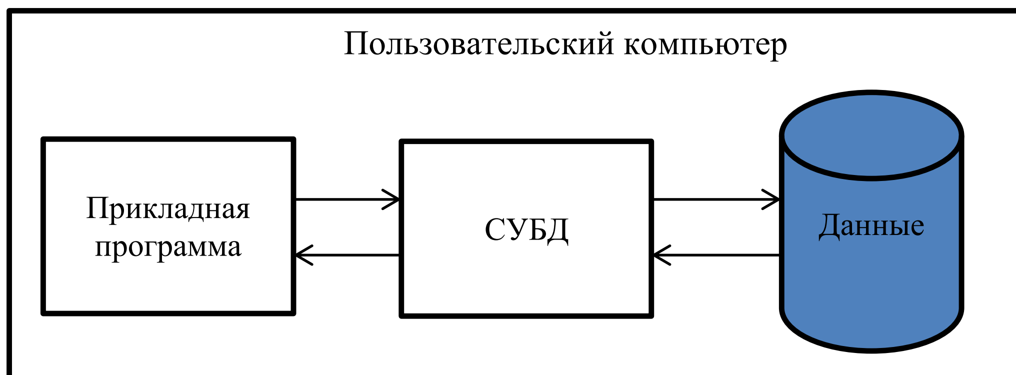


Рис. 1.5. Централизованная архитектура ИС

процессор и внешние устройства – например, пока в прикладной программе одного пользователя шло считывание данных из внешней памяти, программа другого пользователя обрабатывалась процессором), либо режимом разделения времени (пользователям по очереди выделялись кванты времени на выполнении их программ). Такая технология была распространена в период «господства» больших ЭВМ (IBM-370, ЕС-1045, ЕС-1060). Основным недостатком этой модели является резкое снижение производительности при увеличении числа пользователей.

### 1.5.2. Файл-серверная архитектура

Увеличение сложности задач, появление персональных компьютеров и локальных вычислительных сетей явились предпосылками появления новой архитектуры «файл-сервер» (рис. 1.6).

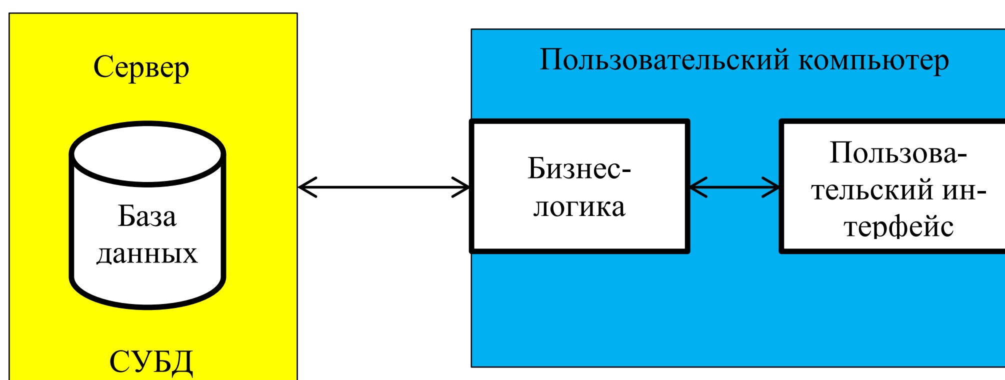


Рис. 1.6. Файл-серверная архитектура ИС

Эта архитектура баз данных с сетевым доступом предполагает назначение одного из компьютеров сети в качестве выделенного сервера, на котором будут храниться файлы базы данных. Компьютеры пользователей соединены с сервером сетью, поэтому доступ к данным, могут получить несколько пользователей одновременно. В соответствии с запросами пользователей файлы с файл-сервера передаются на рабочие станции пользователей, где и осуществляется основная часть обработки данных. Центральный сервер выполняет в основном только роль хранилища файлов, не участвуя в обработке самих данных.

Недостатки архитектуры с файловым сервером вытекают, главным образом, из того, что данные хранятся в одном месте, а обрабатываются в другом. Их нужно передавать по сети, что приводит к очень высоким нагрузкам на сеть и резкому снижению производительности приложения при увеличении числа одновременно работающих клиентов. Например, предположим, что в базе данных на сервере хранится список продаж товаров торговой фирмы. За год было осуществлено 20000 продаж. Пользователю нужно получить сумму продаж за каждый месяц. Для решения этой задачи пользователь должен запросить данные всех 20000 продаж с сервера по сети, после чего на пользовательском компьютере выполнится процедура, которая осуществит подсчет сумм продаж в каждом месяце. Результатом процедуры будет 12 строк. Таким образом, чтобы получить 12 строк придется передать по сети 20000 строк.

Вторым важным недостатком такой архитектуры является децентрализованное решение проблем целостности, из-за их несогласованной обработки разными пользователями.

### 1.5.3. Клиент-серверная архитектура

До определенного момента на СУБД возлагались лишь задачи хранения данных и организации доступа к ним. С развитием технологий в состав СУБД разработчики стали включать новый компонент – процедурный язык программирования. С его помощью в СУБД стало возможным создавать процедуры для обработки данных, которые можно вызывать повторно. Такие процедуры называются хранимыми процедурами. Наличие хранимых процедур дало возможность осуществлять некоторую часть обработки данных на сервере.

Таким образом, архитектура «клиент – сервер» разделяет функции приложения пользователя (называемого клиентом) и сервера (рис. 1.7). Приложение-клиент формирует запрос к серверу, на котором расположена БД, на структурном языке запросов SQL. При этом ресурсы клиентского компьютера не участвуют в физическом выполнении запроса; клиентский компьютер лишь отсылает запрос к серверной БД и получает результат, после чего интерпретирует его необходимым образом и представляет пользователю. Так как клиентскому приложению посылается результат выполнения запроса, по сети «путешествуют» только те данные, которые необходимы клиенту. В итоге снижается нагрузка на сеть. Поскольку выполнение запроса происходит там же, где хранятся данные (на сервере), нет необходимости в пересылке больших пакетов данных.

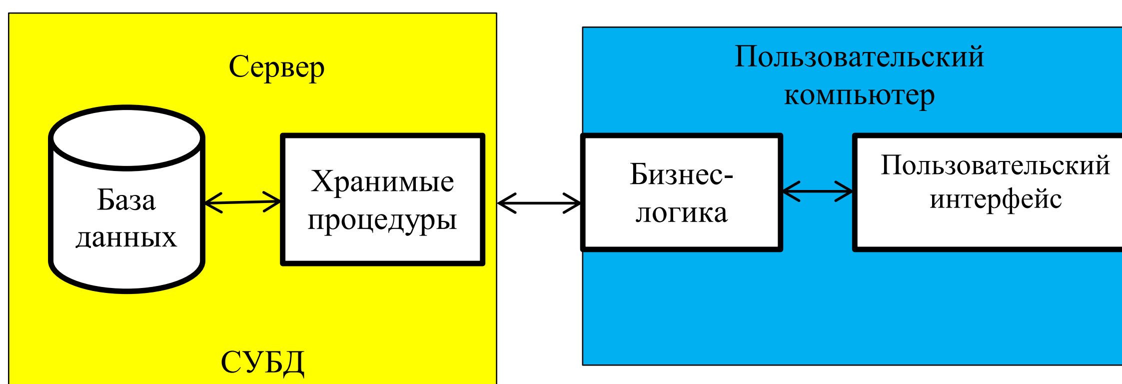



Рис. 1.7. Клиент-серверная архитектура ИС

Все это повышает быстродействие системы и снижает время ожидания результата запроса. Рассмотрим пример из параграфа 1.5.2 в условиях клиент-серверной архитектуры. Пользователь отправит на сервер запрос, который запустит процедуру. Процедура выполнится непосредственно на сервере. Она подсчитает сумму продаж за каждый месяц и отправит полученные 12 строк по сети на клиентский компьютер. Таким образом, произойдет существенная экономия трафика: вместо 20000 строк будет передано по сети всего 12.

При выполнении запросов сервером существенно повышается степень безопасности данных, поскольку правила целостности данных определяются в базе данных на сервере и являются едиными для всех приложений, использующих эту БД. Таким образом, исключает-

ся возможность определения противоречивых правил поддержания целостности.

Клиент-серверная архитектура позволяет разгрузить сеть и поддерживать непротиворечивость данных за счет их централизованной обработки. Однако, языки хранимых процедур не приспособлены для полноценной реализации бизнес-логики. Поэтому бизнес-логика в клиент-серверных ИС по-прежнему реализуется на клиентских компьютерах.

В архитектуре «клиент – сервер» работают так называемые «промышленные» СУБД.

Промышленными они называются из-за того, что именно СУБД этого класса могут обеспечить работу информационных систем масштаба среднего и крупного предприятия, организации, банка. К разряду промышленных СУБД принадлежат MS SQL Server, Oracle, Informix, Sybase, DB2, InterBase и ряд других.

Рассмотрим основные достоинства данной архитектуры по сравнению с архитектурой «файл-сервер»:

• Существенно уменьшается сетевой трафик.

• Уменьшается сложность клиентских приложений (большая часть нагрузки ложится на серверную часть), а, следовательно, снижаются требования к аппаратным мощностям клиентских компьютеров.

• Существенно повышается целостность и безопасность БД.

К числу недостатков можно отнести более высокие финансовые затраты на аппаратное и программное обеспечение, а также то, что большое количество клиентских компьютеров, расположенных в разных местах, вызывает определенные трудности со своевременным обновлением клиентских приложений на всех компьютерах-клиентах. Тем не менее, архитектура «клиент – сервер» хорошо зарекомендовала себя на практике, в настоящий момент существует и функционирует большое количество БД, построенных в соответствии с данной архитектурой.

### 1.5.4. Трехуровневая архитектура

Рассмотрев архитектуру «клиент – сервер», можно заключить, что она является 2-звенной: первое звено – клиентское приложение, второе звено – сервер БД + сама БД. Все недостатки клиент-

серверной архитектуры связаны с тем, что на клиентском компьютере лежит слишком большая нагрузка, которую можно было бы перенести на сервер. Поэтому дальнейшее развитие технологий двигалось в направлении переноса нагрузки с клиентских компьютеров на сервер. В дополнение к хранимым процедурам разработчики стали использовать серверные языки программирования. Это дало возможность создавать в ИС промежуточный уровень - сервер приложений.

В трехзвенной архитектуре вся бизнес-логика (деловая логика), ранее входившая в клиентские приложения, выделяется в отдельное звено, называемое сервером приложений (рис. 1.8). При этом клиентским приложениям остается лишь пользовательский интерфейс.

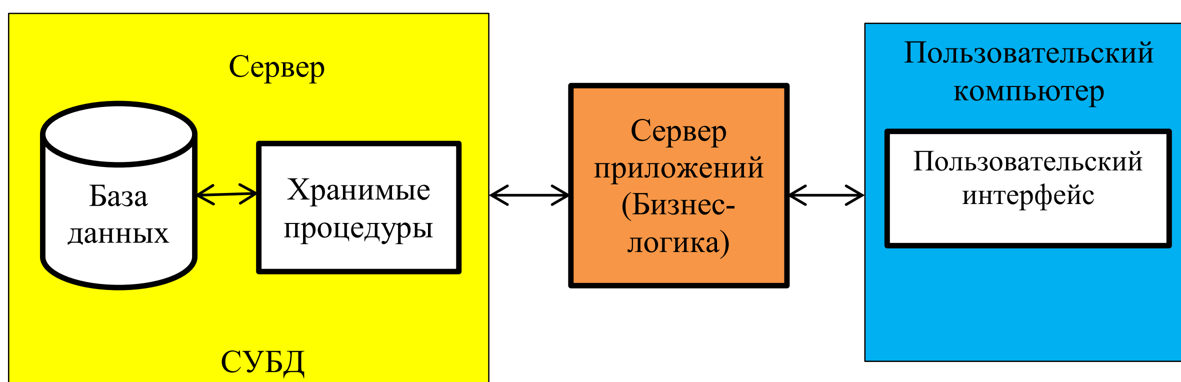



Рис. 1.8. Трехуровневая архитектура ИС

Теперь при изменении бизнес-логики более нет необходимости изменять клиентские приложения и обновлять их у всех пользователей. Кроме того, максимально снижаются требования к аппаратуре пользователей.

Недостатком трехзвенной архитектуру являются высокие расходы на администрирование и обслуживание серверной части.

## 1.6. Организация БД. Трехуровневое представление данных

Первая попытка создания стандартной терминологии и общей архитектуры представления данных была предпринята в 1971 г. по результатам конференции по языкам и системам данных CODASYL. При национальном институте стандартов США был создан комитет планирования стандартов и норм — ANSI/SPARC (ANSI — American National Standard Institute, SPARC — Standards Planning and

Requirements Committee). Этот комитет в 1975 г. предложил обобщенную трёхуровневую модель представления данных, включающую концептуальный, внешний и внутренний уровни (рис. 1.9).

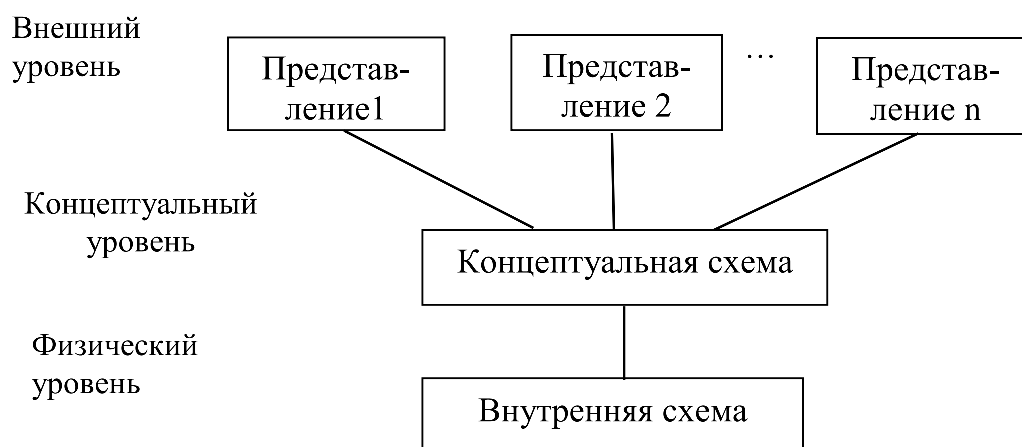



Рис. 1.9. Трехуровневая организация БД

Рассмотрим суть каждого уровня.

*Внешний уровень* отражает требования к данным с точки зрения конкретного пользователя (или приложения), т.е. лица, решающего узкую задачу на конкретном рабочем месте. Так как с базой данных работают несколько пользователей, то внешний уровень состоит из нескольких различных внешних представлений базы данных. Каждый пользователь видит и обрабатывает только те данные, которые ему необходимы. Другие данные, которые ему неинтересны, также могут быть представлены в базе данных, но пользователь может даже не подозревать об их существовании. Например, система распределения работ использует сведения о квалификации сотрудника, но ее не интересуют сведения об окладе, домашнем адресе и телефоне сотрудника, и наоборот, именно эти сведения используются в подсистеме отдела кадров. Описание представления данных для группы пользователей называется *внешней схемой.*

Создание базы данных предполагает интеграцию данных, предназначенных для решения нескольких прикладных задач разных пользователей. Соответственно, при интеграции данных должны учитываться требования к данным каждого пользователя, основанные на его представлении о данных и связях между ними. Далее эти требова-

ния должны обобщаться в единое представление, которое и будет служить основой для построения единой базы данных.

*Концептуальный уровень* отражает обобщенное представление всех пользователей о данных, которое является моделью предметной области, создаваемой база данных. Описание этого представления называется *концептуальной схемой* или *схемой БД*. Концептуальная схема представляет информационное описание предметной области с учетом логических взаимосвязей, поэтому её еще называют инфологической (информационно-логической) моделью.

Концептуальный уровень поддерживает каждое внешнее представление, в том смысле, что любые доступные пользователю данные должны содержаться (или могут быть вычислены) на этом уровне. Однако этот уровень не содержит никаких сведений о методах хранения данных.

Выделение концептуального уровня позволило разработать аппарат централизованного управления базой данных.

*Внутренний* уровень описывает физическую реализацию базы данных и предназначен для достижения оптимальной производительности, обеспечения экономного использования дискового пространства, организации мероприятий по защите данных. Он содержит детальное описание структур данных и физической организации файлов с данными, описание вспомогательных структур (индексов), используемых для ускорения поиска, сведения о распределении дискового пространства для хранения данных и индексов и т.д. Описание БД на внутреннем уровне называется *внутренней схемой* или *схемой хранения.*

Как показывает изучение трехуровневой архитектуры БД, концептуальная схема является самым важным уровнем представления базы данных. Она поддерживает все внешние представления, а сама поддерживается средствами внутренней схемы. Внутренняя схема является всего лишь физическим воплощением концептуальной схемы. Именно концептуальная схема призвана быть полным и точным представлением требований к данным в рамках некоторой предметной области.

Основным назначением трехуровневой архитектуры является обеспечение **независимости от данных,** которая означает, что изменения на нижних уровнях никак не влияют на верхние уровни. Раз-

личают два типа независимости от данных: **логическую и физическую.**

Логическая независимость от данных означает полную защищенность внешних представлений от изменений, вносимых на концептуальном уровне. Ясно, что пользователи, для которых эти изменения предназначались, должны знать о них, но очень важно, чтобы другие пользователи даже не подозревали об этом. Таким образом, логическая независимость предполагает возможность изменения одного приложения без корректировки других приложений, работающих с этой же базой данных.

Физическая независимость от данных означает защищенность концептуальной схемы от изменений способов организации базы данных в памяти ЭВМ, вносимых на внутреннем уровне. Таким образом, физическая независимость предполагает возможность переноса хранимой информации с одних носителей на другие при сохранении работоспособности всех приложений, работающих с данной базой данных.

Таким образом, трехуровневая архитектура позволяет добиться следующего:

- Каждый пользователь имеет возможность обращаться к одним и тем же данным, используя свое собственное представление о них.
- Каждый пользователь имеет возможность изменять свое представление о данных, причем это изменение не оказывает влияния на других пользователей.
- Пользователи не зависят от особенностей хранения в ней данных.
- Администратор базы данных имеет возможность изменять структуру хранения данных в базе, не оказывая влияния на пользовательские представления.
- Внутренняя структура базы данных не зависит от переноса базы данных на новое устройство памяти.
- Администратор базы данных имеет возможность изменять концептуальную структуру базы данных без какого-либо влияния на всех пользователей.

## 1.7. Понятие модели данных и виды моделей

Для формирования представления о данных, их составе и использовании в конкретных условиях служат различные модели данных. Поэтому центральным понятием в области баз данных является понятие модели.

*Модель* – это некоторая абстракция представления предметной области, отражающая только избранные детали.

*Предметная область* – это область применения конкретной БД.

Таким образом, при решении конкретных задач реальная действительность воспроизводится с существенными ограничениями, зависящими от области деятельности, поставленных целей и мощности вычислительных средств, т.е. в БД находят отражение только те факты о предметной области, которые необходимы для функционирования информационной системы, в состав которой входит БД.

В соответствии с рассмотренной ранее трехуровневой архитектурой мы сталкиваемся с понятием модели данных по отношению к каждому уровню. Модели данных, которые используются в теории баз данных, приведены на рис. 1.10.

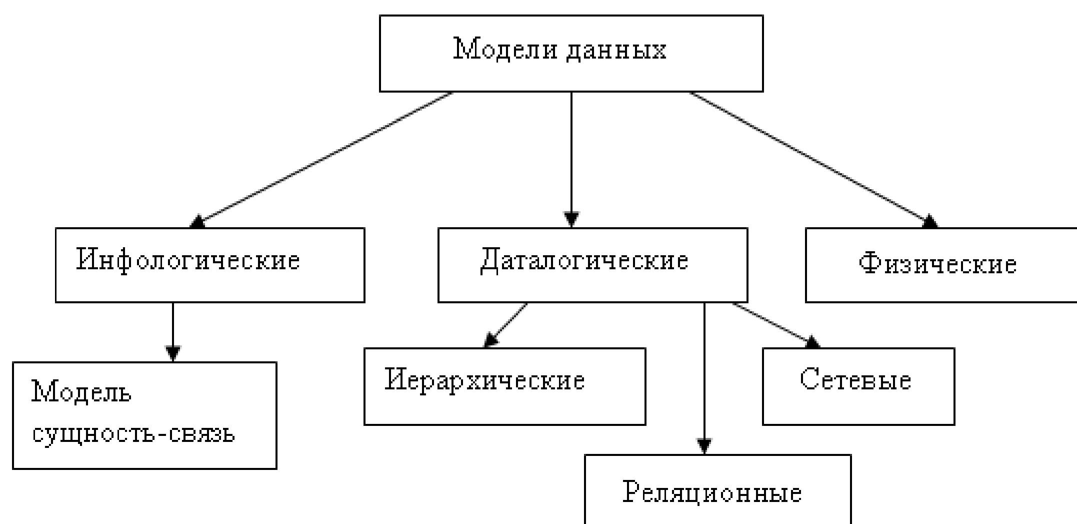



Рис. 1.10. Модели данных

Модель данных на внутреннем уровне оперирует категориями, касающимися организации внешней памяти и структур хранения, используемых в данной операционной среде. В настоящий момент в качестве внутренних моделей используются различные методы размещения данных, основанные на файловых структурах: это организация

файлов прямого и последовательного доступа, индексных файлов и инвертированных файлов, файлов, использующих различные методы хэширования, взаимосвязанных файлов. Кроме того, современные СУБД широко используют страничную организацию данных.

Модель данных, используемая на концептуальном уровне, должна выражать информацию о предметной области в виде, независимом от используемой СУБД. Эти модели называются *инфологическими,* или *концептуальными.*

Данная модель представляет собой набор конструкций, которые описывают структуру БД в виде совокупности сущностей, атрибутов и связей.

При разработке инфологической модели базы данных должны быть решены следующие вопросы:

1) о каких объектах или явлениях реального мира требуется накапливать и обрабатывать информацию в БД;
2) какие их основные характеристики;
3) как связаны между собой эти объекты.

Таким образом, при создании инфологической модели выделяется часть реального мира, определяющая информационные потребности БД, т.е. ее предметную область.

Модели *внешнего уровня* называются подсхемами и используют те же абстрактные категории, что и концептуальные модели данных.

Следует особо отметить, что концептуальная модель данных не зависит от конкретной СУБД или аппаратной платформы, которая используется для реализации БД. Поэтому в дальнейшем в процессе проектирования баз данных необходимо выполнить преобразование концептуальной модели в модель данных конкретной СУБД. Такие модели называются *даталогическими*.

**Даталогические** модели описывают логическую структуру данных, хранимых в базе, и поддерживаются конкретной СУБД.

Даталогические модели делятся на фактографические и документальные. Документальные модели служат для описания слабоструктурированной информации в виде текстов на естественном языке.

К числу классических относятся следующие фактографические модели данных:

• иерархическая,

• сетевая,

• реляционная.

Кроме того, в последние годы появились и стали более активно внедряться на практике многомерная и объектно-ориентированная модели данных.

**Вопросы для самопроверки**

1. Какие принципы обработки данных Вы знаете?
2. Какие недостатки имеет схема обработки данных с использованием файловых систем?
3. Укажите достоинства схемы обработки данных с использованием баз данных.
4. Из чего состоит логическая запись?
5. Из каких составляющих элементов состоит логический файл?
6. Дайте определение термину информация. Приведите примеры.
7. Что обусловило появление систем управления базами данных?
8. Из каких основных компонентов состоит информационная система, использующая базу данных?
9. Что такое база данных?
10. Каковы основные функции системы управления базой данных?
11. Что такое архитектура файл-сервер?
12. Где расположены программы пользователя и программы СУБД в архитектуре клиент-сервер?
13. Что отличает трехзвенную архитектуру от архитектуры клиент-сервер?
14. Дайте определение внешнему, концептуальному и внутреннему представлению данных.
15. Что дает логическая и физическая независимость данных?
16. Какие виды моделей Вы знаете?

# Глава 2. ДАТАЛОГИЧЕСКИЕ МОДЕЛИ ДАННЫХ

## 2.1. Компоненты даталогических моделей данных

Каждая СУБД поддерживает определенные виды и типы данных, а также средства представления связей между данными, составляющими модель данных. Хранимые в базе данные должны иметь определенную логическую структуру, которая описывается некоторой даталогической моделью данных, поддерживаемой СУБД.

Как уже отмечалось, даталогические модели делятся на документальные и фактографические. В этой главе речь будет идти о фактографических моделях, служащих для описания хорошо структурированных данных.

В литературе и в обиходной речи иногда понятия «модель данных» и «модель базы данных» («схема базы данных») используются как синонимы. Однако это неверно, на что указывают многие авторитетные специалисты, в том числе К. Дж. Дейт, М. Р. Когаловский, С. Д. Кузнецов. Модель данных есть *теория*, или *инструмент моделирования*, в то время как модель базы данных (схема базы данных) есть *результат моделирования*. По выражению К. Дейта соотношение между этими понятиями аналогично соотношению между языком программирования и конкретной программой на этом языке [1].

В соответствии с [8], **модель данных** – это совокупность правил порождения структур данных в базе данных, операций над ними, а также ограничений целостности, определяющих допустимые связи и значения данных, последовательность их изменения.

Любая модель данных должна содержать три компонента:

1. Структуру данных
2. Набор допустимых операций, выполняемых на структуре данных.
3. Ограничения целостности.

*Структуры данных*

Все данные размещаются в некоторой структуре для обеспечения информационных потребностей пользователей. Здесь можно провести аналогию с языками программирования, в которых тоже есть типы структур данных, такие как массивы, записи и т.д.

Определим основные структуры даталогических моделей данных.

**Элемент данных** – наименьшая поименованная единица данных, к которой СУБД может обращаться непосредственно и с помощью которой выполняется построение всех остальных структур. Для каждого элемента данных должен быть определён его тип. В разных СУБД могут использоваться разные типы данных, наиболее распространенными из которых являются следующие: целый (integer), символьный (char), дата (date) и т.д.

**Запись –** поименованная совокупность полей.

**Файл –** поименованная совокупность экземпляров записей одного типа.

*Набор допустимых операций*

Такими операциями являются: создание и модификация структур данных, внесение новых данных, удаление и модификация существующих данных, поиск данных по различным условиям. Модель данных предполагает, как минимум, наличие языка определения данных (DDL – Data Definition Language), описывающего структуру их хранения, и языка манипулирования данными (DML – Data Manipulation Language), включающего операции извлечения и модификации данных.

*Ограничения целостности*

Ограничения целостности – механизм поддержания соответствия данных предметной области на основе формально описанных правил. Правила целостности определяются типом данных и предметной областью. Например, значение поля *Количество товара* является целым числом. А ограничения предметной области таковы, что это число не может быть меньше нуля.

В процессе исторического развития в СУБД использовались следующие модели данных: *иерархическая, сетевая, реляционная.* Эти модели отличаются между собой по способу установления связей между данными.

## 2.2. Иерархическая модель данных

Первая информационная система, использующая базы данных, появилась в середине шестидесятых годов и была основана на иерар-

хической модели. *Иерархическая модель* – модель данных, в которой связи между данными имеют вид иерархий. Такую модель можно описать с помощью упорядоченного графа (или дерева) с дугами-связями и узлами-элементами данных. Иерархическая структура предполагала неравноправие между данными - одни жестко подчинены другим.

На рис. 2.1 показан пример иерархической модели данных.

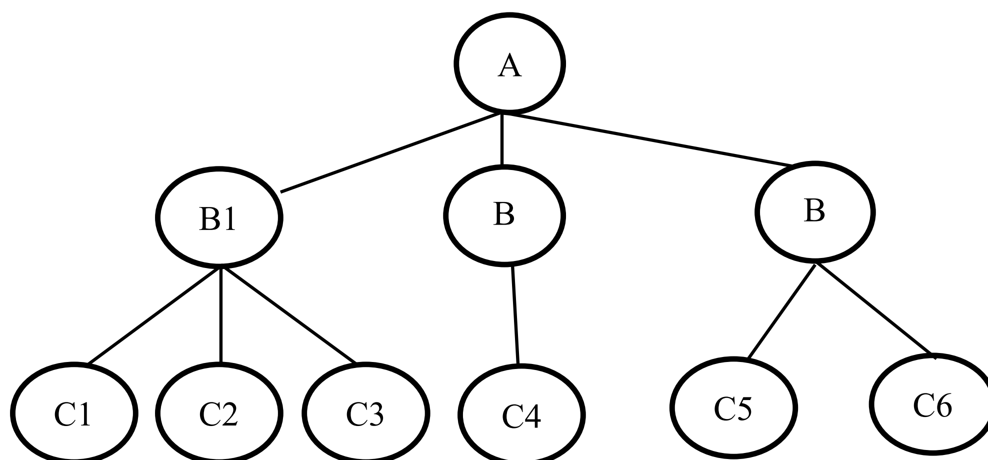



Рис. 2.1. Пример иерархической модели данных

Дерево представляет собой иерархию элементов, называемых узлами. В иерархической модели имеется *корневой узел* или просто корень дерева. Корень находится на самом верхнем уровне и не имеет узлов, стоящих выше него. У одного дерева может быть только один корень. Остальные узлы называются *порожденными* и связаны между собой следующим образом: каждый узел имеет *исходный*, находящийся на более высоком уровне. Если каждый узел может быть связан только с одним исходным узлом, то на последующем уровне он может иметь один, два и больше узлов либо не иметь ни одного. В последнем случае узлы, не имеющие порожденных узлов, называются *листьями.* Доступ к порожденным узлам возможен только через исходный узел, поэтому существует только один путь доступа к каждому узлу.

Появление иерархической модели связано с тем, что в реальном мире очень многие связи соответствуют иерархии, когда один объект выступает как родительский, а с ним может быть связано множество подчиненных объектов.

Основными информационными единицами в иерархической модели являются: *атрибут, сегмент* и *групповое отношение.*

*Атрибут данных* (элемент данных, поле) определяется как минимальная, неделимая единица данных, доступная пользователю с помощью СУБД. Обычно каждому элементу при описании базы данных присваивается уникальное имя. По этому имени к нему обращаются при обработке. Элемент данных также часто называют полем.

*Сегмент* в терминологии Американской Ассоциации по базам данных DBTG (Data Base Task Group) называется *записью,* при этом в рамках иерархической модели определяются два понятия: *тип сегмента* (*тип записи)* и *экземпляр сегмента* (*экземпляр записи).* Тип сегмента определяется составом ее атрибутов. Экземпляр записи - конкретная запись с конкретным значением элементов. Каждый тип сегмента в рамках иерархической модели образует некоторый набор однородных записей. Использование записей позволяет за одно обращение к базе получить некоторую логически связанную совокупность данных. Именно записи изменяются, добавляются и удаляются.

Групповое отношение - иерархическое отношение между записями двух типов. Родительская запись (владелец группового отношения) называется исходной записью, а дочерние записи (члены группового отношения) - подчиненными. Иерархическая база данных может хранить только такие древовидные структуры.

В иерархической модели сегменты объединяются в ориентированный древовидный граф - дерево. При этом полагают, что направленные ребра графа отражают иерархические связи между сегментами.

Корневая запись каждого дерева обязательно должна содержать ключ с уникальным значением. Ключи некорневых записей должны иметь уникальное значение только в рамках группового отношения. Каждая запись идентифицируется полным сцепленным ключом, под которым понимается совокупность ключей всех записей от корневой по иерархическому пути.

Схема иерархической БД представляет собой совокупность отдельных деревьев, содержащих экземпляры типа «запись» (записи). Поля записей хранят значения разных типов данных, составляющие основное содержание БД. Обход всех элементов иерархической БД обычно производится сверху вниз и слева направо.

В качестве примера рассмотрим модель данных организации (рис. 2.2). Организация состоит из отделов, в которых работают сотрудники, и филиалов. В каждом отделе может работать несколько сотрудников, но сотрудник не может работать более чем в одном отделе.

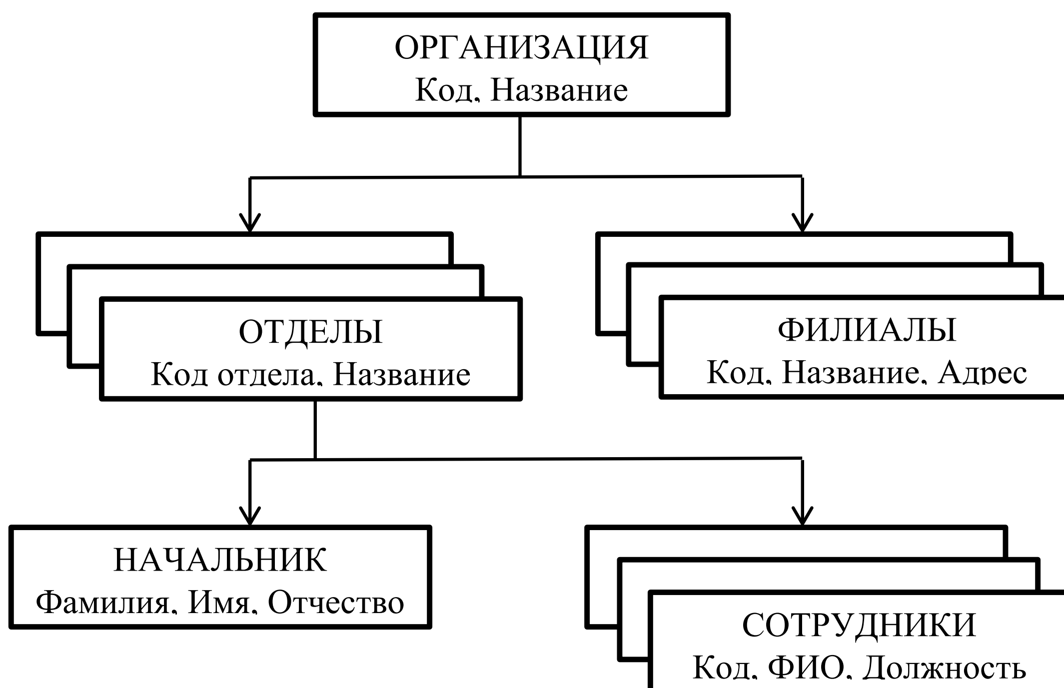



Рис. 2.2. Иерархическая модель данных организации

К основным операциям манипулирования иерархически организованными данными относятся следующие:

- **Добавить** в базу данных новую запись. Для корневой записи обязательно формирование значения ключа.
- **Изменить** значение данных предварительно извлеченной записи. Ключевые данные не должны подвергаться изменениям.
- **Удалить** некоторую запись и все подчиненные ей записи.
- **Извлечь** корневую запись по ключевому значению, допускается также последовательный просмотр корневых записей.
- **Извлечь** следующую запись (следующая запись извлекается в порядке левостороннего *обхода дерева*).

В соответствии с определением типа «дерево», можно заключить, что между предками и потомками автоматически поддерживается контроль целостности связей. Основное правило контроля целостности формулируется следующим образом: потомок не может существовать без родителя, а у некоторых родителей может не быть

потомков. Механизмы поддержания целостности связей между записями различных деревьев отсутствуют.

К достоинствам иерархической модели данных относятся:

- эффективное использование памяти ЭВМ;
- высокая скорость выполнения основных операций над данными;
- удобство работы с иерархически упорядоченной информацией.

К основным недостаткам такого вида модели можно отнести следующие:

- громоздкость для обработки информации с достаточно сложными логическими связями;
- иерархия в значительной степени усложняет операции включения информации о новых объектах в базе данных и удаления устаревших. Так при удалении экземпляров исходного узла автоматически удаляются все экземпляры порожденных узлов.
- исключительно навигационный принцип доступа к данным.

### 2.3. Сетевая модель данных

Прежде чем перейти к рассмотрению сетевой модели данных рассмотрим следующий пример: предприятие со сборочным характером производства выпускает множество изделий, которые включают в себя детали-сборочные единицы (ДСЕ), которые можно разделить на две группы:

- уникальные ДСЕ, входящие только в одно изделие или только в одну сборочную единицу;
- ДСЕ общего применения, которые могут входить в несколько изделий или других ДСЕ (например, гайки, винты, шайбы и т.п.).

В этом случае взаимосвязь между объектами показана на рис. 2.3, откуда видно, что данная схема не является иерархической, так как порожденный элемент ДСЕ4 имеет два исхода.

Такие отношения между объектами, в которых порожденный элемент имеет более одного исходного, описываются в виде *сетевой структуры*. Отличительная черта сетевой структуры от иерархической заключается в том, что любой элемент в сетевой структуре может быть связан с любым другим элементом.

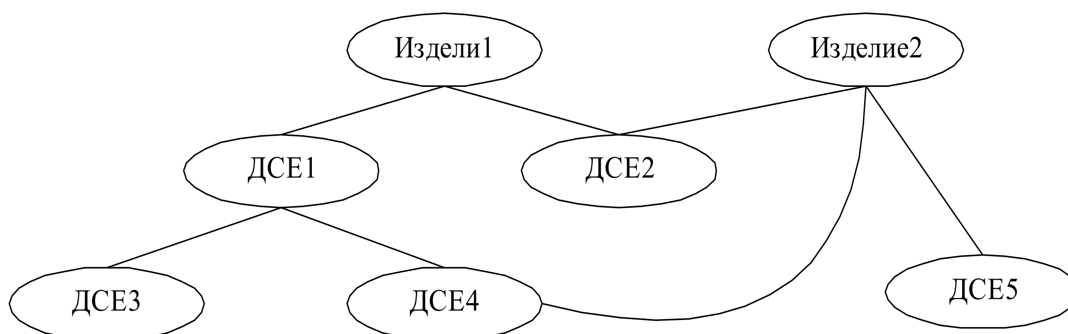



Рис. 2.3. Пример сетевой модели данных

Таким образом, сетевая модель данных позволяет отображать разнообразные взаимосвязи элементов данных в виде произвольного графа, обобщая тем самым иерархическую модель данных. Стандарт сетевой модели впервые был определен в 1971 году организацией CODASYL (Conference of Data System Languages), которая определила базовые понятия модели и формальный язык описания.

Сетевая модель данных определяется в тех же терминах, что и иерархическая. Она состоит из набора записей и набора соответствующих связей. На формирование связи особых ограничений не накладывается. Если в иерархических структурах запись-потомок могла иметь только одну запись-предка, то в сетевой модели данных запись-потомок может иметь произвольное число записей-предков (сводных родителей).

Реализация групповых отношений (связей) в сетевой модели осуществляется с использованием специально вводимых дополнительных полей - указателей (адресов связи или ссылок), которые устанавливают связь между владельцем и членом группового отношения. Запись может состоять в отношениях разных типов (один-к одному 1:1, один-ко-многим 1:*M*, много-ко многим *M*:*M*). Заметим, что если один из вариантов установления связи 1:1 очевиден (в запись – владелец отношения включается дополнительное поле – указатель на запись – член отношения), то возможность представления связей 1:*M* и *M*:*M* таким же образом весьма проблематична. Поэтому наиболее распространенным способом организации связей в сетевых СУБД является введение дополнительного типа записей (и соответственно, дополнительного файла), полями которых являются указатели.

К числу важнейших операций манипулирования данными баз сетевого типа можно отнести следующие:

• поиск записи в БД;
• переход от предка к первому потомку;
• переход от потомка к предку;
• создание новой записи;
• удаление текущей записи;
• обновление текущей записи;
• включение записи в связь;
• исключение записи из связи;
• изменение связей и т. д.

В сравнении с иерархической моделью сетевая модель предоставляет большие возможности в смысле допустимости образования произвольных связей.

Основной недостаток сетевой модели состоит в ее сложности и жесткости схемы БД. Прикладной программист должен детально знать логическую структуру базы данных, поскольку ему необходимо осуществлять навигацию среди различных экземпляров и записей.

Другим недостатком является ослабленный контроль за целостностью связей вследствие допустимости установления произвольных связей между записями.

Пользуясь популярной в технологической сфере терминологией, базы данных, использующие иерархические и сетевые модели данных можно отнести к базам данных первого поколения. Основным способом работы с базами данных этих типов являлся так называемый «навигационный» способ, заключающийся в перемещении по графу, описывающему структуру базы данных, и низкоуровневой поэлементной обработке содержащихся в узлах этого графа структур данных.

Основным недостатком баз данных первого поколения являлась сложность структуры данных даже для достаточно простых систем, что приводило к сложности прикладных программ обработки данных, требующих детального описания навигационных путей к данным. Кроме того, наличие в программах навигационных путей крайне затрудняло модификацию структуры данных, поскольку требовало модификации всех соответствующих программ обработки. Это, в свою очередь, практически не давало возможности разделить функции проектировщика данных и проектировщика прикладных программ, что являлось заметным препятствием при реализации больших проектов.

Поворотным пунктом в теории и практике баз данных явился переход к реляционным базам данных.

## 2.4. Реляционная модель данных

Отмеченные выше недостатки иерархических и сетевых моделей данных обусловлены, главным образом, тем, что в этих моделях записи связаны между собой с помощью указателей.

*Указатель* – это физический адрес, обозначающий место хранения записи на диске. Достоинство указателей состоит в том, что они позволяют быстро извлекать данные, которые связаны между собой. Недостаток заключается в том, что эти связи должны быть определены при проектировании базы данных. Извлечь данные на основе других связей сложно, если вообще возможно. В то же время введение новых связей приводило к изменению всей модели данных и базы данных в целом.

В 1970 году Э. Кодд (E. Codd) опубликовал статью, в которой выдвинул идею о том, что данные нужно связывать не физическими указателями, а в соответствии с их внутренними логическими взаимоотношениями. Таким образом, пользователи смогут комбинировать данные из разных источников на основе логических зависимостей между ними. Это открыло новые возможности для информационных систем, поскольку запросы к базам данных перестали быть ограничены физическими указателями.

### 2.4.1. Формальное описание реляционной модели данных

В основе реляционной модели лежит математическое понятие теоретико-множественное отношения. Поэтому модель и получила название *реляционной* (от английского *relation* — отношение).

Рассмотрим основные понятия теории отношений.

Элементы отношения называют *кортежами* (или *записями*). Каждый кортеж отношения соответствует одному экземпляру объекта, информацию о котором требуется хранить в базе данных. Элементы кортежа принято называть *атрибутами* (или *полями*).

В основе определения отношения лежит понятие домен.

Пусть $A_1, A_2, \ldots, A_n$ имена атрибутов. Каждому имени атрибута $A_i$ соответствует допустимое множество значений, которые может принимать атрибут $A_i$. Это множество значений $D_i$ называется доменом атрибута $A_i$, $i = \overline{1, n}$ . Каждый домен образует значения одного просто-

го типа данных, например, числовые, символьные и др.. Домен может задаваться перечислением элементов, указанием диапазона значений, функцией и т.д.

*Декартовым произведением* $k$ доменов $D_1$, $D_2$, $D_3$,. . . $D_k$ называется множество всех кортежей длины $k$, т.е. состоящих из $k$ элементов – по одному из каждого домена и определяется следующим образом

$$D = D_1 \times D_2 \times D_3 \times \ldots \times D_k$$

Например, если $D_1 = \{A, 1\}$, $D_2 = \{B, C\}$, $D_3 = \{2,3,D\}$, то $k = 3$ и декартово произведение равно

$D = \mathrm{D}_1 \times \mathrm{D}_2 \times \mathrm{D}_3 = \{(A, B, 2), (A, B, 3), (A, B, D), (A, C, 2), (A, C, 3), (A, C, D), (1, B, 2), (1, B, 3), (1, B, D), (1, C, 2), (1, C, 3), (1, C, D)\}$.

Как видно из примера, декартово произведение позволяет получить все возможные комбинации элементов исходных множеств – элементов рассматриваемых доменов.

*Отношением* называют некоторое подмножество декартова произведения одного или более доменов.

Например, если построить произведение трёх доменов Должности ('директор', 'бухгалтер', 'водитель', 'продавец'), Оклады ( 20000 <= x <= 80000), Надбавки (1.1, 1.2, 1.3), то мы получим 4*60001*3=720012 комбинаций. Но реально отношение «Штатное расписание» содержит по одной строке на каждую должность, т.е. является именно подмножеством декартова произведения доменов.

Схемой отношения $R$ называется перечень имен атрибутов отношения с указанием доменов этих атрибутов и обозначается

$$R\ (A_1, A_2, \ldots, A_n);\ \{A_i\} \subseteq D_i\,,$$

где $\{D_i\}$ – множество значений, принимаемых атрибутом $A_i$ ( $i = 1, n$ ).

Отметим следующие свойства отношения:

1. Отношение имеет имя, которое отличается от имен всех других отношений.

2. Каждый атрибут имеет уникальное имя.

3. В отношении нет повторяющихся кортежей.

4. Значения всех атрибутов являются атомарными (неделимыми). Это следует из определения домена как множества значений простого типа данных, т.е. среди значений домена не могут содержаться множества.

5. Порядок рассмотрения атрибутов в схеме отношения не имеет значения, т.к. для ссылки на значение атрибута в кортеже отношения всегда используется имя атрибута.

6. Порядок рассмотрения кортежей в отношении не имеет значения, т.к. отношение представляет собой множество кортежей, а элементы множества, по определению теории множеств, неупорядоченны.

Значение атрибута может быть пустым, т.е. иметь значение NULL. Значение NULL приписывается атрибуту в кортеже, если значение атрибута неизвестно. Значение NULL не привязано к определённому типу данных, т.е. может назначаться данным любых типов.

Отношение имеет простую графическую интерпретацию, оно может быть представлено в виде таблицы, столбцы которой соответствуют атрибутам, а каждая строка есть кортеж. На рис. 2.4 приведен пример представления отношения СОТРУДНИК.

Отношение СОТРУДНИК (таблица)
Атрибут Отдел (заголовок столбца)
Схема отношения (строка заголовков)
Кортеж (строка)

| ФИО | Отдел | Должность | Д_рождения |
|---|---|---|---|
| Иванов И.И. | 002 | Начальник | 27.09.51 |
| Петров П.П. | 001 | Заместитель | 15.04.55 |
| Сидоров И.П. | 002 | Инженер | 13.01.70 |

Значение атрибута (значение поля в записи)

Рис. 2.4. Представление отношения СОТРУДНИК

В дальнейшем термины «отношение» и «таблица» будут использоваться как синонимы.

Поскольку в рамках одной таблицы не удается описать более сложные логические структуры данных из предметной области, применяют *связывание* таблиц.

Таким образом, реляционная база данных с логической точки зрения может быть представлена множеством двумерных таблиц самого различного предметного наполнения.

Совокупность схем отношений, используемых для представления предметной области, называется схемой реляционной базы данных.

*Реляционная схема базы данных* - список, в котором даются имена реляционных таблиц с перечислением их атрибутов (ключи подчеркнуты) и определений внешних ключей.

В дальнейшем каждую таблицу будем описывать следующим образом:

<ИМЯ ТАБЛИЦЫ> (<СПИСОК АТРИБУТОВ>),

где список атрибутов – это множество неповторяющихся имен атрибутов, в котором ключевые атрибуты будут выделены подчеркиванием. Например,

СОТРУДНИК( ФИО, Отдел, Должность, Д_рождения)

Физическое размещение данных в реляционных базах на внешних носителях легко осуществляется с помощью обычных файлов.

### 2.4.2. Ключи отношения

Одно из требований к отношению (таблицы) – это наличие в отношении ключа.

*Ключ* – это минимальный набор атрибутов отношения, однозначно идентифицирующий каждый из его кортежей. Минимальность означает, что исключение из набора любого атрибута не позволяет однозначно найти требуемый кортеж по значениям оставшихся атрибутов.

Каждое отношение обладает хотя бы одним ключом. Это гарантируется тем, что в отношении нет повторяющихся кортежей, а это значит, что, по крайней мере, вся совокупность атрибутов обладает свойством однозначной идентификации кортежей отношения.

Ключ, содержащий более одного атрибута, называется *составным ключом*.

Очень часто отношение может содержать несколько ключей, которые называют *возможными (потенциальными) ключами*. Один из них принимается за *первичный ключ*.

При выборе первичного ключа следует отдавать предпочтение несоставным ключам.

Так, для идентификации студента можно использовать либо уникальный номер зачетной книжки, либо набор из фамилии, имени, отчества, номера группы и может быть дополнительных атрибутов, так как не исключено появление в группе двух студентов с одинаковыми фамилиями, именами и отчествами.

Не допускается, чтобы первичный ключ (любой атрибут, участвующий в первичном ключе) принимал значение NULL. Иначе возникнет противоречивая ситуация, когда появится не обладающий индивидуальностью кортеж. По тем же причинам необходимо обеспечить уникальность *первичного ключа*.

Нецелесообразно также использовать ключи с длинными текстовыми значениями (предпочтительнее использовать целочисленные атрибуты).

Ключи обычно используют для достижения следующих целей:

1) исключения дублирования значений в ключевых атрибутах (остальные атрибуты в расчет не принимаются);

2) упорядочения кортежей. Возможно упорядочение по возрастанию или убыванию значений всех ключевых атрибутов;

3) ускорения работы с кортежами отношения;

4) организации связывания отношений.

*Внешние ключи* предназначены для установления связей между отношениями.

Столбец одной таблицы, значения в котором совпадают со значениями столбца, являющегося первичным ключом другой таблицы, называется внешним ключом.

Например, имеются две таблицы СТУДЕНТ(ФИО, Номер_зач_кн, Группа) и ЭКЗАМЕН(Номер_зач_кн, Дисциплина, Оценка), которые моделируют сдачу студентами экзаменационной сессии. В первой таблице первичным ключом является атрибут Номер_зач_кн. Так как каждый студент сдает несколько дисциплин в процессе сессии, то первичным ключом в таблице ЭКЗАМЕН будет набор атрибутов Номер_зач_кн, Дисциплина, т.е. ключ является составным. Оба отношения связаны через атрибуты Номер_зач_кн, которые представляют собой внешние ключи.

Внешний ключ, как и первичный, тоже может быть составным, если он ссылается на составной первичный ключ.

Если таблица связана с несколькими другими таблицами, она может иметь несколько внешних ключей.

Реляционная модель накладывает на внешние ключи ограничение для обеспечения целостности данных, называемое *ссылочной целостностью*. Это означает, что каждому значению внешнего ключа должны соответствовать строки в связываемых отношениях.

Кортежи отношения часто могут быть представлены внутренними номерами, которые не имеют смысла вне системы. Внутренний номер часто называют *суррогатным ключом*. Суррогатные ключи целесообразно использовать в следующих случаях:

- естественный атрибут должен играть роль ключа, но не обладает необходимым свойством уникальности;
- выбранный в качестве внешнего ключа атрибут имеет большую длину (например, длинная строка символов). Чтобы избежать его многократного дублирования, целесообразно присвоить ему код;
- естественный атрибут может менять свое значение со временем, например, название предприятия или фамилия сотрудника.

### 2.4.3. Типы связи таблиц

Как уже говорилось ранее, реляционная модель представляет базу данных в виде множества взаимосвязанных таблиц (отношений). Между таблицами могут устанавливаться бинарные (между двумя таблицами), тернарные (между тремя таблицами) и, в общем случае, n-арные связи. Рассмотрим наиболее часто встречающиеся *бинарные* связи.

При связывании двух таблиц выделяют основную (родительскую) и подчиненную (дочернюю) таблицы. Это означает, что одна запись родительской таблицы может быть связана с одной или несколькими записями дочерней таблицы. Для поддержки этих связей обе таблицы должны содержать наборы атрибутов, по которым они связаны, т.е. внешние ключи.

В реляционных базах данных допускается три типа связи:

- один – к – одному (1:1);
- один – ко – многим (1 : М);
- многие – ко – многим (М : М).

Тип связи определяется тем, как соотносятся первичные ключи с внешними ключами в обеих таблицах (табл. 2.1).

Дадим характеристику названным типам связи между двумя таблицами и приведем примеры их использования.

#### Связь типа 1:1

Связь один – к – одному означает, что каждая запись одной таблицы соответствует одной записи в другой таблице.

Таблица 2.1 Типы связи таблиц

| Типы связи таблиц | 1:1 | 1:M | M: M |
|---|---|---|---|
| Внешние ключи родительской таблицы | являются первичным ключом | являются первичным ключом | не являются первичным ключом |
| Внешние ключи дочерней таблицы | являются первичным ключом | не являются первичным ключом | не являются первичным ключом |

В качестве примера рассмотрим две таблицы ЛИЧНОСТЬ (Код_личности, ФИО, Адрес, Телефон, Дата_рожд.) и СЛУЖАЩИЙ (Код_служащего, Должность, Квалификация (Разряд), Зарплата, Дата_поступления, Дата_увольнения). Обе таблицы содержат информацию о сотрудниках некоторой компании. В таблице ЛИЧНОСТЬ содержатся сведения о личности сотрудника, а в таблице СЛУЖАЩИЙ – профессиональные сведения. Между этими таблицами существует связь один–к–одному, поскольку для одного человека может существовать только одна запись, содержащая профессиональные сведения.

Связь между этими таблицами поддерживается при помощи внешних ключей, в качестве которых используются совпадающие поля: Код_личности (таблица ЛИЧНОСТЬ) и Код_служащего (таблица СЛУЖАЩИЙ). Отметим, что эти поля (ключи) имеют разные наименования. Связь между таблицами устанавливается на основании значений совпадающих полей, но не их наименований.

На практике связи вида 1:1 используются сравнительно редко, так как хранимую в двух таблицах информацию легко объединить в одну таблицу, которая занимает гораздо меньше места в памяти ЭВМ. Возможны случаи, когда удобнее иметь не одну, а две и более таблицы. Причинами этого может быть необходимость ускорить обработку, повысить удобство работы нескольких пользователей с общей информацией, обеспечить более высокую степень защиты информации и т. д. Приведем пример, иллюстрирующий последнюю из приведенных причин.

Пусть имеются сведения о выполняемых в некоторой организации научно-исследовательских работах. Эти данные включают в себя следующую информацию по каждой из работ: тему (полное наимено-

вание работ), шифр (код), даты начала и завершения работы, количество этапов, головного исполнителя и другую дополнительную информацию. Все работы имеют гриф «Для служебного пользования» или «секретно».

В такой ситуации всю информацию целесообразно хранить в двух таблицах: в одной из них — всю секретную информацию (например, шифр, полное наименование работы и головной исполнитель), а в другой — всю оставшуюся несекретную информацию. Обе таблицы можно связать по шифру работы. Первую из таблиц целесообразно защитить от несанкционированного доступа.

**Связь типа 1 : М**

Связь 1 : М имеет место в случае, когда одной записи родительской таблицы соответствует несколько записей дочерней таблицы.

Например, рассмотрим ситуацию, когда надо описать карьеру некоторого индивидуума. Каждый человек в своей трудовой деятельности сменяет несколько мест работы в разных организациях, где он работает в разных должностях. Кроме того, важно не только отследить переход работника из одной организации в другую, но и прохождение его по служебной лестнице в рамках одной организации. Тогда мы должны создать две таблицы: СОТРУДНИК (<u>Паспорт</u>, Фамилия, Имя, Отчество) для моделирования всех работающих людей и КАРЬЕРА (Дата, <u>Место работы, Должность</u>, Номер приказа, Паспорт) для моделирования записей в их трудовых книжках.

Внешним ключом является поле Паспорт, причем в таблице КАРЬЕРА это поле не является ключевым.

**Связь типа М : М**

Связь много – ко – многим возникает между двумя таблицами в тех случаях, когда:

- одна запись из первой таблицы может быть связана более чем с одной записью второй таблицы;
- одна запись из второй таблицы может быть связана более чем с одной записью первой таблицы.

Например, преподаватель может вести занятия по разным предметам у разных студентов, а студенты, изучая разные предметы, учатся у нескольких преподавателях. Между этими двумя таблицами имеется связь типа много – ко – многим. Современные СУБД связь М : М

не поддерживают. Для того, чтобы организовать такую связь необходимо использовать дополнительную таблицу – таблицу связи, которая связана с каждой таблицей, а тип связи один-ко-многим.

### 2.4.4. Манипулирование данными в реляционной модели

Основной единицей обработки данных в реляционной модели является отношение (таблица), а не отдельные его кортежи (записи) БД, и результатом обработки также является отношение. Далее полученное новое отношение может быть использовано в другой операции.

Для манипулирования данными в реляционной модели используются два формальных аппарата:

- реляционная алгебра, основанная на теории множеств;
- реляционное исчисление, базирующееся на исчислении предикатов первого порядка.

Оба этих аппарата лежат в основе языков манипулирования данными (ЯМД), предназначенных выполнять соответствующие операции над отношениями (таблицами) для получения информации из баз данных. Наиболее важной частью ЯМД является его раздел для формулировки запросов.

Отличаются два этих формальных аппарата уровнем процедурности. Запрос, представленный на языке реляционной алгебры, может быть реализован как последовательность элементарных алгебраических операций с учетом их старшинства и возможного наличия скобок, обеспечивающих пошаговое решение задачи. Поэтому языки реляционной алгебры являются процедурными.

Для формулы реляционного исчисления соответствующая однозначная последовательность действий, вообще говоря, отсутствует. Формула только устанавливает условия, которым должны удовлетворять кортежи результирующего отношения. Поэтому языки реляционного исчисления являются непроцедурными или декларативными. *Непроцедурный язык* – язык, позволяющий формулировать, что нужно сделать, а не как этого добиться.

Механизмы реляционной алгебры и реляционного исчисления эквивалентны, т.е. для любого допустимого выражения реляционной алгебры можно построить эквивалентную формулу реляционного исчисления и наоборот.

Реляционная алгебра и реляционное исчисление в настоящее время в реализациях конкретных реляционных СУБД сейчас не ис-

пользуется в чистом виде. Стандартом доступа к реляционным данным стал язык SQL (Structured Query Language). Тем не менее, знание алгебраических и логических основ языков баз данных часто бывает полезно на практике.

#### 2.4.4.1. Реляционная алгебра

Существует много подходов к определению наборов операторов и способов их интерпретации, но в принципе все они являются более или менее равносильными. Здесь будет рассмотрен классический подход, предложенный Э. Коддом.

В этом варианте множество операторов состоит из восьми операций, составляющих две группы по четыре оператора в каждой:

- традиционные операции над множествами: *объединение, пересечение, вычитание* и *декартово произведение* (все они модифицированы с учетом того, что их операндами являются отношения, а не произвольные множества).
- специальные реляционные операторы: *выборка, проекция, соединение и деление*.

Кроме того, в состав алгебры включается операция *присваивания*, позволяющая сохранить в базе данных результаты вычисления алгебраических выражений. Операция присвоения является стандартной операцией компьютерного языка. Операцию присвоения будем обозначать знаками «:=».

Рассмотрим операции в том порядке, в котором они перечислены выше.

Прежде, чем говорить об операциях, введем понятие: два отношения являются совместимыми по объединению, если имеют одинаковое число атрибутов и $i$-атрибут одного отношения должен быть определен на том же домене, что и $i$-атрибут второго отношения.

**Объединение (union) ($R$:=$R1 \cup R2$)**

Объединением двух совместимых по объединению отношений $R1$ и $R2$ является отношение $R$, включающее в себя все кортежи отношений $R1$ и $R2$ без повторов.

Например, пусть имеются отношение $R1(A,B,C)$ (рис. 2.5, а) и отношение $R2(A,B,C)$ (рис. 2.5, б). Тогда объединение $R$:=$R1$ union $R2$ будет таким, как приведено на рис. 2.5, в.

*R*1

| A | B | C |
|---|---|---|
| 1 | 3 | 4 |
| 2 | 5 | 7 |
| 6 | 8 | 1 |
| 9 | 3 | 2 |
| 4 | 5 | 7 |

а)

*R*2

| A | B | C |
|---|---|---|
| 2 | 4 | 6 |
| 6 | 8 | 1 |
| 5 | 7 | 3 |
| 4 | 5 | 7 |

б)

*R*

| A | B | C |
|---|---|---|
| 1 | 3 | 4 |
| 2 | 5 | 7 |
| 6 | 8 | 1 |
| 9 | 3 | 2 |
| 4 | 5 | 7 |
| 2 | 4 | 6 |
| 5 | 7 | 3 |

в)

Рис. 2.5. Пример операции объединения

Результатом выполнения этой операции является новое отношения *R*, содержащее кортежи, которые принадлежат отношению *R*1 или отношению *R*2 или им обоим.

Этот пример, впрочем, как и другие примеры, подтверждает тезис о том, что с помощью одной операции реляционной алгебры решается простейшая информационная задача. Операция объединение используется для решения практической задачи – слияние файлов однотипных записей.

Заметим, что с помощью операции объединения может быть реализовано добавление новой записи к имеющемуся отношению *R:=R* union *R*2. В этом случае *R* – исходное  отношение, *R2*– отношение, содержащее одну добавляемую запись.

**Пересечение (intersect)  (*R* :=*R*1 ⋂ *R*2)**

Пересечением двух совместимых по объединению отношений *R*1 и *R*2 является отношение *R*, включающее в себя все кортежи как отношения *R*1, так и отношения *R*2.

Например, пусть имеются отношение *R*1(*A,B,C*) (рис. 2.6, а) и отношение *R*2(*A,B,C*) (рис. 2.6, б). Тогда пересечение *R:=R*1 intersect *R*2 будет таким, как приведено на рис. 2.6, в.

*R*1

| A | B | C |
|---|---|---|
| 1 | 3 | 4 |
| 2 | 5 | 7 |
| 6 | 8 | 1 |
| 9 | 3 | 2 |
| 4 | 5 | 7 |

а)

*R*2

| A | B | C |
|---|---|---|
| 2 | 4 | 6 |
| 6 | 8 | 1 |
| 5 | 7 | 3 |
| 4 | 5 | 7 |

б)

*R*

| A | B | C |
|---|---|---|
| 6 | 8 | 1 |
| 4 | 5 | 7 |

в)

Рис. 2.6. Пример операции пересечения

Результатом выполнения операции пересечения будет реляционная таблица, состоящая из всех строк, встречающихся в обеих исходных таблицах. Например, если *R* пересечение *R*1 и *R*2: *R* := *R*1 intersect *R*2, то *R* состоит из тех строк, которые есть и в *R*1, и в *R*2.

**Разность (except) (*R* :=*R*1 - *R*2)**

Разностью двух совместимых по объединению отношений *R*1 и *R*2 является отношение *R*, кортежи которого принадлежат отношению *R*1 и не принадлежат отношению *R*2, т.е. кортежи отношения *R*1, которых нет в отношении *R*2.

Например, пусть имеются отношение *R*1(*A,B,C*) (рис. 2.7, а) и отношение *R*2(*A,B,C*) (рис. 2.7, б). Тогда разность *R:*=*R*1 except *R*2 будет такой, как приведено на рис. 2.7, в.

*R*1

| A | B | C |
|---|---|---|
| 1 | 3 | 4 |
| 2 | 5 | 7 |
| 6 | 8 | 1 |
| 9 | 3 | 2 |
| 4 | 5 | 7 |

а)

*R*2

| A | B | C |
|---|---|---|
| 2 | 4 | 6 |
| 6 | 8 | 1 |
| 5 | 7 | 3 |
| 4 | 5 | 7 |

б)

*R*

| A | B | C |
|---|---|---|
| 1 | 3 | 4 |
| 2 | 5 | 7 |
| 9 | 3 | 2 |

в)

Рис. 2.7. Пример операции разность

**Произведение (product) (*R* := *R*1 * *R*2)**

Это бинарная операция над разносхемными отношениями.

Если отношение *R*1($A_1$, $A_2$, …, $A_m$) имеет *m* атрибутов, а отношение *R*2($B_1$, $B_2$, …, $B_n$) – *n* атрибутов, то произведением является отношение *R*($A_1$, $A_2$, …, $A_m$, $B_1$, $B_2$, …, $B_n$), имеющее (*m* + *n*) атрибутов. Причем первые *m* атрибута образуют атрибуты отношения *R*1, а последние *n* атрибута – атрибуты отношения *R*2.

Например, пусть имеются отношение *R*1(*A,B,C*) (рис. 2.8, а) и отношение *R*2(*D,E*) (рис. 2.8, б). Тогда произведение *R:*=*R*1 product *R*2 будет таким, как приведено на рис. 2.8, в.

Следует заметить, что найти очевидные применения операции произведения практически невозможно. Операция произведения имеет большое значение в качестве составной части операции соединения, которая вероятно, является наиболее важной операцией в реляционной алгебре.

$R1$

| A | B | C |
|---|---|---|
| 1 | 4 | 7 |
| 2 | 5 | 8 |
| 3 | 6 | 9 |

а)

$R2$

| D | E |
|---|---|
| a | c |
| b | f |

б)

$R$

| A | B | C | D | E |
|---|---|---|---|---|
| 1 | 4 | 7 | a | c |
| 1 | 4 | 7 | b | f |
| 2 | 5 | 8 | a | c |
| 2 | 5 | 8 | b | f |
| 3 | 6 | 9 | a | c |
| 3 | 6 | 9 | b | f |

c)

Рис. 2.8. Пример произведения отношений

### Селекция (select) (Выборка)

*Селекция* – унарная операция реляционной алгебры, производящая отбор кортежей из отношения на основании некоторых условий, которые накладываются на значения определённых атрибутов.

Условия задаются логическим выражением вида «Имя атрибута – Знак сравнения – Значение», которые могут соединяться логическими операторами AND (И), OR (ИЛИ) и иметь оператор NOT (ОТРИЦАНИЕ). Знаками сравнения являются следующие знаки: =, <>, >, >=,<, <=.

В качестве примера рассмотрим отношение СКЛАД (рис. 2.9, а).

Нам необходима информация о диодах, хранящихся на складе, по цене меньше 2 руб. Ответ на этот запрос можно получить, выбрав из отношения СКЛАД кортежи, в которых НАИМЕНОВАНИЕ = «Диод», а значение атрибута ЦЕНА < 2.00. В реляционной алгебре это делается следующим образом:

SELECT (СКЛАД: НАИМЕНОВАНИЕ = 'Диод' AND ЦЕНА < 2.00)

Результатом такой выборки будет отношение, представленное на рис. 2.9, б.

| НОМЕНКЛАТУРНЫЙ НОМЕР | НАИМЕНОВАНИЕ | МАРКА | КОЛИЧЕСТВО | ЦЕНА |
|---|---|---|---|---|
| 7120027 | Диод | Д18 | 205 | 0.75 |
| 7120381 | Диод | 2Д522Б | 487 | 1.80 |
| 7121301 | Диод | КД521А | 803 | 2.35 |
| 7509118 | Микросхема | К555ЛА3 | 64 | 4.34 |
| 7509069 | Микросхема | К555ТЛ2 | 4551 | 2.17 |

а)

| НОМЕНКЛАТУРНЫЙ НОМЕР | НАИМЕНОВАНИЕ | МАРКА | КОЛИЧЕСТВО | ЦЕНА |
|---|---|---|---|---|
| 7120381 | Диод | 2Д522Б | 487 | 1.80 |
| 7121301 | Диод | КД521А | 803 | 2.35 |

б)

Рис. 2.9. Пример операции выборка

### Проекция (project)

*Проекция* – это унарная операция реляционной алгебры, создающая новое отношение путем исключения атрибутов из существующего отношения.

Пусть имеется отношение *R*1(A,B,C) (рис. 2.10, а). Выполним операцию проекция по атрибутам C и A. В результате получаем отношение *R*, представленное на рис. 2.10, б.

*R*1

| A | B | C |
|---|---|---|
| 1 | 3 | 4 |
| 2 | 5 | 7 |
| 6 | 8 | 1 |
| 9 | 3 | 2 |
| 4 | 5 | 7 |

а)

*R*

| C | A |
|---|---|
| 4 | 1 |
| 7 | 2 |
| 1 | 6 |
| 2 | 9 |
| 7 | 4 |

б)

Рис. 2.10. Пример операции проекция

### Соединение (join)

*Операция соединения* позволяет строить новое отношение посредством конкатенции (сцепления) кортежей двух исходных отношений. Однако канкатенция производится лишь при выполнении заданного логического условия.

Если условием является равенство значений двух атрибутов исходных отношений, такая операция называется **эквисоединением**. *Естественным* называется эквисоединение по одинаковым атрибутам исходных отношений.

Рассмотрим следующий пример. Пусть имеются отношения *R1*(*A*,*B*,*C*) (рис. 2.11, а) и *R*2(*D*,*E*) (рис. 2.11, б). Выполним операцию соединения этих двух отношений при условии, что значения атрибута *B* отношения *R*1 меньше значения атрибута *D* таблицы *R*2. Результат представлен на рис. 2.11, в.

*R*1

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

а)

*R*2

| D | E |
|---|---|
| 3 | 1 |
| 6 | 2 |

б)

*R*

| A | B | C | D | E |
|---|---|---|---|---|
| 1 | 2 | 3 | 3 | 1 |
| 1 | 2 | 3 | 6 | 2 |
| 4 | 5 | 6 | 6 | 2 |

в)

Рис. 2.11. Пример операции соединение

Поясним подробнее формирование записей этой таблицы. В отношении *R*1 значение второго атрибута (*B*) первого кортежа меньше значения атрибута *D* отношения *R*2 (2 < 3 и 2 < 6), поэтому формируем первые два кортежа отношения *R* путем объединения первого кортежа отношения *R*1 и соответствующих кортежей отношения *R*2. Далее обрабатываем второй кортеж отношения *R*1. Значение второго атрибута второго кортежа отношения *R*1 удовлетворяет заданному условию только для второго кортежа отношения *R*2, поэтому формируется только один кортеж: второй кортеж отношения *R*1 и второй кортеж отношения *R*2. Третий кортеж отношения *R*1 не удовлетворяет заданному условию, поэтому его атрибуты и не встречаются в результирующем отношении.

Тот же результат может быть получен посредством выполнения последовательности операций произведения и выборки:

*R* := *R*1 product *R*2

*R* := SELECT (*R*1.B < *R*2.D)

**Деление (division)**

*Операция деление* в определенном смысле обратна операции произведения.

Пусть отношение *R*1 содержит атрибуты ($A_1$,$A_2$, …, $A_m$, $B_1$, $B_2$, …,$B_n$), а отношение *R*2– атрибуты ($B_1$, $B_2$, …,$B_n$). Тогда результирующее отношение *R* содержит атрибуты ($A_1$,$A_2$, …, $A_m$). Кортеж отношения *R*1 включается в результирующее отношение, если его декартово произведение с отношением *R*2 входит в *R*1.

Пример: пусть имеются отношения *R*1(*A,B,C,D*) (рис. 2.12, а) и *R*2(*C,D*) (рис. 2.12, б). Тогда частное *R* будет таким как показано на рис. 2.12, в.

Строка *ab* принадлежит *R*, поскольку строки *abcd* и *abef* принадлежит *R*1. По аналогичной причине строка *ed* принадлежит *R*. Строка *bc* является единственной в первых двух столбцах *R*1, не принадлежащая *R*, так как *bccd* не принадлежит *R*.

*R*1

| A | B | C | D |
|---|---|---|---|
| a | b | c | d |
| a | b | e | f |
| b | c | e | f |
| e | d | c | d |
| e | d | e | f |
| a | b | d | e |

а)

*R*2

| C | D |
|---|---|
| c | d |
| e | f |

б)

*R*

| A | B |
|---|---|
| a | b |
| e | d |

в)

Рис. 2.12. Пример операции деления

#### 2.4.4.2. Свойства операций реляционной алгебры

Рассмотрим основные алгебраические тождества, справедливые для операций реляционной алгебры.

Операции объединения, пересечения, произведения и соединения *коммутативны*, т.е. для отношений *R*1 и *R*2 справедливо:

*R*1 union *R*2 = *R*2 union *R*1;

*R*1 intersect *R*2 = *R*2 intersect *R*1;

*R*1 product *R*2 = *R*2 product *R*1;

*R*1 join *R*2 = *R*2 join *R*1.

Операции пересечения, объединения, произведения и соединения ассоциативны, т.е. для отношений *R*1, *R*2 и *R*3 справедливо:

*R*1 intersect (*R*2 intersect *R3*) = (*R*1 intersect *R*2) intersect *R*3;

*R*1 union (*R*2 union *R3*) = (*R*1 union *R*2) union *R*3;

*R*1 product ( *R*2 product *R*3) = (*R*1 product *R*2) product *R*3;

*R*1 join ( *R*2 join *R*3) = (*R*1 join *R*2) join *R*3.

Пары операций объединения, пересечения, произведения (или соединения) подчиняются дистрибутивным законам.

Пересечение дистрибутивно относительно объединения:

*R*1 intersect (*R*2 union *R*3) = (*R*1 intersect *R*2) union (*R*1 intersect *R*3).

Объединение дистрибутивно относительно пересечения:

*R*1 union (*R*2 intersect *R3*) = (*R*1 union *R*2) intersect (*R*1 union *R3*).

Произведение и соединение дистрибутивны относительно операций объединения и пересечения:

*R*1 product (*R*2 union *R3*) = (*R*1 product *R*2) union (*R*1 product *R3*);

*R*1 product (*R*2 intersect *R3*) = (*R*1 product *R*2) intersect (*R*1 product *R3*).

Кроме этого, имеются тождества, связывающие унарные операции друг с другом и с бинарными операциями: можно заменять одну операцию несколькими более простыми и обратно. Существование алгебраических тождеств позволяет преобразовывать алгебраические выражения в эквивалентные. Результат вычисления эквивалентных выражений будет, конечно, одинаковым, однако сложность вычисления может отличаться очень значительно (на несколько порядков). СУБД может выбрать среди эквивалентных способов записи запроса в виде выражения такой, выполнение которого требует меньшего количества вычислительных ресурсов (таких как процессорное время или количество операций обмена данными).

#### 2.4.4.2. Реляционное исчисление

Так же, как и выражения реляционной алгебры, формулы реляционного исчисления определяются над отношениями, и результатом вычисления также является отношение.

Реляционное исчисление позволяет путем использования исчисления предикатов, кванторов и переменных выборки (переменные-кортеж, переменные на доменах) описать отношения и операции над ними в виде аналитического выражения или формулы.

Напомним, что предикатом $P(x_1, x_2, ..., x_n)$ называется функция, принимающая значения "истина" или "ложь", от аргументов, определенных в конкретных областях $D_1, D_2, ..., D_n$. При построении функции используются логические операторы конъюнкция ($\wedge$), дизъюнкция ($\vee$), отрицание, импликация ($\rightarrow$), эквивалентность ($\leftrightarrow$). Кроме того, в функции могут использоваться термы сравнения $x_i\ \theta\ x_j$, где $\theta$ - символ операции сравнения $=, \neq, >, <, \geq, \leq$.

При записи формул полезно ввести понятие "свободных" и "связанных" переменных. Вхождение переменной в формулу является "связанным", если этой переменной предшествует квантор существования $\exists$ (читается некоторые, хотя бы один существует и т.п.) и квантор всеобщности $\forall$(читается все, всякий, каков бы ни был и т.п.). В противном случае переменную называют "свободной". Таким образом, кванторы существования $\exists$ и всеобщности $\forall$позволяют отнести формулу ко всему рассматриваемому множеству. Так выражение $\exists\ x( f(x) > a)$ означает, что среди элементов множества $X$ найдется по

крайней мере один, при котором оказывается истинным неравенство, заключенное в скобках. Если использовать квантор всеобщности $\forall x($ $f(x) > a)$, то получим высказывание: для всех элементов множества $X$ некоторая функция $f(x)$ больше заданного значения $a$.

В реляционном исчислении принято связывать с отношением $R(\text{A}_1, \text{A}_2, \ldots, \text{A}_n)$ некоторый предикат $P(x_1, x_2, \ldots, x_n)$, аргументы которых имеют одинаковые области определения, таким образом, что если $P(a_1, a_2, \ldots, a_n) = 1$, то кортеж $(a_1, a_2, \ldots, a_n)$ принадлежит отношению $R$. В противном случае кортеж не входит в состав указанного отношения. Отсюда следует, что посредством задания некоторого предиката может быть задано и соответствующее ему отношение.

Пример построения отношения с помощью реляционного исчисления. Пусть задано отношение $R1(\text{A}_1, \text{A}_2)$ (рис. 2.13, а).

Поставим отношению $R1$ в соответствие предикат $P1$: $R1(\text{A}_1, \text{A}_2) \leftrightarrow P1(x_1, x_2)$, где переменные $x_1$ и $x_2$ имеют области определения $\text{A}_1$ и $\text{A}_2$. Тогда предикат $P2(x_1) \leftrightarrow \forall x_2\ P1(x_1, x_2) \wedge (x_2 > 2)$ формирует новое отношение $R2(\text{A}_3)$, в которую войдут лишь те значения $x_1$, для которых соответствующие значения $x_2$ в отношение $R1$ оказались больше 2 (рис. 2.13,б).

$R1(\text{A}_1, \text{A}_2)$

| $\text{A}_1$ | $\text{A}_2$ |
|---|---|
| 5 | 1 |
| 10 | 4 |
| 7 | 2 |
| 9 | 8 |

а)

$R2(\text{A}_3)$

| $\text{A}_3$ |
|---|
| 10 |
| 9 |

б)

Рис. 2.13. Пример создания отношения с помощью реляционного исчисления

Таким образом, реляционное исчисление позволяет записывать различного вида запросы и создавать новые отношения.

Следует отметить, что между реляционной алгеброй и реляционным исчислением существует тесная связь. Это означает, что любой запрос, который можно сформулировать при помощи логического исчисления, также можно сформулировать, пользуясь реляционной алгеброй, и наоборот. Так операцию объединения $R{:=}R1$ union $R2$ эк-

вивалентно можно записать следующим образом $R1 \vee R2$, а операцию разность $R := R1$ except $R2$ как $R1 \wedge \neg R2$.

Рассмотренные языки реляционной алгебры и реляционного исчисления служат основой реальных языков манипулирования данными реляционных БД. По своим выразительным способностям эти языки эквивалентны друг другу. Однако языки исчисления – это непроцедурные языки, так как их средствами описать лишь свойства желаемого результата, не указывая, как его получить. Выражения реляционной алгебры, напротив, специфицируют конкретный порядок выполнения операций. При этом пользователь должен сам выполнить оптимизацию своего запроса при его формулировке.

**Пример.** Пусть даны два отношения:

СОТРУДНИКИ (сотр_номер, сотр_имя, сотр_зарпл,отд_номер),
ОТДЕЛЫ(отд_номер, отд_кол, отд_нач).

Здесь атрибут *отд_кол* означает количество сотрудников в отделе, а *отд_нач* – фамилию начальника отдела. Мы хотим узнать имена и номера сотрудников, являющихся начальниками отделов с количеством работников более 10. Выполнение этого запроса средствами реляционной алгебры распадается на четко определенную последовательность шагов:

1. Выполнить соединение отношений СОТРУДНИКИ и ОТДЕЛЫ по условию *сотр_номер = отдел_нач:*

2. Из полученного отношения произвести выборку по условию *отд_кол* > 10:

3. Получить проекцию результатов предыдущей операции на атрибуты *сотр_имя, сотр_номер:*

Заметим, что порядок выполнения шагов может повлиять на эффективность выполнения запроса. Так, время выполнения приведенного выше запроса можно сократить, если поменять местами этапы 1 и 2. В этом случае сначала из отношения СОТРУДНИКИ будет сделана выборка всех кортежей со значением атрибута отдел_кол > 10, а затем выполнено соединение результирующего отношения с отношением ОТДЕЛЫ. Машинное время экономится за счет того, что в операции соединения участвуют меньшие отношения.

На языке реляционного исчисления данный запрос может быть записан как:

выдать *сотр_имя* и *сотр_но*м для СОТРУДНИКИ таких, что существует ОТДЕЛ с таким же, что и *сотр_ном* значением отд_нач и значением отд_кол большим 10.

Здесь мы указываем лишь характеристики результирующего отношения, но не говорим о способе его формирования. СУБД сама должна решить, какие операции и в каком порядке надо выполнить над отношениями СОТРУДНИКИ и ОТДЕЛЫ. Задача оптимизации выполнения запроса в этом случае также ложится на СУБД.

В общем случае языки манипулирования данными содержат операции, выходящие за рамки возможностей реляционной алгебры и реляционного исчисления. Это прежде всего команды: добавить данные, модифицировать данные, удалить данные. Кроме этих операций обычно представляются и другие дополнительные возможности, например, в формулы реляционного исчисления и операции реляционной алгебры (операции выбора) могут включаться арифметические вычисления.

### 2.4.5. Достоинства и недостатки реляционной модели данных

*Достоинство* реляционной модели данных заключается в простоте, понятности и удобстве физической реализации на ЭВМ. Именно простота и понятность для пользователя явились основной причиной их широкого использования. Проблемы же эффективности обработки данных этого типа оказались технически вполне разрешимыми.

В качестве основного недостатка реляционной модели можно указать дублирование информации для организации связей между таблицами.

Например, база данных имеет следующую схему:

*R*1(*A1, A2, A3, A4*)

*R*2(*A5, A6*)

*R*3(*A7, A8*)

*R*4(*A1, A5*)

*R*5(*A1, A7*)

*R*6(*A5, A7*),

где R1, R2, R3, R4, R5, R6 – это таблицы, а A1, A2, A3, A4, A5, A6, A7 – это атрибуты таблиц. Не трудно заметить, атрибуты A1, A5, A8 присутствуют в нескольких таблицах.

Отсутствие специальных механизмов навигации (как в иерархической или сетевой моделях), с одной стороны, ведёт к упрощению модели, а с другой – к многократному увеличению времени на извлечение данных, т.к. во многих случаях требуется просмотреть всё отношение для поиска нужных данных.

Не смотря на указанные недостатки, реляционная модель данных широко используется в теории и практики баз данных.

### 2.4.6. Ограничения целостности базы данных

Важной компонентой реляционной базы данных являются ограничения целостности базы данных.

В процессе работы с базой данных значения атрибутов отношений и экземпляры связей между ними могут меняться во времени. Поэтому в любой момент реляционная база данных содержит некоторую определенную конфигурацию данных, и эта конфигурация должна отражать реальную действительность (целостность данных). Следовательно, определение базы данных нуждается в расширении, включающем *правила целостности данных,* назначение которых в том, чтобы информировать СУБД о разного рода ограничениях реального мира.

Примерами правил целостности могут быть:

- количество продаваемого товара должно быть больше 0;
- номер паспорта является уникальным значением;
- возраст принимаемого сотрудника на работу обязательно не меньше 14 лет.

Фактически ограничения целостности – это правила, которым должны удовлетворять значения данных в БД. Учесть их это задача разработчика БД. Ограничения целостности включаются в структуру базы данных с помощью средств языка SQL.

За выполнением ограничений целостности следит СУБД в процессе своего функционирования. Она проверяет ограничения целостности каждый раз, когда они могут быть нарушены (например, при добавлении данных, при удалении данных и т.п.), и гарантирует их соблюдение. Если какая-либо команда нарушает ограничение целостности, она не будет выполнена и система выдаст соответствующее сообщение об ошибке. Например, если задать в качестве ограничения правило «Остаток денежных средств на счёте не может быть отрицательным», то при попытке снять со счёта денег больше, чем там есть,

система выдаст сообщение об ошибке и не позволит выполнить эту операцию. Таким образом, ограничения целостности обеспечивают логическую непротиворечивость данных при переводе БД из одного состояния в другое.

Более сложные правила целостности реализуются с помощью хранимых процедур и триггеров, представляющие определенные последовательности команд языка SQL при изменении или добавлении данных в БД.

**Вопросы для самопроверки**

1. Что такое сетевая модель данных?
2. Что такое иерархическая модель данных?
3. Что такое реляционная модель данных?
4. Дайте определение каждому из следующих понятий в контексте реляционной модели данных: а) отношение; б) атрибут; в) домен; г) кортеж.
5. Что такое ключ отношения?
6. Укажите различия между потенциальными ключами и первичным ключом отношения.
7. Что означает понятие «внешний ключ»?
8. Каковы основные достоинства реляционной модели?
9. Какой формальный аппарат используется в реляционной модели для описания запросов к базе данных?
10. Что называется объединением отношений?
11. Что называется разностью отношений?
12. Что такое операция «декартово произведение»?
13. С помощью какой операции выбираются нужные столбцы таблицы?
14. С помощью какой операции выбираются нужные кортежи отношения?
15. В чем смысл операций соединения?
16. В чем отличие языков реляционной алгебры и реляционного исчисления?
17. Для чего используются ограничения целостности данных?
18. Какова роль СУБД в процессе поддержки целостности данных?

# Глава 3. ЯЗЫК SQL

## 3.1. Общие сведения о языке SQL

### 3.1.1. История возникновения и стандарты языка SQL

Язык SQL (Structured Query Language) – Структурированный Язык Запросов – предназначен для работы с реляционными базами данных. Он дает возможность выполнять операции над таблицами (создание, удаление, изменение структуры) и над данными таблиц (выборка, изменение, добавление, удаление).

История возникновения языка SQL начинается в 1970 году, когда доктор Э.Ф. Кодд предложил реляционную модель в качестве новой модели базы данных. Для доказательства жизнеспособности новой модели данных в компании IBM был создан мощный исследовательский проект, получивший название System/R. Проект включал разработку собственно реляционной СУБД и специального языка запросов к базе данных. Первоначально язык получил название SEQUEL (Structured English Query Language) – структурированный английский язык запросов и произносился как «сиквэл». Позже по юридическим соображениям («SEQUEL» был товарной маркой британской авиастроительной группы компаний «Hawker Siddeley») язык SEQUEL был переименован в язык SQL (Structured Query Language) и официальное произношение стало побуквенным «эс-кью-эль».

В дальнейшем этот язык применялся во многих коммерческих СУБД и в силу своего широкого распространения постепенно стал стандартом “де-факто” для языков манипулирования данными в реляционных СУБД.

С момента создания и до наших дней язык SQL претерпел массу изменений, но идеология осталась неизменной.

Совершенно очевидно, что язык SQL никогда не получил бы мирового признания, если бы на него не было никаких стандартов. Стандартизация – важная часть технологических процессов конца XX века. Именно наличие разработанных и официально признанных стандартов позволило утвердиться многим современным технологиям (не только в индустрии разработки программного обеспечения, но и во многих других сферах человеческой деятельности).

В результате выработки общих требований к средствам обработки данных в 1989 г. появился стандарт SQL-89. Иногда этот стандарт называют стандартом ANSI/ISO (ANSI - Американский Нацио-

нальный Институт Стандартов, ISO - Международная Организация по Стандартизации). Подавляющее большинство СУБД поддерживают этот стандарт полностью. Однако развитие информационных технологий, связанных с базами данных, и необходимость реализации переносимых приложений потребовали доработки и расширения первого стандарта.

В 1992 году появилась версия стандарта SQL-92, который часто называется SQL2. В настоящий момент большинство производителей СУБД внесли изменения в свои продукты с тем, чтобы они в большей степени удовлетворяли стандарту SQL-92. Следующим стандартом стал SQL:1999 (SQL3), в котором добавлена поддержка регулярных выражений, рекурсивных запросов, поддержка триггеров, базовые процедурные расширения, нескалярные типы данных и некоторые объектно-ориентированные возможности. В 2003 году был принят стандарт SQL:2003. В него были введены расширения для работы с XML-данными, оконные функции. Позднее в него были внесены некоторые модификации, например, функциональность работы с XML-данными значительно расширена. Так появился стандарт SQL:2006. В стандарте SQL:2008 улучшены возможности оконных функций, устранены некоторые неоднозначности стандарта SQL:2003. В стандарте SQL:2011 реализована поддержка хронологических баз данных, а также поддержка конструкции FETCH. В стандарт SQL:2016 добавлена защита на уровне строк, включены полиморфные табличные функции и текстовый формат обмена данными, основанный на JavaScript.

Элегантность и независимость от специфики компьютерных технологий, а также его поддержка лидерами промышленности в области технологии реляционных баз данных, сделало SQL основным стандартным языком. По этой причине, любой, кто хочет работать с базами данных, должен знать SQL.

Язык SQL является непроцедурным языком, с помощью которого программист определяет только требуемый результат, не указывая алгоритм его достижения. Поэтому первоначально он не содержал команды управления ходом вычислительного процесса, организации подпрограмм, ввода-вывода, описания типов и многое другое, что присуще традиционным языкам программирования. В связи с этим язык SQL автономно не используется. Обычно команды SQL встраи-

ваются в язык программирования СУБД. Кроме того, команды SQL могут выполняться непосредственно в интерактивном режиме.

В архитектуре «клиент-сервер» язык SQL занимает очень важное место. Именно он используется как язык общения клиентского программного обеспечения с серверной СУБД, расположенной на удаленном компьютере. Так, клиент посылает серверу запрос на языке SQL, а сервер разбирает его, интерпретирует, выбирает план выполнения, выполняет запрос и отсылает клиенту результат.

Несмотря на то, что стандарты обозначают некоторое общее понимание того, каким должен быть язык взаимодействия с базой данных, различные производители реализуют его в своих программных продуктах (СУБД) по-разному. Связано это с тем, что для расширения функциональных возможностей и повышения эффективности разработчики конкретной СУБД добавляют к стандартному языку SQL дополнительные команды и функции, исходя из собственного понимания их необходимости, сохраняя при этом некоторые особенности предыдущих версий. Поскольку сферы интересов пользователей различных СУБД отличаются друг от друга, различаются и создаваемые расширения. Таким образом, существует несколько диалектов языка SQL.

Далее речь пойдет о языке Transact-SQL, который используется в СУБД компании Microsoft.

### 3.1.2. Элементы языкаTransact-SQL

Несмотря на наличие стандартов, практически в каждой СУБД применяется свой диалект языка. Для Microsoft SQL Server таким языком является Transact SQL.

Подобно всем другим языкам программирования Transact-SQL представляет собой набор элементов, все множество которых можно разбить на следующие группы:

алфавит;
идентификаторы;
комментарии;
операторы;
типы данных;
функции.

#### 3.1.2.1. Алфавит языкаTransact-SQL

В Transact-SQL используются символы латинского алфавита, цифры, символы подчеркивания (_), процента (%), звездочка (*), вопросительный (?) и восклицательный знаки (!), подстановочный знак (#), знаки арифметических операций. В качестве разделителей используется двойная кавычка (“), апостроф (‘), запятая (,), точка (.), точка с запятой (;), двоеточие (:), квадратные и круглые скобки.

Практически во всех современных системах программирования в настоящее время допускается применение символов кириллицы.

Запятые используются для разделения элементов списка, например, имен столбцов таблицы: ИМЯ, АДРЕС, ГОРОД.

Квадратные скобки используются для задания имен столбцов, которые содержат недопустимые символы, включая пробелы и разделители. Часто имя столбца таблицы может быть образовано из нескольких слов [Название компании].

Если в запрос включены поля нескольких таблиц, то для разделения имени таблицы и имени поля используется точка, например: Фирма.Адрес.

Символы и строки символов заключаются в одинарные кавычки, например, 'К', 'Петров И.П.'.

#### 3.1.2.2. Идентификаторы

Идентификаторы используются для ссылки на объекты баз данных. Построение идентификаторов выполняется на основе следующих правил:

- число символов не более 128, причем в качестве первого может использоваться только буква, символ подчеркивания и знаки @, #, далее могут следовать буквы, цифры или символы #, @, $ и символ подчеркивания;
- идентификатор, начинающийся с символа @, указывает на локальную переменную; два первых символа @@ говорят о том, что идентификатор указывает на глобальную переменную; символ # в начале идентификатора указывает на то, что идентификатор, является именем временной таблицы или процедуры;
- в идентификаторе объектов не могут встречаться пробелы.

Хотя пробелы в идентификаторах запрещены, это ограничение можно обойти, заключив идентификатор в квадратные скобки.

Любую таблицу можно уникально идентифицировать следующим составным именем:

имя БД **.** имя владельца **.** имя таблицы или представления.

К любому столбцу любой таблицы можно обратиться, используя составное имя:

имя БД **.** имя владельца **.** имя таблицы (представления) . имя столбца

Каждый идентификатор в составном имени отделяется от предыдущего точкой.

Имя владельца может быть опущено, если это не приводит к конфликтам имен.

#### 3.1.2.3. Комментарии

Обязательной составляющей любой программы является ее документирование, одним из компонентов которого служат комментарии.

*Комментарий* – это текстовая строка, которая игнорируется при выполнении программы и служит для пояснения выполняемых действий.

Комментарии можно определить двумя способами:

- с помощью символов /* (начало комментария) и */ (конец комментария), между которыми можно разместить любое количество строк комментария;
- с помощью двух символов --, за которыми можно разместить одну строку комментария.

#### 3.1.2.4. Операторы

***Оператор*** – это символ, обозначающий действие, выполняемое над одним или несколькими выражениями.

Операторы Transact-SQL делятся на следующие категории:

1. Арифметические операторы.
2. Операторы присваивания.
4. Операторы сравнения.
5. Логические операторы.
6. Оператор для слияния строк.

*Арифметические операторы* применимы только к числовым значениям и должны иметь два числовых операнда. Исключение составляет знак минус (-), изменяющий знак операнда. В этом случае

минус называется *унарным минусом*. В табл. 3.1 приведены арифметические операторы.

Таблица 3.1. Арифметические операторы

| **Оператор** | **Описание** |
|---|---|
| + | Суммирование двух операндов |
| - | Определение разности двух операндов |
| - | Изменение знака операнда |
| * | Перемножение двух операндов |
| / | Деление первого операнда на второй операнд |
| % | Определение остатка целочисленного деления. Например, 14 % 4 возвращает 2 |

*Оператор присваивания* (=) присваивает значение переменной. Ключевое слово AS служит оператором для присваивания псевдонимов (alias) таблицам или заголовкам столбцов.

*Операторы сравнения* используются для сравнения символов, чисел, дат и возвращают значение TRUE (истина) или FALSE (ложь) в зависимости от результатов сравнения (табл. 3.2). Исключением является случай, когда один из операндов имеет значение NULL. В этом случае любое сравнение возвращает значение NULL.

Таблица 3.2. Операторы сравнения

| **Оператор** | **Описание** | **Пример** | **Результат** |
|---|---|---|---|
| < | Меньше | 10 < 55 | TRUE |
| | | 10 < NULL | NULL |
| <= | Меньше или равно | 4 <= 9 | TRUE |
| = | Равно | 2 = 3 | FALSE |
| >= | Больше или равно | 2 >= 3 | FALSE |
| > | Больше | 33 > 12 | TRUE |
| <> | Не равно | 2 <> 5 | TRUE |

*Логические операторы* применяются в командах языка SQL для проверки истинности какого-либо условия. Логические операторы возвращают булево значение TRUE или FALSE. В табл. 3.3 приведены операторы сравнения.

Таблица 3.3. Логические операторы.

| **Логический оператор** | **Действие** |
|---|---|
| ALL | TRUE, если весь набор сравнений дает результат TRUE |
| AND | TRUE, если оба булевых выражения дают результат TRUE |
| ANY | TRUE, если хотя бы одно сравнение из набора дает результат TRUE |
| BETWEEN | TRUE, если операнд находится внутри диапазона |
| EXISTS | TRUE, если подзапрос возвращает хотя бы одну строку |
| IN | TRUE, если операнд равен одному выражению из списка |
| LIKE | TRUE, если операнд совпадает с шаблоном |
| NOT | Обращает значение любого другого булева оператора |
| OR | TRUE, если любое булево выражение равно TRUE |
| SOME | TRUE, если несколько сравнений из набора дают результат TRUE |

*Оператор слияния строк (конкатенация)* объединяет две строки символов в единую строку. Например, ‘FDS’ + ‘ASD’ дает строку ‘FDSASD’.

*Приоритет операторов*

Если в выражении присутствует несколько операторов, то порядок их выполнения определяется приоритетом операторов. Ниже перечислены уровни приоритета операторов (от самого высокого к самому низкому).

1. () – выражения в скобках.
2. -  – унарные операторы.
3. *, /, % – арифметические операторы умножения и деления.
4. +, - – арифметические операторы сложения и вычитания.
5. =, >, <, >=, <=, <> – операторы сравнения.
6. NOT.
7. AND.
8. ALL, ANY, BETWEEN, IN, LIKE, OR, SOME.
9. = – присваивание значения переменной.

Если операторы имеют одинаковый приоритет, вычисления производятся слева направо. Для изменения порядка выполнения операторов используются круглые скобки. Выражения в скобках вычисляются первыми. Если применяются вложенные скобки, то первыми вычисляются выражения в наиболее глубоко вложенных скобках.

#### 3.1.2.5. Типы данных

*Тип данных* – это характеристика, которая задается для столбца таблицы или переменной. При этом определяется тип хранящейся в них информации. Например: целые числа, числовые данные с плавающей запятой, данные денежного типа, дата, время, текст, двоичные данные и так далее. У каждого столбца, выражения, переменной или параметра есть определенный тип данных. Понятие типа данных в языке MS SQL Server полностью адекватно понятию типа данных в современных языках программирования. СУБД MS SQL Server имеет большое различных типов данных, описание которых приведено в приложении 1.

В Microsoft SQL Server в случаях, когда оператор объединяет два выражения с разными типами данных, происходит неявное преобразование типов. Если такое преобразование не поддерживается, SQL сервер будет выдавать ошибку. Чтобы определять какой тип данных из выражений преобразовывать, SQL Server применяет правила приоритета типов данных.

Для выполнения явных преобразований SQL Server предлагает универсальные функции CONVERT и CAST, с помощью которых значения одного типа преобразовываются в значения другого типа (если такие изменения возможны). CONVERT и CAST примерно одинаковы и могут быть взаимозаменяемыми:

CONVERT (тип_данных[(длина)], выражение)
CAST (выражение AS тип_данных).

#### 3.1.2.6. Функции

*Функция* очень похожа на операцию, но вместо выполнения одного действия, *функция* может состоять из *множества* операций.

В Transact-SQL функции могут использоваться в самых различных случаях: в столбцах со значениями по умолчанию, в вычисляе-

мых столбцах в таблицах или представлениях, в условии отбора в предложении WHERE и т. д.

В СУБД MS SQL Server имеется возможность создания своих собственных функций, которые называются пользовательскими функциями.

Transact-SQL также предоставляет большое количество встроенных функций, описание которых приведено в приложении 2.

Встроенные функции языка Transact-SQL могут быть агрегатными или скалярными.

Агрегатные функции выполняют вычисления над группой значений столбца и всегда возвращают одно значение результата этих вычислений.

Скалярные функции Transact-SQL используются в создании скалярных выражений. Скалярная функция выполняет вычисления над одним значением или списком значений, тогда как агрегатная функция выполняет вычисления над группой значений из нескольких строк. Скалярные функции можно разбить на следующие категории:

строковые функции;

математические функции;

функции для работы с датами.

### 3.1.3. Структура языка SQL

Основу языка SQL составляют команды, которые должны предоставлять пользователям следующие возможности:

создавать базу данных и таблицы с полным описанием их структуры;

выполнять основные операции манипулирования данными (добавление, изменение, удаление данных);

выполнять запросы, осуществляющие преобразование данных в необходимую информацию.

Поэтому, в язык SQL в качестве составных частей входят:

- язык определения данных (Data Definition Language, DDL);
- язык манипулирования данными (Data Manipulation Language, DML);
- язык управления данными (Data Control Language, DCL);
- команды управления транзакциями (Transaction Control Language, TCL).

Подчеркнем, что это не отдельные языки, а различные команды одного языка. Такое деление проведено только лишь с точки зрения различного функционального назначения этих команд.

Каждая команда языка SQL состоит из имени команды и одной или нескольких фраз, начинающихся с ключевого слова и содержащих различные языковые конструкции. Ключевое слово первой фразы является именем всей команды.

*Язык определения данных* (DDL) используется для создания и изменения структуры базы данных и ее составных частей – таблиц, индексов, представлений (виртуальных таблиц), а также триггеров и хранимых процедур. Основными его командами являются:

CREATE DATABASE (создать базу данных)
CREATE TABLE (создать таблицу)
CREATE VIEW (создать представление)
CREATE INDEX (создать индекс)
CREATE TRIGGER (создать триггер)
CREATE PROCEDURE (создать хранимую процедуру)
ALTER DATABASE (модифицировать базу данных)
ALTER TABLE (модифицировать таблицу)
ALTER VIEW (модифицировать представление)
ALTER INDEX (модифицировать индекс)
ALTER TRIGGER (модифицировать триггер)
ALTER PROCEDURE (модифицировать хранимую процедуру)
DROP DATABASE (удалить базу данных)
DROP TABLE (удалить таблицу)
DROP VIEW (удалить представление)
DROP INDEX (удалить индекс)
DROP TRIGGER (удалить триггер)
DROP PROCEDURE (удалить хранимую процедуру).

*Язык манипулирования данными* (DML) позволяет получать доступ и манипулировать данными в существующих объектах базы данных (таблицах и представлениях). Он состоит из четырех команд:

SELECT (выбрать);
INSERT (вставить);
UPDATE (обновить);
DELETE (удалить).

*Язык управления данными* (DCL) используется для управления правами доступа к данным и выполнением процедур в многопользовательской среде. Более точно его можно назвать “язык управления доступом”. Он состоит из двух основных команд:

GRANT (дать права на ряд действий над некоторым объектом БД);

REVOKE (забрать права).

*Команды управления транзакциями* (TCL) позволяют управлять изменениями, сделанными в результате DML-запросов. Пользователь группирует DML-запросы (один или несколько) в логическую единицу, называемую транзакцией, которая является атомарной, т.е. выполняется целиком или не выполняется вовсе. Благодаря этим командам можно осуществлять следующие действия:

BEGIN TRANSACTION (начать транзакцию);
COMMIT TRANSACTION (завершить транзакцию);
ROLLBACK TRANSACTION (откатить транзакцию);
SAVEPOINT TRANSACTION (сохранить промежуточную точку выполнения транзакции).

Заметим, что в базовом языке SQL отсутствуют управляющие операторы: переходы, циклы и т. п. – вследствие чего SQL не является языком программирования. Связано это с первоначальной идеологией реляционных баз данных, когда предполагалось, что при работе с базой данных конечный пользователь будет непосредственно вводить команды SQL в консольном режиме. Однако усложнение структуры баз данных привело к необходимости программной поддержки SQL. В современных СУБД она обеспечивается как с помощью процедурных расширений языка, так и посредством использования реализующих команды SQL библиотек для стандартных языков программирования.

**Вопросы для самопроверки**

1. Как записываются комментарии в языке Transact SQL?
2. Какие классы операторов существуют в языке Transact SQL?
3. Что такое тип данных в контексте баз данных?
4. Что определяет тип данных поля таблицы?
5. К какому типу относятся DECIMAL, NUMERIC?

6. В чем различие между типами CHAR и VARCHAR?
7. Какие типы встроенных функций существуют в TRANSACT SQL?
8. К какому типу относятся функции ABS, LOG и SQRT?
9. К какому типу относятся функции RTRIM, STR и UPPER?
10. К какому типу относятся функции GETDATE, MONTH и DATEADD?
11. Какие языки баз данных объединены в языке SQL?
12. Какие команды относятся к языку управления данными (DCL)?
13. Какие команды относятся к языку определения данными (DDL)?
14. Какие команды относятся к языку манипулирования данными (DML)?
15. Что является операндами в командах языка SQL?
16. Что является результатами выполнения команд языка SQL?

## 3.2. Извлечение данных из таблиц

Описывая команды SQL, логически правильно было бы начать с команд создания базы данных, создания таблиц, наполнения их содержимым и только потом переходить к командам извлечения данных. Однако с практической точки зрения более удобно начать с наиболее важной и сложной команды SQL – команды SELECT.

### 3.2.1. Команда SELECT

#### 3.2.1.1. Базовый синтаксис команды SELECT

Команда SELECT предназначена для извлечения информации из базы данных и позволяет делать выборку одной или нескольких строк или столбцов из одной или нескольких таблиц. При этом результатом выполнения всегда является таблица, и, даже если это только одно число, все равно оно рассматривается как таблица с одной строкой и одним столбцом.

Выполнение оператора SELECT не изменяет хранящихся в базе данных, однако в момент его выполнения запрашиваемые данные обычно блокируются от изменений.

С помощью команды SELECT можно найти и просмотреть данные, причем сделать это можно несколькими способами, и, будучи

все правильными, они, тем не менее, могут существенно отличаться по времени исполнения, что особенно важно для больших баз данных. Поэтому писать запросы на языке SQL совсем не просто. Этому надо учиться, как учатся решать математические задачи или составлять алгоритмы решения задач на ЭВМ.

Базовый синтаксис команды SELECT:

SELECT [ALL | DISTINCT] < список выбираемых полей | *>
FROM [<Имя базы данных.>]<Имя таблицы> | <Имя представления>]
[WHERE < Условия выборки или соединения >]
[GROUP BY <Выражение>]
[HAVING <Выражение>]
[ORDER BY <Выражение>]

**Примечание.**

*Для описания синтаксиса команд примем следующие обозначения:*

[ ] – *содержимое этих скобок является необязательным;*

< > – *содержимое этих скобок заменяется соответствующими ключевыми словами, литералами, идентификаторами или выражениями (в зависимости от контекста);*

| – *заменяет слово ИЛИ. Другими словами этот символ означает обязательный выбор из списка возможностей.*

*Хотя ключевые слова можно записывать в любом регистре, будем для выразительности записывать их прописными буквами.*

В первом предложении команды SELECT определяются поля (столбцы), которые будут входить в результат выполнения запроса.

Опция ALL (это значение установлено по умолчанию) выводит все записи, попавшие в результат запроса. Значит, в результирующий набор могут попасть одинаковые строки. И это нарушение принципов теории отношений. Опция DISTINCT исключает из результата запроса повторяющиеся записи. DISTINCT опускает строки, где все выбранные поля идентичны, т.е. дубликаты строк результата не включаются в набор. Строки, в которых некоторые значения одинаковы, а некоторые различны – будут сохранены.

Аргумент <список выбираемых полей> содержит список полей, а также допустимых выражений, включаемых в результирующую

таблицу. Элементы списка разделяются запятыми. Каждый элемент этого списка генерирует один столбец результатов запроса. В имя поля можно включать имя выбираемой таблицы или имя локального псевдонима таблица. Это бывает необходимо, если поля разных таблиц имеют одинаковые имена. Псевдоним и имя поля разделяются символом “.” (точка). Символ * (звездочка) означает, что в результирующий набор включаются все столбцы из исходных таблиц запроса.

Предложение FROM содержит список таблиц, из которых осуществляется выборка. Таблицы в этом списке разделяются запятой. Для каждой таблицы можно указать имя локального псевдонима, которое должно следовать сразу же за именем таблицы и отделяться от него одним или несколькими пробелами.

В предложении WHERE задаются условия отбора записей из исходных таблиц.

Предложение GROUP BY группирует строки в запросе на основании значения в одном или более полях результирующей таблицы.

Предложение HAVING задает условие фильтра, которому должны удовлетворять группы, чтобы быть включенными в результат запроса. Предложение HAVING следует использовать только вместе с GROUP BY.

Предложение ORDER BY сортирует результат запроса на основании одного или нескольких полей результирующей таблицы.

Обратите внимание, что каждое предложение в команде SELECT необходимо использовать, придерживаясь синтаксического порядка. Например, предложение GROUP BY должно идти до ORDER BY. Иначе вместо ожидаемого результата появится сообщение об ошибке.

Порядок выполнения команды SELECT:

1. FROM – вначале определяются имена используемых таблиц;

2. WHERE – из указанной таблицы выбираются записи, удовлетворяющие заданным условиям;

3. GROUP BY – выполняется группировка полученных записей, т.е. образуются группы строк, имеющих одно и то же значение в указанном столбце;

4. HAVING – выбор группы строк, удовлетворяющих указанным условиям;

5. ORDER BY – выполняется сортировка записей в указанном порядке;

6. SELECT – устанавливается, какие столбцы должны выводиться.

Гибкость и мощь языка SQL состоит в том, что он позволяет объединить все операции реляционной алгебры в одной команде, “вытаскивая” таким образом, любую требуемую информацию.

Костяк этой команды состоит из предложений SELECT, FROM, WHERE, условий поиска и выражений. Любой самый сложный запрос начинается с шаблона:

SELECT (выбрать) <Список полей>
FROM (из) <Список таблиц>
[WHERE (где) <Условия выборки или соединения >]

Для иллюстрации основных возможностей языка SQL мы будем использовать демонстрационную базу данных промышленного предприятия, которая отражает процесс поступления материалов и товаров на его склад. Демонстрационная база данных описана в приложении 3.

#### 3.2.1.2. Список выбираемых столбцов

Первое предложение запроса, создаваемого с помощью команды SELECT, является обязательным. Оно включает в себя список выбираемых элементов, отделяемых друг от друга запятой. Элементами этого списка могут быть названия полей базы данных, вычисляемые поля, значения которых определяются с помощью соответствующих выражений, переменные, константы и функции.

В простейшем варианте команда SELECT, которая выбирает все поля и все записи из одной таблицы, например Склад, выглядит следующим образом:

SELECT * FROM Склад

Результат выполнения запроса приведен на рис. 3.1.

В этом случае поля будут выводиться на экран в том порядке, как они создавались.

Этот пример иллюстрирует выполнение операции выборки. *Операция выборки* позволяет получить все строки (записи) либо часть строк одной таблицы.

```
SELECT * FROM Склад
```

| | Ном_ном | Наименование | Цена | Количество |
|---|---|---|---|---|
| 1 | 10010 | Принтер XEROX | 6520,00 | 15 |
| 2 | 10020 | Принтер Canon | 3990,00 | 10 |
| 3 | 10030 | Принтер Pantum | 5200,00 | 8 |
| 4 | 10040 | Принтер HP | 6990,00 | 7 |
| 5 | 10050 | Монитор Xiaomi 23.8" | 9820,00 | 6 |
| 6 | 10060 | Монитор ASUS 21.5" | 5820,00 | 9 |
| 7 | 10070 | Монитор AOC 18.5" | 4790,00 | 12 |
| 8 | 10080 | Монитор Samsung 28" | 16790,00 | 5 |
| 9 | 10090 | Мат. плата ASUS | 47520,00 | 8 |
| 10 | 10100 | Мат. плата BioStar | 7632,00 | 26 |
| 11 | 10110 | Мат. плата INTEL | 3408,00 | 35 |
| 12 | 10120 | Процессор Intel (i7) | 34320,00 | 7 |
| 13 | 10130 | Процессор Intel Xeon | 48560,00 | 5 |

Рис. 3.1. Простейший вариант команды SELECT

Если нам не нужны все поля таблицы, необходимо требуемые поля перечислить в предложение SELECT, например:

SELECT Наименование, Количество, Цена FROM Склад

Результат выполнения запроса приведен на рис. 3.2.

```
SELECT Наименование, Количество, Цена FROM Склад
```

| | Наименование | Количество | Цена |
|---|---|---|---|
| 1 | Принтер XEROX | 15 | 6520,00 |
| 2 | Принтер Canon | 10 | 3990,00 |
| 3 | Принтер Pantum | 8 | 5200,00 |
| 4 | Принтер HP | 7 | 6990,00 |
| 5 | Монитор Xiaomi 23.8" | 6 | 9820,00 |
| 6 | Монитор ASUS 21.5" | 9 | 5820,00 |
| 7 | Монитор AOC 18.5" | 12 | 4790,00 |
| 8 | Монитор Samsung 28" | 5 | 16790,00 |
| 9 | Мат. плата ASUS | 8 | 47520,00 |
| 10 | Мат. плата BioStar | 26 | 7632,00 |
| 11 | Мат. плата INTEL | 35 | 3408,00 |
| 12 | Процессор Intel (i7) | 7 | 34320,00 |
| 13 | Процессор Intel Xeon | 5 | 48560,00 |

Рис. 3.2. Команда SELECT с требуемыми полями

Операция, которая позволяет выделить подмножество столбцов таблицы, называется *операцией проекции*.

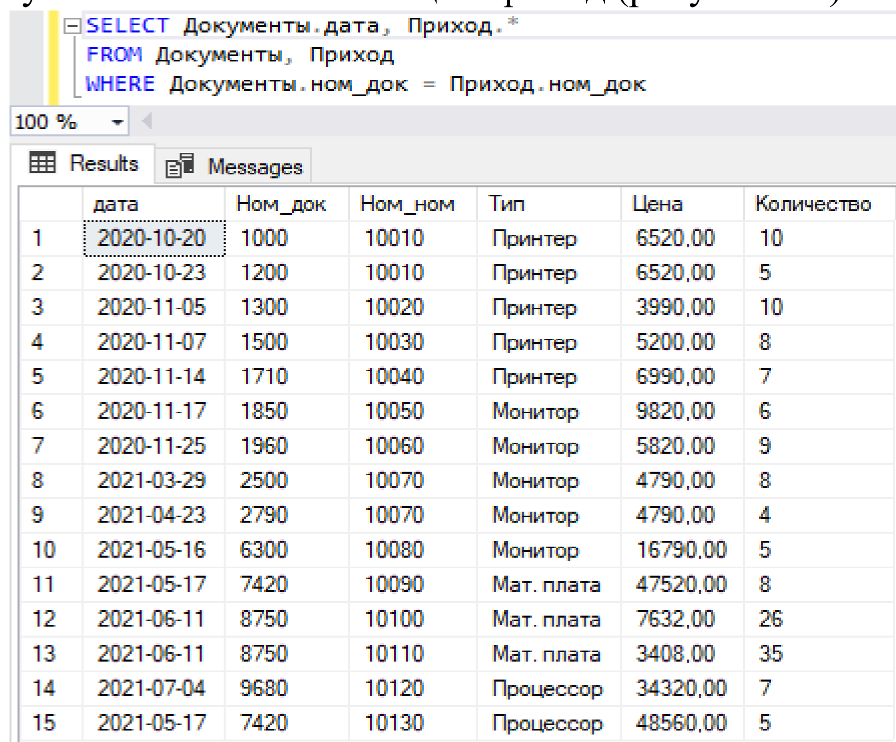
Порядок, в котором поля выводятся в итоге, зависит только от того, в какой последовательности они перечислены в запросе. Если необходимо во второй колонке вывести цену, то следует изменить порядок указания полей в списке:

```
SELECT Наименование, Цена, Количество FROM Склад
```

В запросах из нескольких таблиц можно использовать для столбцов одной таблицы звездочку, а для другой перечислить нужные поля. Например, для таблиц Документ и Приход можно написать запрос:

```
SELECT Документ.дата, Приход.*
FROM Документ, Приход
WHERE Документ.ном_док = Приход.ном_док
```

В этом случае результат запроса будет содержать поле дата таблицы Документ и все поля таблицы Приход (рисунок 3.3).



```
SELECT Документы.дата, Приход.*
FROM Документы, Приход
WHERE Документы.ном_док = Приход.ном_док
```



| | дата | Ном_док | Ном_ном | Тип | Цена | Количество |
|---|---|---|---|---|---|---|
| 1 | 2020-10-20 | 1000 | 10010 | Принтер | 6520,00 | 10 |
| 2 | 2020-10-23 | 1200 | 10010 | Принтер | 6520,00 | 5 |
| 3 | 2020-11-05 | 1300 | 10020 | Принтер | 3990,00 | 10 |
| 4 | 2020-11-07 | 1500 | 10030 | Принтер | 5200,00 | 8 |
| 5 | 2020-11-14 | 1710 | 10040 | Принтер | 6990,00 | 7 |
| 6 | 2020-11-17 | 1850 | 10050 | Монитор | 9820,00 | 6 |
| 7 | 2020-11-25 | 1960 | 10060 | Монитор | 5820,00 | 9 |
| 8 | 2021-03-29 | 2500 | 10070 | Монитор | 4790,00 | 8 |
| 9 | 2021-04-23 | 2790 | 10070 | Монитор | 4790,00 | 4 |
| 10 | 2021-05-16 | 6300 | 10080 | Монитор | 16790,00 | 5 |
| 11 | 2021-05-17 | 7420 | 10090 | Мат. плата | 47520,00 | 8 |
| 12 | 2021-06-11 | 8750 | 10100 | Мат. плата | 7632,00 | 26 |
| 13 | 2021-06-11 | 8750 | 10110 | Мат. плата | 3408,00 | 35 |
| 14 | 2021-07-04 | 9680 | 10120 | Процессор | 34320,00 | 7 |
| 15 | 2021-05-17 | 7420 | 10130 | Процессор | 48560,00 | 5 |

Рис. 3.3. Комбинированный способ задания списка выбираемых полей

*Выражения и функции в выборках*

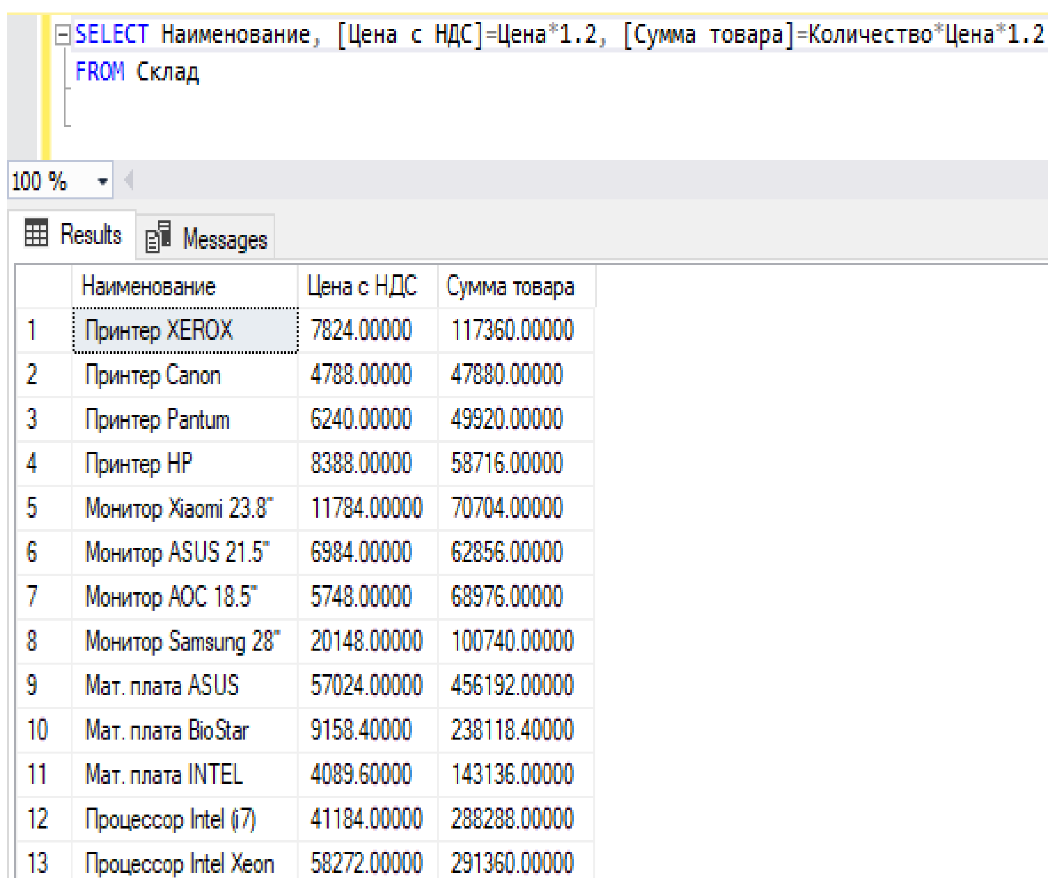
Помимо полей таблиц, в список выводимых полей могут входить выражения и функции.

Все числовые поля могут входить в арифметические выражения. Правила вычисления их значений ни чем не отличаются от правил, известных по школьному курсу арифметики.

Например, из таблицы Склад выбрать наименование товара, цену с учетом НДС (20%) и сумму товара:

```
SELECT Наименование, [Цена с НДС] = Цена*1.2, [Сумма товара] = Количество*Цена*1.2
FROM Склад
```

Результат выполнения запроса приведен на рис. 3.4.



```
SELECT Наименование, [Цена с НДС]=Цена*1.2, [Сумма товара]=Количество*Цена*1.2
FROM Склад
```

100 %

Results | Messages

| | Наименование | Цена с НДС | Сумма товара |
|---|---|---|---|
| 1 | Принтер XEROX | 7824.00000 | 117360.00000 |
| 2 | Принтер Canon | 4788.00000 | 47880.00000 |
| 3 | Принтер Pantum | 6240.00000 | 49920.00000 |
| 4 | Принтер HP | 8388.00000 | 58716.00000 |
| 5 | Монитор Xiaomi 23.8" | 11784.00000 | 70704.00000 |
| 6 | Монитор ASUS 21.5" | 6984.00000 | 62856.00000 |
| 7 | Монитор AOC 18.5" | 5748.00000 | 68976.00000 |
| 8 | Монитор Samsung 28" | 20148.00000 | 100740.00000 |
| 9 | Мат. плата ASUS | 57024.00000 | 456192.00000 |
| 10 | Мат. плата BioStar | 9158.40000 | 238118.40000 |
| 11 | Мат. плата INTEL | 4089.60000 | 143136.00000 |
| 12 | Процессор Intel (i7) | 41184.00000 | 288288.00000 |
| 13 | Процессор Intel Xeon | 58272.00000 | 291360.00000 |

Рис. 3.4. Команда SELECT с вычисляемыми полями

В качестве элементов списка вывода можно использовать функции (приложение 2).

Очень часто в запросах требуется производить обобщенное групповое значение полей. Это делается с помощью агрегатых функций.

К агрегатным функциям относятся специальные функции, действующие “по вертикали“. Это функции вычисления суммы (SUM), максимального (MAX) и минимального (MIN) значений столбцов, арифметического среднего (AVG), а также количества строк, удовлетворяющих заданному условию (COUNT).

Агрегатные функции используются подобно именам полей в предложении SELECT запроса, но с одним исключением, они берут имя поля как аргумент. Только числовые поля могут использоваться с функциями SUM и AVG. С функции COUNT, MAX, и MIN могут использовать числовые, символьные поля, а также поля типа дата.

Пример: определить, сколько видов принтеров хранится на складе, их суммарную, минимальную, максимальную и среднюю цену

SELECT [Количество принтеров] = COUNT(*), [Сумма без НДС] = SUM(Цена), [Минимальная цена] = MIN(Цена), [Максимальная цена] = MAX(Цена), [Средняя цена] = AVG(Цена)
FROM Приход
WHERE Тип = 'Принтер'

Результат выполнения запроса приведен на рис. 3.5.

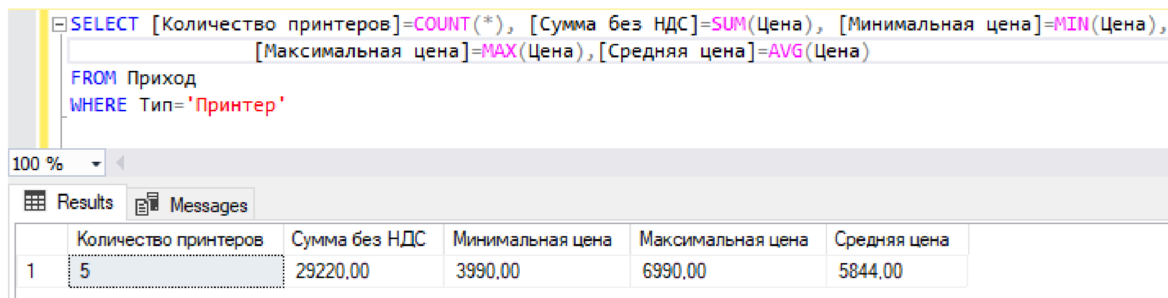



Рис. 3.5. Команда SELECT с агрегатными функциями

В качестве элементов списка вывода можно использовать различные функции, например, функции для работы с датами. Запрос

SELECT Ном_док, [Месяц] = Month(Дата)
FROM Документ

сформирует список номеров документов и номеров месяцев их оформления.

Результат выполнения запроса приведен на рис. 3.6.

```
SELECT Ном_док, [Месяц]=Month(Дата)
FROM Документ
```

100 %

Results | Messages

| | Ном_док | Месяц |
|---|---|---|
| 1 | 1000 | 10 |
| 2 | 1200 | 10 |
| 3 | 1300 | 11 |
| 4 | 1500 | 11 |
| 5 | 1710 | 11 |
| 6 | 1850 | 11 |
| 7 | 1960 | 11 |
| 8 | 2500 | 3 |
| 9 | 2790 | 4 |
| 10 | 6300 | 5 |
| 11 | 7420 | 5 |
| 12 | 8750 | 6 |
| 13 | 9680 | 7 |

Рис. 3.6. Команда SELECT с функцией для работы с датами

В запросах можно использовать строковые функции работы с данными. Например,

SELECT Ном_ном , [Наименование] = LEFT(Наименование,10)
FROM Склад

Из поля Наименование будут выделены первые 10 символов. Результат выполнения запроса приведен на рис. 3.7.

```
SELECT Ном_ном , [Наименование]=LEFT(Наименование,10)
FROM Склад
```

100 %

Results | Messages

| | Ном_ном | Наименование |
|---|---|---|
| 1 | 10010 | Принтер XE |
| 2 | 10020 | Принтер Ca |
| 3 | 10030 | Принтер Pa |
| 4 | 10040 | Принтер HP |
| 5 | 10050 | Монитор Xi |
| 6 | 10060 | Монитор AS |
| 7 | 10070 | Монитор AO |
| 8 | 10080 | Монитор Sa |
| 9 | 10090 | Мат. плата |
| 10 | 10100 | Мат. плата |
| 11 | 10110 | Мат. плата |
| 12 | 10120 | Процессор |
| 13 | 10130 | Процессор |

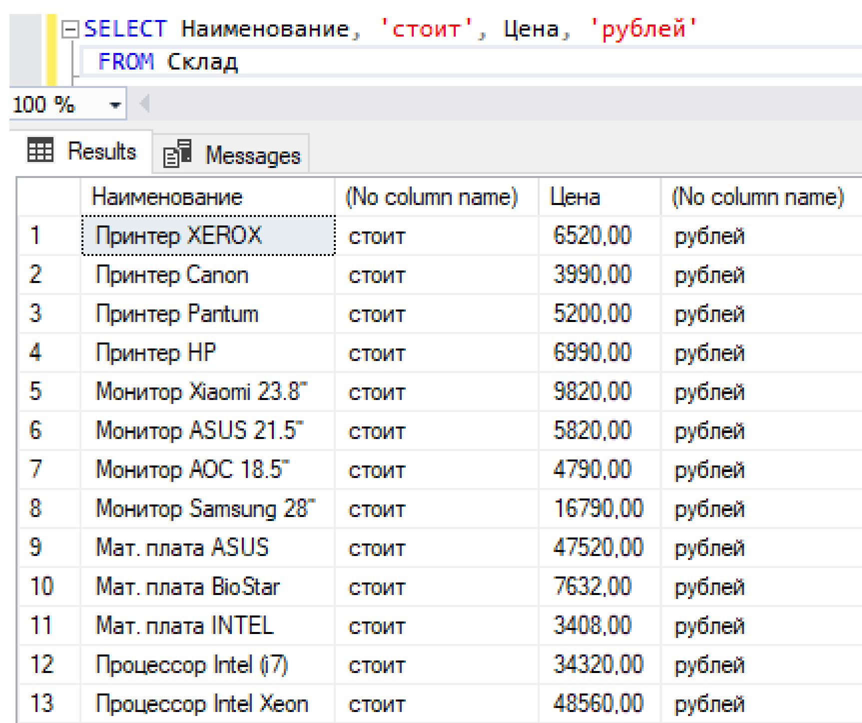
Рис. 3.7. Команда SELECT с функцией для работы со строками

В запросе можно использовать пользовательскую функцию.

Для придания большей наглядности получаемому результату можно использовать литералы. *Литералы* - это строковые константы, которые применяются наряду с наименованиями столбцов и, таким образом, выступают в роли “псевдостолбцов”. Строка символов, представляющая собой литерал, должна быть заключена в кавычки или апострофы. Например, получить список наименований товаров и их цену

```
SELECT Наименование, 'стоит', Цена, 'рублей'
 FROM Склад
```

Результат выполнения запроса приведен на рис.е 3.8.



```
SELECT Наименование, 'стоит', Цена, 'рублей'
 FROM Склад
```

100 %

Results | Messages

| | Наименование | (No column name) | Цена | (No column name) |
|---|---|---|---|---|
| 1 | Принтер XEROX | стоит | 6520,00 | рублей |
| 2 | Принтер Canon | стоит | 3990,00 | рублей |
| 3 | Принтер Pantum | стоит | 5200,00 | рублей |
| 4 | Принтер HP | стоит | 6990,00 | рублей |
| 5 | Монитор Xiaomi 23.8" | стоит | 9820,00 | рублей |
| 6 | Монитор ASUS 21.5" | стоит | 5820,00 | рублей |
| 7 | Монитор AOC 18.5" | стоит | 4790,00 | рублей |
| 8 | Монитор Samsung 28" | стоит | 16790,00 | рублей |
| 9 | Мат. плата ASUS | стоит | 47520,00 | рублей |
| 10 | Мат. плата BioStar | стоит | 7632,00 | рублей |
| 11 | Мат. плата INTEL | стоит | 3408,00 | рублей |
| 12 | Процессор Intel (i7) | стоит | 34320,00 | рублей |
| 13 | Процессор Intel Xeon | стоит | 48560,00 | рублей |

Рис. 3.8. Использование литералов в списке выводимых полей

### *Определение заголовков столбцов*

По умолчанию заголовками столбцов в итоговой выборке являются их имена, которые присвоены им при создании таблицы. Это часто бывает неудобно, особенно для вычисляемых полей. Так как имя заголовка такого столбца либо пусто, либо присваивается системой.

Если не устраивают имена столбцов, формируемые по умолчанию, то для придания наглядности получаемым результатам, программист может изменить их следующими способами:

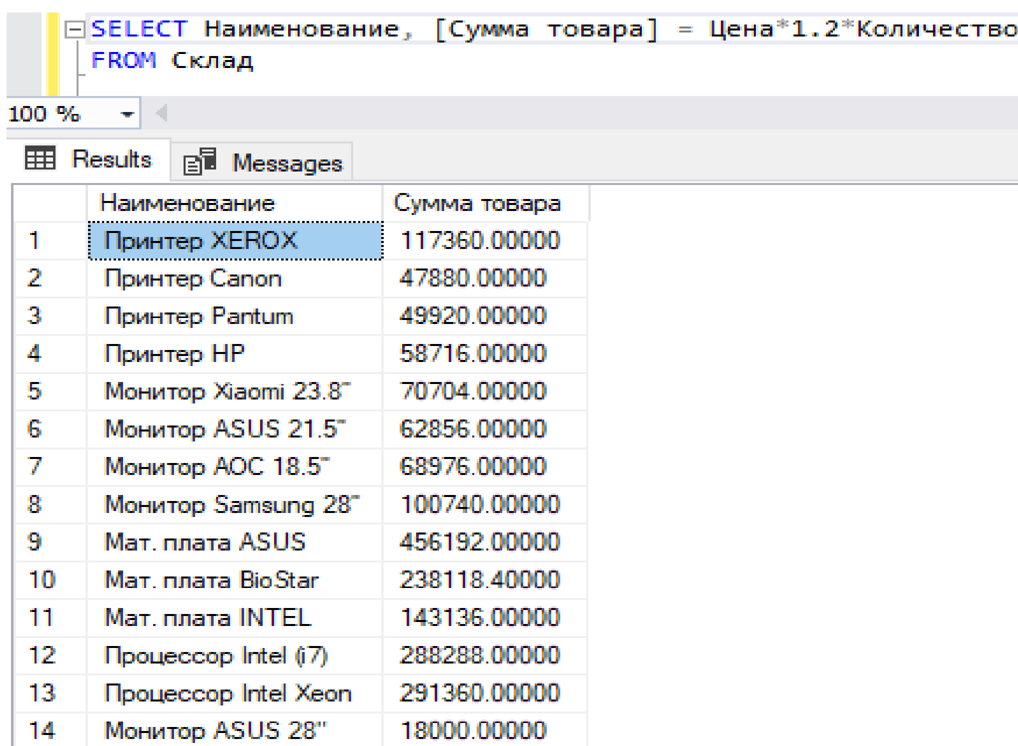
SELECT <Заголовок столбца> = <Имя столбца>
или
SELECT <Имя столбца> AS <Заголовок столбца>

В приведенных ниже примерах продемонстрированы оба способа определения заголовков:

```
SELECT Наименование, [Сумма товара] = Cena*1.20*Kolvo
FROM Склад

SELECT  Наименование, (Kolvo*Cena*1.20) AS 'Сумма товара'
FROM Склад
```

На рис. 3.9 приведен результат запросов.




| | Наименование | Сумма товара |
|---|---|---|
| 1 | Принтер XEROX | 117360.00000 |
| 2 | Принтер Canon | 47880.00000 |
| 3 | Принтер Pantum | 49920.00000 |
| 4 | Принтер HP | 58716.00000 |
| 5 | Монитор Xiaomi 23.8" | 70704.00000 |
| 6 | Монитор ASUS 21.5" | 62856.00000 |
| 7 | Монитор AOC 18.5" | 68976.00000 |
| 8 | Монитор Samsung 28" | 100740.00000 |
| 9 | Мат. плата ASUS | 456192.00000 |
| 10 | Мат. плата BioStar | 238118.40000 |
| 11 | Мат. плата INTEL | 143136.00000 |
| 12 | Процессор Intel (i7) | 288288.00000 |
| 13 | Процессор Intel Xeon | 291360.00000 |
| 14 | Монитор ASUS 28" | 18000.00000 |

Рис. 3.9. Определение заголовков столбцов

Определения заголовков присутствуют в запросах, приведенных на рис. 3.4 – 3.7.

### 3.2.1.3. Использование таблиц, входящих в базу данных

В обязательном предложении FROM перечисляются все объекты базы данных (один или несколько), из которых производится выборка данных. Каждая таблица или представление, которые упоминается в запросе, должны быть перечислены в предложении FROM.

При описании таблиц, входящих в базу данных, из которых производится выборка можно указывать полное имя таблицы:

FROM <Имя базы данных>.<Имя таблицы>

Это обязательно, если делается ссылка на таблицу не из текущей базы данных. Аналогично производится обращение к представлениям.

Установить текущую базу данных можно командой:

USE <Имя базы данных>

В случаях, когда база данных не является текущей и в информационной системе установлено несколько серверов баз данных, при выборке имя столбца необходимо указывать полный адрес:

<Имя сервера>.<Имя базы данных>.<Владелец>.<Имя таблицы>.<Имя столбца>.

Упоминание нескольких таблиц в предложении FROM обеспечивает выполнение операции соединения. *Операция соединения* позволяет соединять строки из более чем одной таблицы (по некоторому условию) для образования новых строк данных.

### 3.2.1.4. Выборка строк

Таблицы имеют тенденцию становиться очень большими, поскольку с течением времени, все большее и большее количество строк в нее добавляется. Поскольку обычно из них только определенные строки интересуют пользователя в данное время, SQL дает возможность устанавливать критерии, чтобы определить какие строки будут выбраны для вывода. Для этого необходимо использовать предложения WHERE.

WHERE - предложение команды SELECT позволяющее устанавливать условие, которое может быть или верным или неверным для любой строки таблицы. Команда извлекает только те строки из таблицы, для которой такое условие верно.

Когда предложение WHERE присутствует, программа просматривает всю таблицу по одной строке и исследует каждую строку, чтобы определить, верно ли заданное условие.

Условия предложения WHERE могут включать в себя операторы сравнения (=, #, <, <=, >, >=), интервала (BETWEEN), вхождения в список (IN), соответствия шаблону LIKE, логические (AND, OR, NOT), сравнения с неопределенным значением (IS NULL).

Например, требуется вывести названия всех поставщиков из Владимирской области. Это можно сделать с помощью команды:

```
SELECT Название, Регион
 FROM Поставщик
 WHERE Регион = 'Владимир.обл.'
```

Результат выполнения запроса приведен на рис. 3.10.

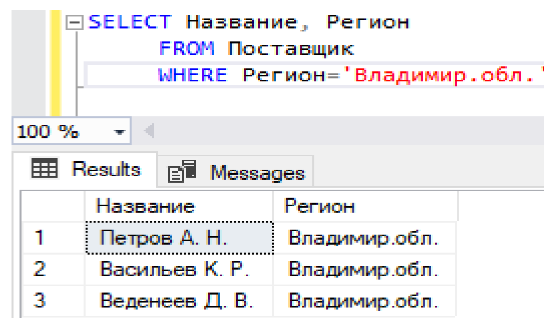



Рис. 3.10. Список поставщиков из Владимирской области

Оператор BETWEEN задает диапазон значений, для которого выражение принимает значение true. Формат этого оператора:

<выражение> BETWEEN <нижнее значение диапазона> AND <верхнее значение диапазона>

Например, вывести все записи таблицы Склад, у которых значение поля Ном_ном находится в диапазоне от 10020 до 10080

```
SELECT *
FROM Склад
WHERE Ном_ном BETWEEN 10020 AND 10080
```

Результат выполнения запроса приведен на рис. 3.11.

```
SELECT *
FROM Склад
WHERE Ном_ном BETWEEN 10020 AND 10080
```

100 %

Results | Messages

| | Ном_ном | Наименование | Цена | Количество |
|---|---|---|---|---|
| 1 | 10020 | Принтер Canon | 3990,00 | 10 |
| 2 | 10030 | Принтер Pantum | 5200,00 | 8 |
| 3 | 10040 | Принтер HP | 6990,00 | 7 |
| 4 | 10050 | Монитор Xiaomi 23.8" | 9820,00 | 6 |
| 5 | 10060 | Монитор ASUS 21.5" | 5820,00 | 9 |
| 6 | 10070 | Монитор AOC 18.5" | 4790,00 | 12 |
| 7 | 10080 | Монитор Samsung 28" | 16790,00 | 5 |

Рис. 3.11. Команда SELECT с оператором BETWEEN

Тот же результат вернет запрос с использованием операторов сравнения:

```
SELECT *
FROM Склад
WHERE (Ном_ном>=10020) AND (Ном_ном<=10080)
```

Значения, определяющие нижнюю и верхнюю границы диапазона, могут не являться реальными величинами из базы данных. И это очень удобно - ведь не всегда можно указать точные значения диапазонов. Например, получить список поставщиков, фамилии которых находятся между Вас и Ро

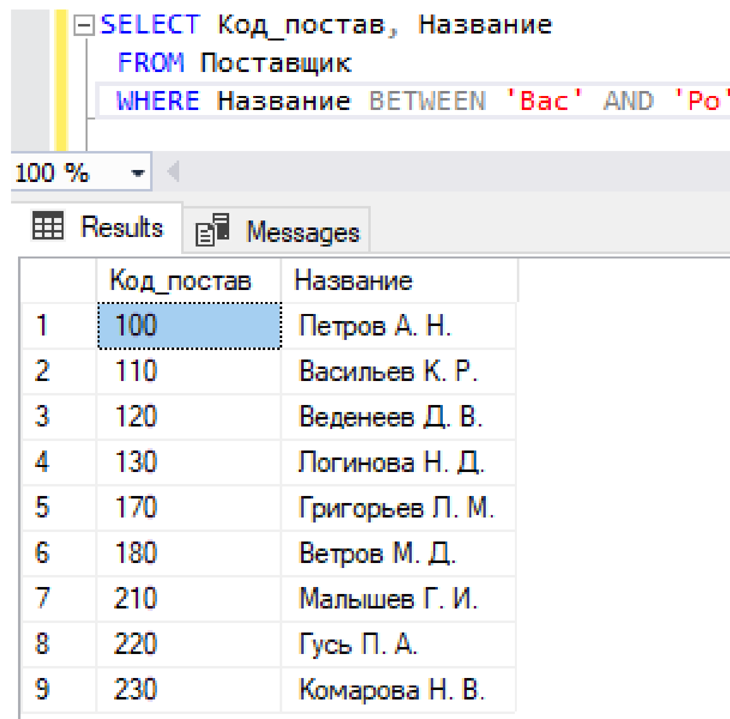
```
SELECT Код_постав, Название
FROM Поставщик
WHERE Название BETWEEN 'Вас' AND 'Ро
```

В данном примере значений 'Вас' и 'Ро' в базе данных нет. Однако, все поставщики, входящие в диапазон, в нижней части которого начало название поставщика совпадает с 'Вас' (т.е. выполняется условие "больше или равно"), а в верхней части - не более 'Ро', попадут в выборку.

Результат выполнения запроса приведен на рис. 3.12.

Отметим, что при выборке с использованием оператора BETWEEN поле, на которое накладывается диапазон, считается упорядоченным по возрастанию.

Разрешено также использовать конструкцию  NOT  BETWEEN, что позволяет получить выборку записей, указанные поля которых имеют значения меньше нижней границы и больше верхней границы.




| | Код_постав | Название |
|---|---|---|
| 1 | 100 | Петров А. Н. |
| 2 | 110 | Васильев К. Р. |
| 3 | 120 | Веденеев Д. В. |
| 4 | 130 | Логинова Н. Д. |
| 5 | 170 | Григорьев П. М. |
| 6 | 180 | Ветров М. Д. |
| 7 | 210 | Малышев Г. И. |
| 8 | 220 | Гусь П. А. |
| 9 | 230 | Комарова Н. В. |

Рис. 3.12. Команда SELECT с оператором BETWEEN с не реальными значениями нижней и верхней границ диапазона

Оператор IN проверяет, входит ли заданное значение, предшествующее ключевому слову “IN” (например, значение столбца или функция от него) в указанный в скобках список. Если заданное проверяемое значение равно какому-либо элементу в списке, то оператор принимает значение true**.** Например, вывести все записи таблицы Склад, у которых значение поля номенклатурный номер равно 10040, 10080, 10110:

```
SELECT *
FROM Склад
WHERE Ном_ном IN (10040, 10080, 10110)
```

Результат выполнения запроса приведен на рис. 3.13.

Разрешено также использовать конструкцию  NOT  IN.

```
SELECT *
FROM Склад
WHERE Ном_ном IN (10040,10080,10110)
```

100 %

Results | Messages

| | Ном_ном | Наименование | Цена | Количество |
|---|---|---|---|---|
| 1 | 10040 | Принтер HP | 6990,00 | 7 |
| 2 | 10080 | Монитор Samsung 28" | 16790,00 | 5 |
| 3 | 10110 | Мат. плата INTEL | 3408,00 | 35 |

Рис. 3.13. Команда SELECT с оператором IN

Оператор соответствия шаблону LIKE используется только с символьными данными. Он определяет, совпадает ли проверяемая символьная строка с заданным шаблоном. Шаблон может включать все разрешенные символы (с учетом верхнего и нижнего регистров), а также специальные символы-шаблоны:

% (символ процент) - замещает любое количество символов,

_ (символ подчеркивание) - замещает только один символ.

Например, получить список поставщиков, названия которых начинаются с буквы 'С' и содержать любое количество символов:

```
SELECT Название
FROM Поставщик
WHERE Название LIKE 'С%'
```

Результат выполнения запроса приведен на рис. 3.14.

```
SELECT Название
        FROM Поставщик
        WHERE Название LIKE 'С%'
```

100 %

Results | Messages

| | Название |
|---|---|
| 1 | Сталь Д. П. |
| 2 | Сабаев Т. Р. |

Рис. 3.14. Команда SELECT выводящая список поставщиков, названия которых начинаются с буквы 'С'

Команда

```
SELECT Название
FROM Поставщик
WHERE Название LIKE '_етров'
```

выполняет поиск и выдает все имена, состоящие из шести букв и заканчивающиеся сочетанием «етров» (Петров, Ветров и т.п.).

Результат выполнения запроса приведен на рис. 3.15.

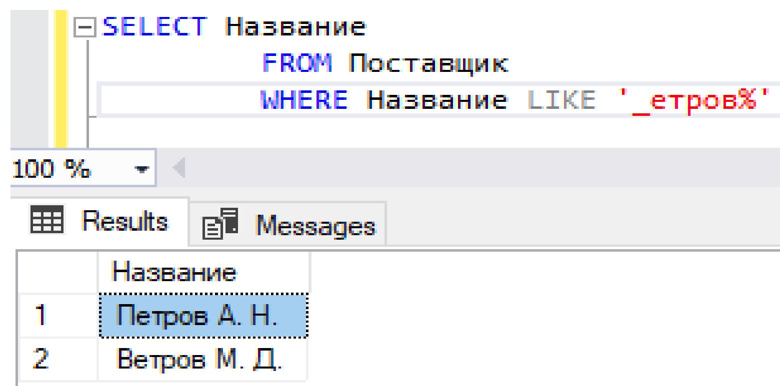



Рис. 3.15. Команда SELECT с оператором LIKE

В некоторых диалектах языка SQL, в том числе и Transact-SQL, поддерживаются еще два вида подстановки:

[] (квадратные скобки) - вместо символа строки будет подставлен один из возможных символов, указанный в диапазоне или наборе.

[^] - вместо соответствующего символа строки будут подставлены все символы, кроме указанных в диапазоне или наборе.

Рассмотрим примеры использования таких видов подстановки. Напишем запрос, с помощью которого выбираются названия регионов, начинающихся на любую букву в промежутке от В до К

```
SELECT Регион
FROM Поставщик
WHERE Регион LIKE '[В-К]%'
```

Результат выполнения запроса приведен на рис. 3.16.

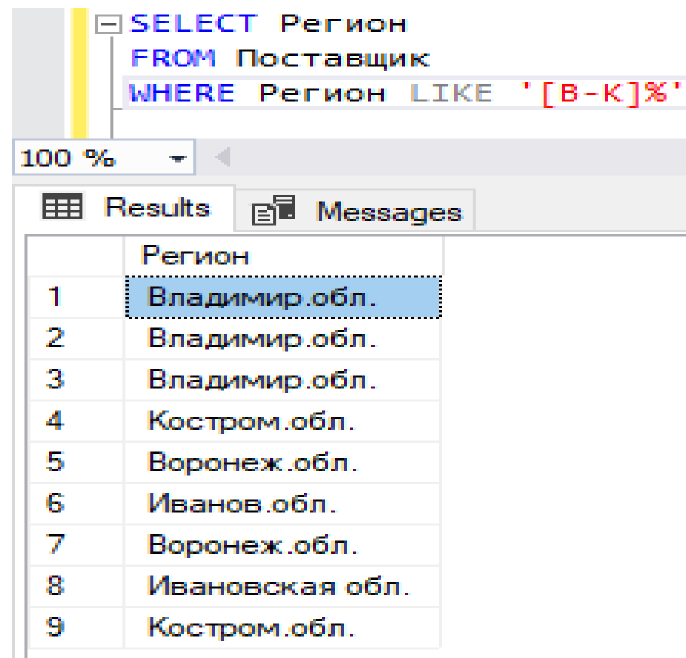



Рис. 3.16. Команда SELECT с оператором LIKE, использующий подстановку []

В следующем примере, команда

```
SELECT Регион
FROM Поставщик
WHERE Регион LIKE 'В[^л]%'
```

выполнит поиск и выдаст все названия поставщиков, начинающиеся на ‘В’, в которых вторая буква отличается от ‘л’.

Результат выполнения запроса приведен на рис. 3.17.

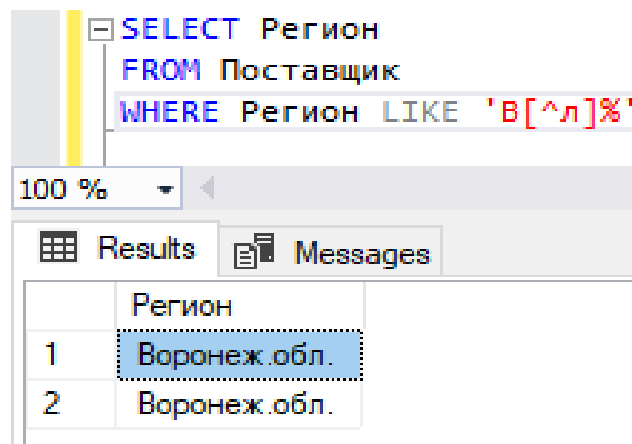



Рис. 3.17. Команда SELECT с оператором LIKE, использующий подстановку [^]

Разрешено также использовать конструкцию NOT LIKE.

Допускается использование нескольких условий отбора, которые объединяются логическими операторами AND, OR, NOT.

Если на складе необходимо найти монитор 28 дюймов любой модели с ценой от 15000 до 17000 рублей, то запрос может выглядеть так:

```
SELECT *
FROM Склад
WHERE Наименование LIKE 'Монитор%28_'
          AND Cena BETWEEN 15000 AND 17000
```

Результат выполнения запроса приведен на рисунке 3.18.

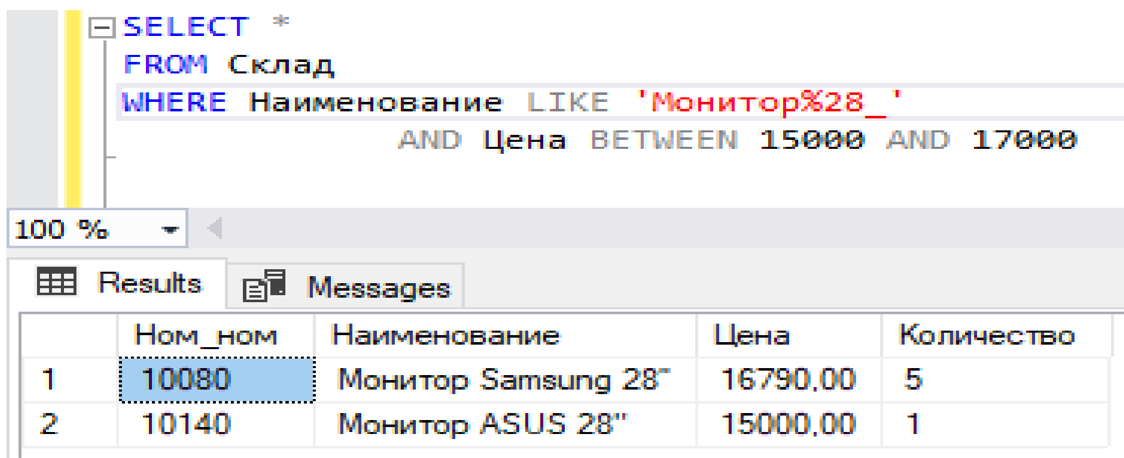



Рис. 3.18. Команда SELECT с двумя условиями отбора строк

Для выявления равенства значения некоторого атрибута неопределенному значению применяется операторы IS NULL или IS NOT NULL. Их форматы следующие:

<имя атрибута (поля)> IS [NOT ]NULL .

Если в данной строке указанный атрибут имеет неопределенное значение, то оператор IS NULL возвращает значение true, а оператор IS NOT NULL – false, иначе оператор IS NULL принимает значение false, а оператор IS NOT NULL – true.

Неопределенное значение интерпретируется в реляционной модели как значение, неизвестное на данный момент времени. Это значение в любой момент времени может быть заменено на некоторое конкретное значение. При сравнении неопределенных значений не действуют стандартные правила сравнения: одно неопределенное значение никогда не считается равным другому неопределенному значению.

Введение неопределенных значений вызвало необходимость модификации двузначной логики и превращение ее в трехзначную.

Все логические операции, проводимые с неопределенными значениями, подчиняются этой логике в соответствии со следующей таблицей истинности (табл. 3.4).

Таблица 3.4. Таблица истинности

| **A** | **B** | **NotA** | **A And B** | **A Or B** |
|---|---|---|---|---|
| True | True | False | True | True |
| True | False | False | False | True |
| True | Null | False | Null | True |
| False | True | True | False | True |
| False | False | True | False | False |
| False | Null | True | False | Null |
| Null | True | Null | Null | True |
| Null | False | Null | False | Null |
| Null | Null | Null | Null | Null |

#### 3.2.1.5. Группировка данных

При подготовке сложных выборок часто не обойтись без группировки данных. Группировка позволяет получить вычисляемую информацию о сводных характеристиках подгруппы таблицы. Например, сгруппировав данные в таблице Склад по полю Наименование, можно получить сведения о сумме товаров каждого наименования. Для того чтобы сгруппировать записи в запросе, используется предложение GROUP BY:

GROUP BY <Имя столбца (имя поля)>[,<Имя столбца (имя поля)>…]

С концептуальной точки зрения это предложение (группировать по) перекомпоновывает таблицу, представленную предложением FROM, в группы таким образом, чтобы в каждой группе все строки имели одно и то же значение поля, указанного в предложении GROUP BY. Это, конечно, не означает, что таблица физически перестраивается. Таким образом, в результате группировки все записи таблицы, для которых значения колонок совпадают, отображаются в выборке **единственной строкой**.

Если совместно с GROUP BY используется предложение WHERE, то оно обрабатывается первым, а группированию подвергаются только те строки, которые удовлетворяют условию поиска.

Кроме того, существуют ограничения на элементы списка возвращаемых столбцов. Все элементы этого списка должны иметь одно значение для каждой группы строк. Это означает, что возвращаемым столбцом может быть:

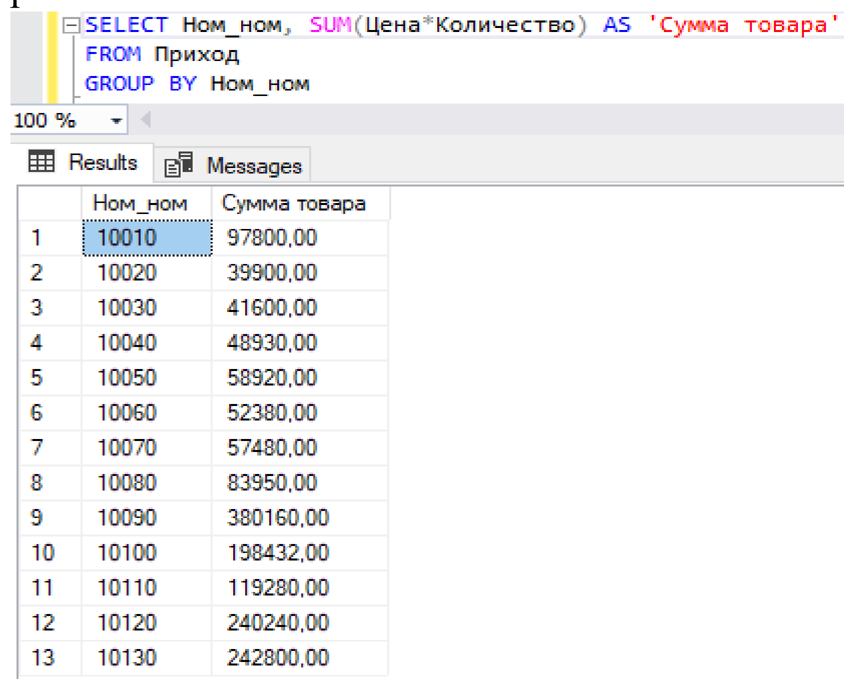
- константа;
- само поле, указанное в предложении GROUP BY;
- функция, которая подсчитывает какое-либо результирующее для данной группы значение. Поэтому в этом случае обычно используются функции AVG(), COUNT(), MAX(), MIN(), SUM().
- выражение, включающее в себя перечисленные выше элементы.

Например, определить из таблицы Приход общую сумму товара для каждого наименования

```
SELECT Ном_ном, SUM(Цена*Количество) AS 'Сумма товара'
FROM Приход
GROUP BY Ном_ном
```

Результат выполнения запроса приведен на рис. 3.19.

В результате выполнения предложения GROUP BY остаются только уникальные значения столбцов, по умолчанию отсортированные по возрастанию.




| | Ном_ном | Сумма товара |
|---|---|---|
| 1 | 10010 | 97800,00 |
| 2 | 10020 | 39900,00 |
| 3 | 10030 | 41600,00 |
| 4 | 10040 | 48930,00 |
| 5 | 10050 | 58920,00 |
| 6 | 10060 | 52380,00 |
| 7 | 10070 | 57480,00 |
| 8 | 10080 | 83950,00 |
| 9 | 10090 | 380160,00 |
| 10 | 10100 | 198432,00 |
| 11 | 10110 | 119280,00 |
| 12 | 10120 | 240240,00 |
| 13 | 10130 | 242800,00 |

Рис. 3.19. Команда SELECT с предложением GROUP BY

На практике в список возвращаемых столбцов запроса с группировкой всегда входят столбец группировки и агрегатная функция. Если последняя не указана, значит, запрос можно более просто выразить с помощью предиката DISTINCT без использования предложения GROUP BY. И наоборот, если не включить в результаты запроса столбец группировки, вы не сможете определить, к какой группе относится каждая строка результатов.

Еще одно ограничение запросов с группировкой обусловлено тем, что в SQL игнорируется информация о первичных и внешних ключах при анализе правильности запроса с группировкой. Рассмотрим следующий запрос: подсчитать общее количество поставок для каждого поставщика.

```
SELECT Документ.Код_постав, Название, COUNT(Документ.Код_постав)
FROM Документ, Поставщик
WHERE Документ.Код_постав = Поставщик.Код_постав
GROUP BY Документ.Код_поставщика
```

Зная природу данных, можно сказать, что запрос правильный, поскольку группировка по коду поставщика — фактически то же самое, что и группировка по наименованию поставщика. Говоря более точно, столбец группировки Код_постав является первичным ключом таблицы Поставщик, поэтому столбец Название должен иметь одно значение для каждой группы. Тем не менее, выдается сообщение об ошибке, представленное на рис. 3.20.

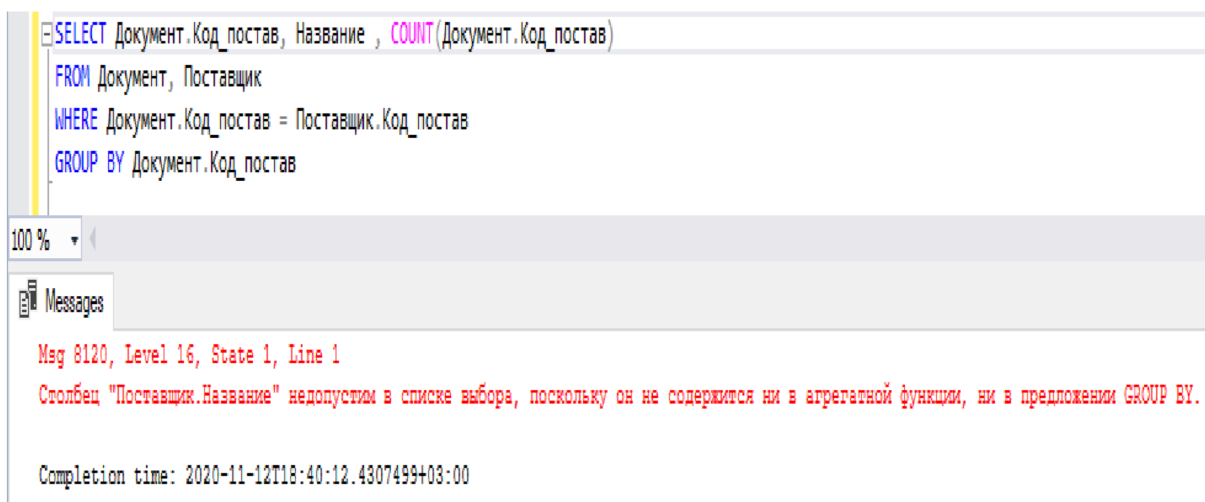
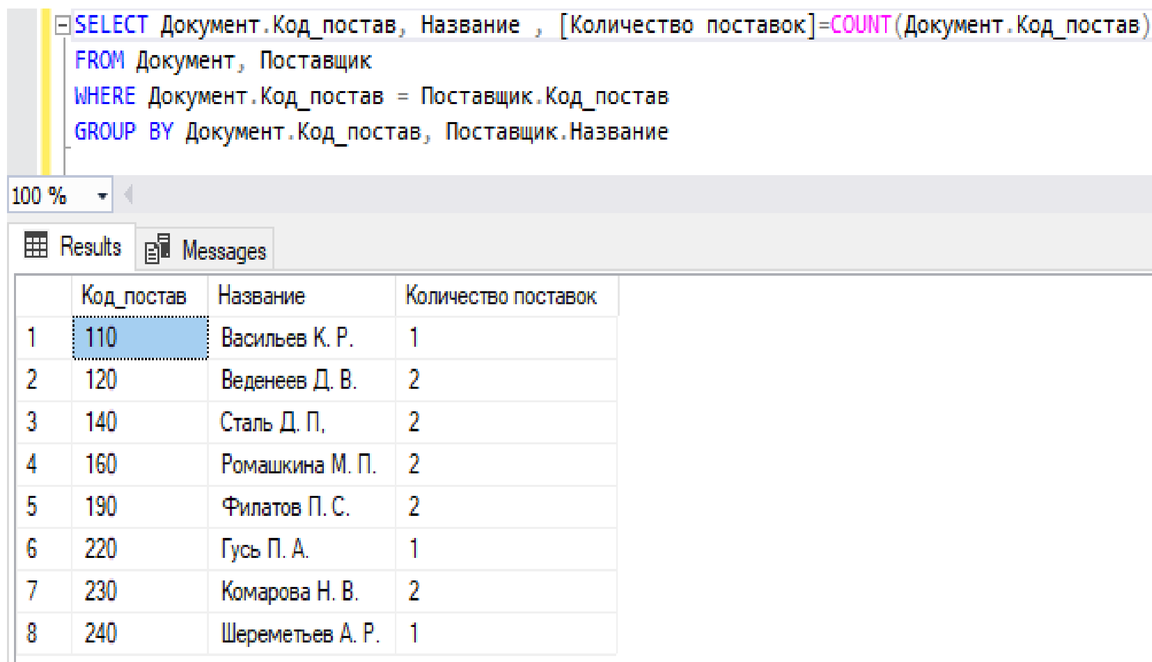



Рис. 3.20. Сообщение об ошибке

Чтобы решить эту проблему, можно в данном примере просто включить этот столбец в предложение GROUP BY:

SELECT Документ.Код_постав, Название, [Количество поставок] = COUNT(Документ. Код_постав)
FROM Документ, Поставщик
WHERE Документ.Код_постав = Поставщик.Код_постав
GROUP BY Документ.Код_поставщика, Поставщик. Название

Результат выполнения запроса приведен на рис. 3.21.




| | Код_постав | Название | Количество поставок |
|---|---|---|---|
| 1 | 110 | Васильев К. Р. | 1 |
| 2 | 120 | Веденеев Д. В. | 2 |
| 3 | 140 | Сталь Д. П, | 2 |
| 4 | 160 | Ромашкина М. П. | 2 |
| 5 | 190 | Филатов П. С. | 2 |
| 6 | 220 | Гусь П. А. | 1 |
| 7 | 230 | Комарова Н. В. | 2 |
| 8 | 240 | Шереметьев А. Р. | 1 |

Рис. 3.21. Команда SELECT с группировкой по двум полям

Конечно, если название поставщика в результатах запроса не требуется, можно вообще исключить его из списка возвращаемых столбцов:

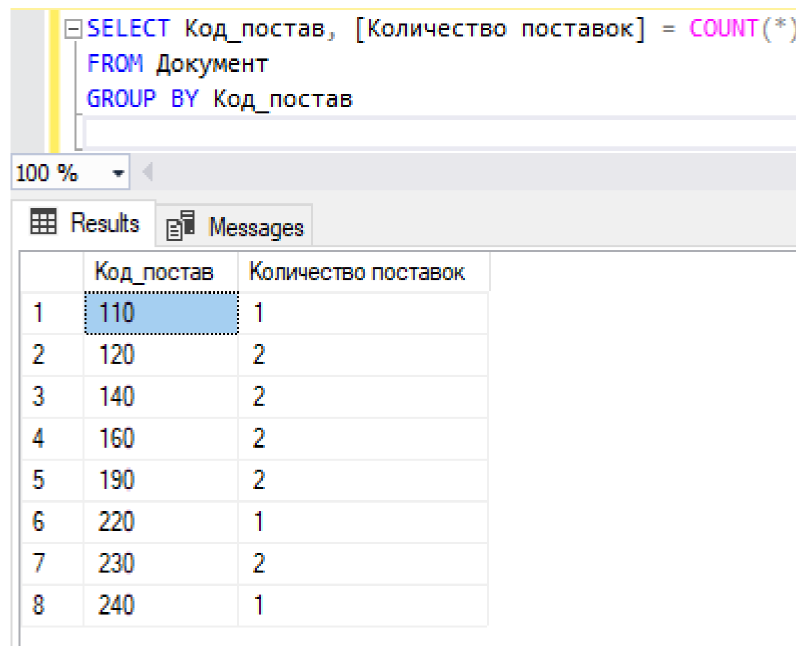
SELECT Код_постав, [Количество поставок] = COUNT(*)
FROM Документ
GROUP BY Код_постав

Результат выполнения запроса приведен на рис. 3.22.

Часто бывает нужным вывести не все промежуточные итоги, а только какую-либо их часть. Для этого применяется предложение

HAVING <условие>,

где <условие> – это логическое выражение, составленное из агрегатных функций и/или столбцов группировки.




| | Код_постав | Количество поставок |
|---|---|---|
| 1 | 110 | 1 |
| 2 | 120 | 2 |
| 3 | 140 | 2 |
| 4 | 160 | 2 |
| 5 | 190 | 2 |
| 6 | 220 | 1 |
| 7 | 230 | 2 |
| 8 | 240 | 1 |

Рис. 3.22. Команда SELECT с группировкой

Есть существенное отличие между предложениями HAVING и WHERE. Агрегатные функции можно включать только в предложение HAVING. В этом случае агрегатная функция вычисляется для каждой группы.

Например, определить мониторы, поступившие на общую сумму более 50000 руб. При этом, полагая, что один и тот же товар мог поступать неоднократно.

```
SELECT Ном_ном AS 'Номенклатурный номер', SUM(Цена*Количество) AS 'Сумма'
FROM Приход
WHERE Тип = 'Монитор'
GROUP BY Ном_ном
HAVING SUM(Цена*Количество) > 50000
```

Результат выполнения запроса приведен на рис. 3.23.

```
SELECT Ном_ном AS 'Номенклатурный номер',  SUM(Цена*Количество) AS 'Сумма'
FROM Приход
WHERE Тип='Монитор'
GROUP BY Ном_ном
HAVING SUM(Цена*Количество)>50000
```

|   | Номенклатурный номер | Сумма |
|---|---|---|
| 1 | 10050 | 58920,00 |
| 2 | 10060 | 52380,00 |
| 3 | 10070 | 57480,00 |
| 4 | 10080 | 83950,00 |

Рис. 3.23. Команда SELECT с предложениями GROUP BY и HAVING

Имейте в виду, что критерии, устанавливаемые с помощью WHERE, делают выборки, проверяя запись за записью, а предложение HAVING отбирает группы.

Предложение HAVING не может использоваться отдельно от предложения GROUP BY.

#### 3.2.1.6. Порядок вывода данных

Для вывода данных отсортированных по какому-либо столбцу, используется предложение ORDER BY. Это предложение имеет вид:

ORDER BY <Имя столбца | Номер столбца [ASC | ESC]>[,<Имя столбца | Номер столбца> [ASC | DESC]…]

Способ упорядочивания определяется дополнительными зарезервированными словами ASC и DESC. Способом по умолчанию - если ничего не указано - является упорядочивание “по возрастанию” (ASC). Если же указано слово DESC, то упорядочивание будет производиться “по убыванию”.

Подчеркнем еще раз, что *предложение* ORDER BY *должно указываться в самом конце запроса*.

Порядок строк может задаваться одним из двух способов:

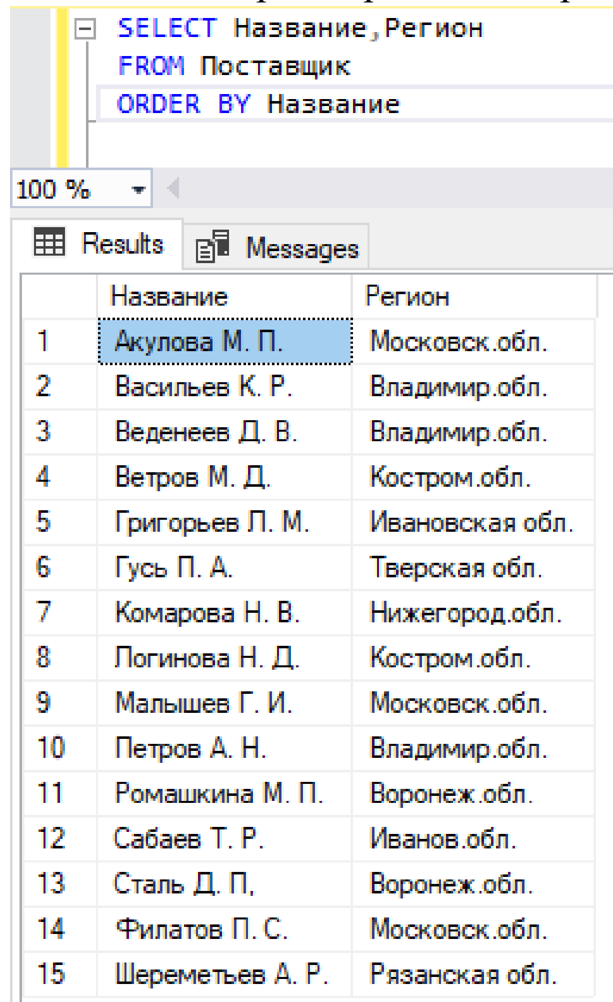
- именами столбцов
- номерами столбцов.

*Упорядочивание с использованием имен столбцов*

Например, получить список поставщиков, упорядоченный по их названиям в алфавитном порядке

```
SELECT Название, Регион
FROM Поставщик
ORDER BY Название
```

Результат выполнения запроса приведен на рис. 3.24.




| | Название | Регион |
|---|---|---|
| 1 | Акулова М. П. | Московск.обл. |
| 2 | Васильев К. Р. | Владимир.обл. |
| 3 | Веденеев Д. В. | Владимир.обл. |
| 4 | Ветров М. Д. | Костром.обл. |
| 5 | Григорьев П. М. | Ивановская обл. |
| 6 | Гусь П. А. | Тверская обл. |
| 7 | Комарова Н. В. | Нижегород.обл. |
| 8 | Логинова Н. Д. | Костром.обл. |
| 9 | Малышев Г. И. | Московск.обл. |
| 10 | Петров А. Н. | Владимир.обл. |
| 11 | Ромашкина М. П. | Воронеж.обл. |
| 12 | Сабаев Т. Р. | Иванов.обл. |
| 13 | Сталь Д. П, | Воронеж.обл. |
| 14 | Филатов П. С. | Московск.обл. |
| 15 | Шереметьев А. Р. | Рязанская обл. |

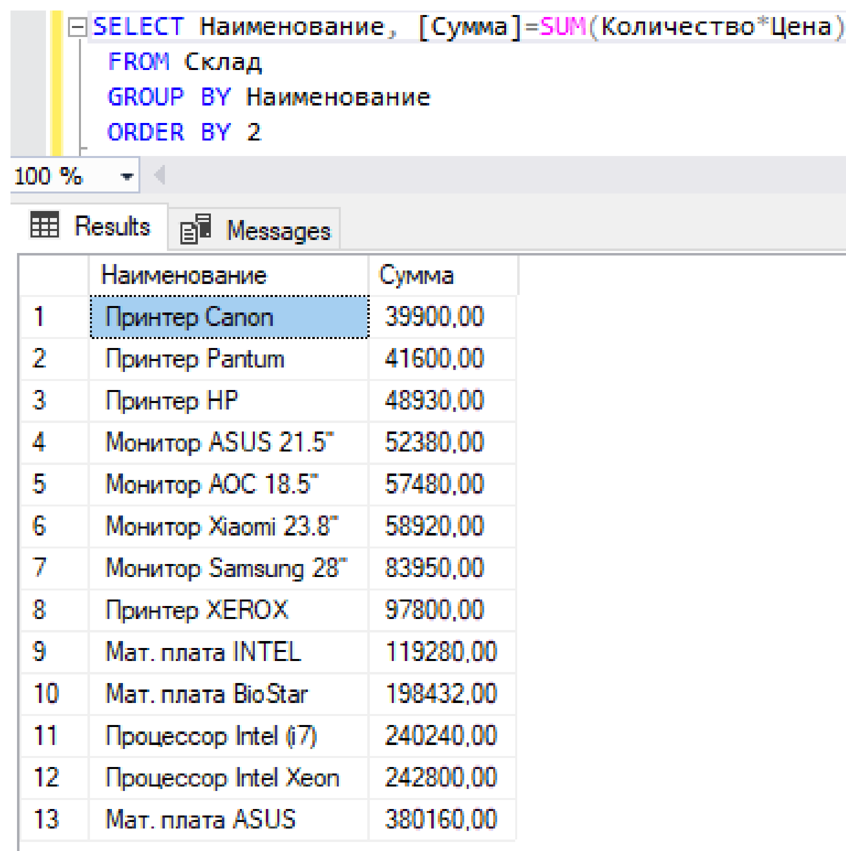
Рис. 3.24. Команда SELECT с сортировкой списка поставщиков

*Упорядочивание с использованием номеров столбцов*

Например, сделаем выборку наименований товаров на складе и сумм, упорядоченную по суммам.

```
SELECT Наименование, [Сумма] = SUM(Количество*Цена)
FROM Склад
GROUP BY Наименование
ORDER BY 2
```

Результат выполнения запроса приведен на рис. 3.25.




| | Наименование | Сумма |
|---|---|---|
| 1 | Принтер Canon | 39900,00 |
| 2 | Принтер Pantum | 41600,00 |
| 3 | Принтер HP | 48930,00 |
| 4 | Монитор ASUS 21.5" | 52380,00 |
| 5 | Монитор AOC 18.5" | 57480,00 |
| 6 | Монитор Xiaomi 23.8" | 58920,00 |
| 7 | Монитор Samsung 28" | 83950,00 |
| 8 | Принтер XEROX | 97800,00 |
| 9 | Мат. плата INTEL | 119280,00 |
| 10 | Мат. плата BioStar | 198432,00 |
| 11 | Процессор Intel (i7) | 240240,00 |
| 12 | Процессор Intel Xeon | 242800,00 |
| 13 | Мат. плата ASUS | 380160,00 |

Рис. 3.25. Команда SELECT с сортировкой сумме

Допускается использование нескольких уровней вложенности при упорядочивании выводимой информации по столбцам. При этом разрешается смешивать оба способа задания порядка строк.

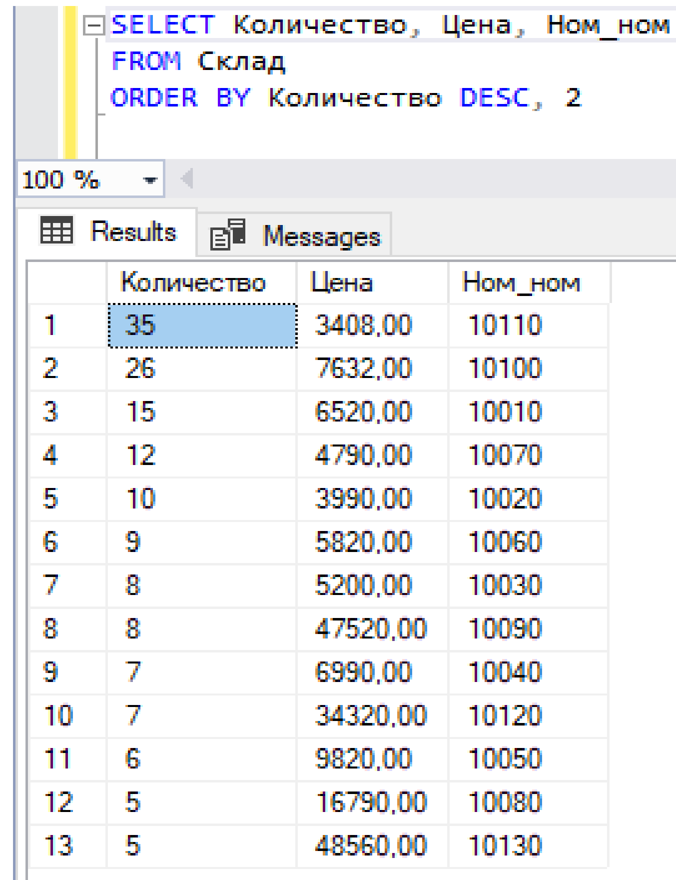
Вывести список товаров на складе, упорядоченный по убыванию их количества, а в пределах одинакового количества - по возрастанию цены:

```
SELECT Количество, Цена, Ном_ном
FROM Склад
ORDER BY Количество DESC, 2
```

Результат выполнения запроса приведен на рис. 3.26.

Столбец, определяющий порядок вывода строк, не обязательно должен присутствовать в списке выбираемых столбцов. Например, команда

SELECT Название
FROM Поставщик
ORDER BY Рейтинг
позволяет получить список поставщиков, отсортированный по их рейтингу.



```
SELECT Количество, Цена, Ном_ном
FROM Склад
ORDER BY Количество DESC, 2
```

| | Количество | Цена | Ном_ном |
|---|---|---|---|
| 1 | 35 | 3408,00 | 10110 |
| 2 | 26 | 7632,00 | 10100 |
| 3 | 15 | 6520,00 | 10010 |
| 4 | 12 | 4790,00 | 10070 |
| 5 | 10 | 3990,00 | 10020 |
| 6 | 9 | 5820,00 | 10060 |
| 7 | 8 | 5200,00 | 10030 |
| 8 | 8 | 47520,00 | 10090 |
| 9 | 7 | 6990,00 | 10040 |
| 10 | 7 | 34320,00 | 10120 |
| 11 | 6 | 9820,00 | 10050 |
| 12 | 5 | 16790,00 | 10080 |
| 13 | 5 | 48560,00 | 10130 |

Рис. 3.26. Команда SELECT с двух уровневой сортировкой

## Вопросы для самопроверки

1. Какие служебные слова обязательно присутствуют в команде SELECT?
2. Какие служебные слова могут отсутствовать в команде SELECT?
3. После какого служебного слова в команде SELECT указывается выбор столбцов?
4. Как исключить повторяющиеся при выводе записи?
5. После какого служебного слова в команде SELECT указывается выбор строк?

6. Какие операнды могут использоваться при формировании условия выборки записей?

7. Какие символы-заменители могут использоваться в шаблоне?

8. Какие агрегатные функции могут использоваться в запросах?

9. В каких случаях используется группировка выводимых записей?

10. Как задать сортировку по нескольким полям?

**Практические задания**

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 запрос, который выводит заказчиков, в фамилиях которых третья буква - "к".
- Напишите запрос, который сосчитает сумму покупок на текущую дату.
- Напишите запрос, который находит средний рейтинг заказчиков в каждом городе.
- Напишите запрос, который выводит все покупки со значениями суммы выше, чем 1000 руб.
- Напишите запрос к таблице ЗАКАЗЧИК, который мог бы найти высший рейтинг заказчика в каждом городе.
- Напишите запрос, который выводил бы список заказчиков в нисходящем порядке их рейтинга. Вывод поля рейтинга должен сопровождаться именем заказчика и его номером.
- Напишите запрос, который бы выводил общую сумму покупок на каждый день.

### 3.2.2. Выборка данных из нескольких таблиц

При обсуждении предыдущих примеров мы, как правило, использовали выборки из одной таблицы. На практике запросы к одной таблице составляют не более 10 % от общего количества запросов.

Для вывода *связанной* информации, хранящейся в нескольких таблицах, в языке SQL используется *операция соединения*. *Операция*

*соединения* позволяет соединять по некоторому условию строки из более чем одной таблицы для образования новых строк данных.

Соединение выполняется путем подбора строк с одинаковыми значениями в общих для нескольких таблиц столбцах. Необходимо определить одну из таблиц так, чтобы один из столбцов первой таблицы дублировался во второй, тогда этот столбец будет общим для обеих таблиц (эти столбцы, как мы уже знаем, называются внешними ключами).

Можно объединять и отображать строки разных таблиц и манипулировать данными  с помощью тех же команд, которые используются при работе с одной таблицей.

В этом проявляется одна из наиболее важных особенностей запросов SQL - способность определять связи между многочисленными таблицами и выводить информацию из них в рамках этих связей. Именно эта операция придает гибкость и легкость языку SQL.

Операции соединения подразделяются на два вида - внутренние и внешние. Внешние соединения поддерживаются стандартом SQL - 92 и содержат зарезервированное слово “JOIN”, в то время как внутренние соединения (или просто соединения) могут задаваться как без использования такого слова (в стандарте SQL-89), так и с использованием слова “JOIN” (в стандарте SQL -92).

#### 3.2.2.1. Внутреннее соединение

Внутреннее соединение возвращает только те строки, для которых условие соединения принимает значение true.

*Соединение с помощью предложения WHERE*

Общий синтаксис соединения двух таблиц следующий:

SELECT <список столбцов>
FROM <Имя таблицы1>, <Имя таблицы2>
WHERE [<Имя таблицы1>.] <Имя столбца> <Оператор объединения> [<Имя таблицы2>.] <Имя столбца>

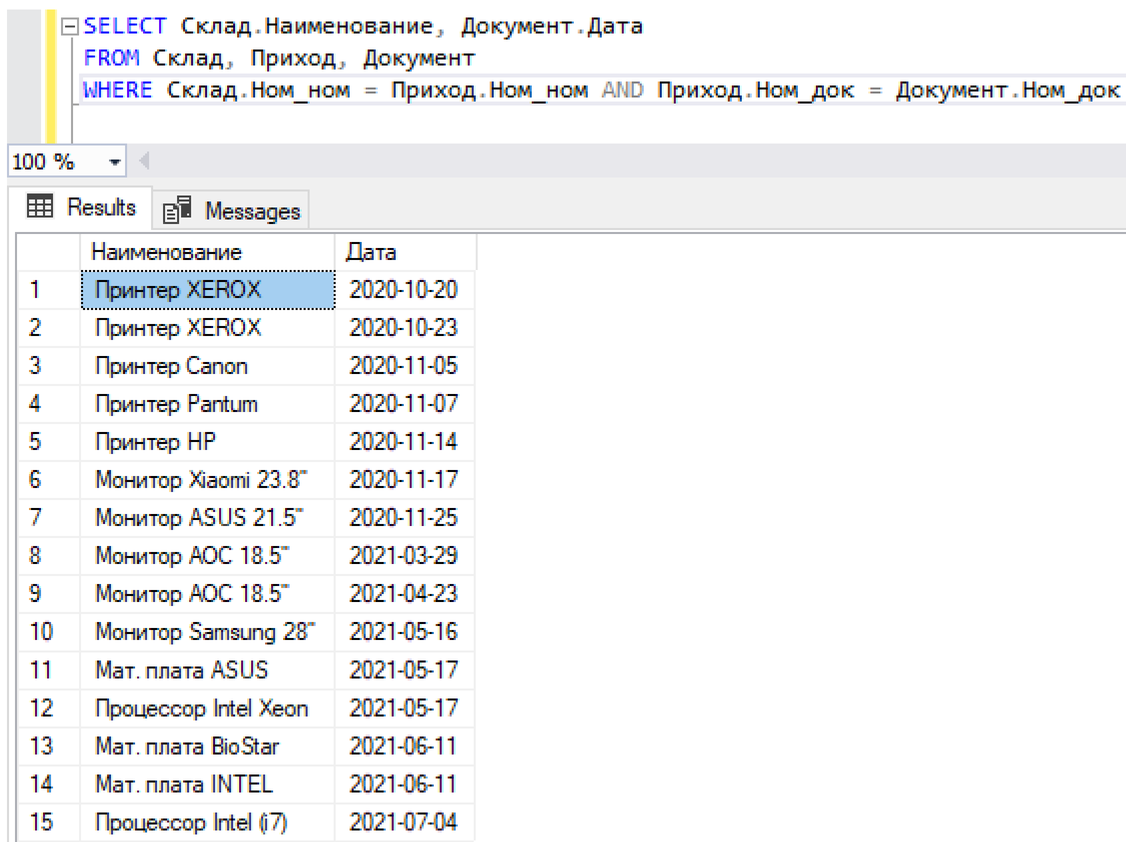
Предложение FROM должно включать две таблицы, а столбцы, указанные в предложении WHERE, должны быть совместимы. Если столбцы таблиц имеют одинаковые имена, то перед ними необходимо указать названия таблиц с точкой. Обычно при связывании двух и бо-

лее таблиц используется оператор «=», но можно задействовать и другие операторы.

Рассмотрим пример выборки информации из нескольких таблиц. Пусть нам необходимо вывести наименование материалов и товаров вместе с датой их поступления на склад. Наименование товара хранится в таблице Склад, а дата поступления товара – в таблице Документ. Причем в этих таблицах нет одинаковых столбцов и, следовательно, соединить их напрямую нельзя. Однако таблицы Склад и Документ можно соединить, используя таблицу Приход:

```
SELECT Склад.Наименование, Документ. Дата
FROM Склад, Приход, Документ
WHERE Склад.Ном_ном = Приход.Ном_ном AND
        Приход.Ном_док = Документ. Ном_док
```

Результат выполнения запроса приведен на рис. 3.27.



```
SELECT Склад.Наименование, Документ.Дата
FROM Склад, Приход, Документ
WHERE Склад.Ном_ном = Приход.Ном_ном AND Приход.Ном_док = Документ.Ном_док
```



| | Наименование | Дата |
|---|---|---|
| 1 | Принтер XEROX | 2020-10-20 |
| 2 | Принтер XEROX | 2020-10-23 |
| 3 | Принтер Canon | 2020-11-05 |
| 4 | Принтер Pantum | 2020-11-07 |
| 5 | Принтер HP | 2020-11-14 |
| 6 | Монитор Xiaomi 23.8" | 2020-11-17 |
| 7 | Монитор ASUS 21.5" | 2020-11-25 |
| 8 | Монитор AOC 18.5" | 2021-03-29 |
| 9 | Монитор AOC 18.5" | 2021-04-23 |
| 10 | Монитор Samsung 28" | 2021-05-16 |
| 11 | Мат. плата ASUS | 2021-05-17 |
| 12 | Процессор Intel Xeon | 2021-05-17 |
| 13 | Мат. плата BioStar | 2021-06-11 |
| 14 | Мат. плата INTEL | 2021-06-11 |
| 15 | Процессор Intel (i7) | 2021-07-04 |

Рис. 3.27. Внутреннее соединение

При написании запросов часто используют псевдонимы таблиц - временное имя таблицы.

Псевдонимы таблиц используются для сокращения SQL-кода, чтобы его было легче читать или когда вы выполняете самосоединение (т.е. перечисление одной и той же таблицы более одного раза в операторе FROM).

Синтаксис псевдонима таблицы в SQL:

Имя таблицы [AS] Псевдоним,

где Имя таблицы – оригинальное имя таблицы, которой вы хотите указать псевдоним; Псевдоним – псевдоним для назначения.

Рассмотренный выше запрос, но с использованием псевдонимов таблиц выглядит следующим образом

```
SELECT s.Наименование, d.Дата
FROM Склад s, Приход p, Документ d
WHERE s.Ном_ном = p.Ном_ном AND p.Ном_док = d. Ном_Док
```

Результат выполнения запроса приведен на рис. 3.27.

Псевдоним действителен только в рамках команды SQL.

Как видно из примера, для того чтобы связать две таблицы, мы использовали два столбца, по одному из каждой таблицы. Связующие столбцы имеют один и тот же тип данных. В рассмотренном примере совпадает все, вплоть до названия.

*Соединение с использованием опции JOIN*

В SQL-Server для соединения таблиц можно использовать предложение INNER JOIN. Стандарт соединения с использованием опции JOIN:

- в предложении FROM слева и справа от фразы INNER JOIN указываются соединяемые таблицы;
- условия соединения помещаются в предложение ON.

Для выше рассмотренного примера запрос с использованием фразы INNER JOIN выглядит следующим образом:

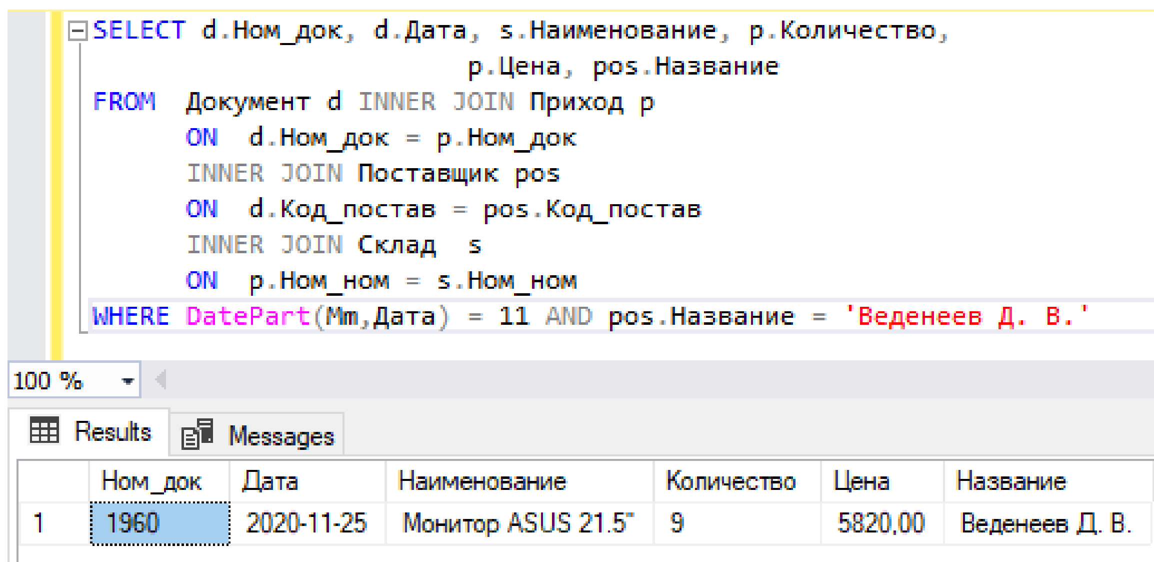
```
SELECT Наименование, Дата
FROM Склад s INNER JOIN Приход p
ON s.Ном_ном = p.Ном_ном
 INNER JOIN Документ d
ON p.Ном_док = d.Ном_док
```

Результаты этого запроса представлены на рис. 3.27

Приведем пример выборки товаров, поступивших на склад в июле месяце от поставщика Веденеева Д.В.. Такой запрос имеет вид:

SELECT d. Ном_док, d.Дата, s.Наименование, p.Количество, p.Цена, pos. Название
 FROM Документ d INNER JOIN Приход p
 ON d. Ном_док = p. Ном_док
 INNER JOIN Поставщик pos
 ON d.Код_постав = pos.Код_постав
 INNER JOIN Склад s
 ON p. Ном_ном = s. Ном_ном
WHERE DatePart(Mm,Дата) = 7 AND pos. Название = ‘Веденеев Д.В.’

Результат выполнения запроса приведен на рис. 3.28.




| | Ном_док | Дата | Наименование | Количество | Цена | Название |
|---|---|---|---|---|---|---|
| 1 | 1960 | 2020-11-25 | Монитор ASUS 21.5" | 9 | 5820,00 | Веденеев Д. В. |

Рис. 3.28. Запрос, использующий 4 таблицы базы данных

В данном примере создадим запрос об итоговом количестве и об итоговой сумме каждого поступившего на склад товара с 20 марта 2020 года по 24 апреля 2021 года:

SELECT p.Ном_ном, SUM(p.Количество) AS Количество,
 SUM(p.Количество * p.Цена) AS Сумма;
FROM Приход p INNER JOIN Документ d
 ON p. Ном_док = d. Ном_док
 WHERE d.Дата BETWEEN ‘2021-03-20’ AND ‘2021-04-24’
 GROUP BY p.Ном_ном

Результат выполнения запроса приведен на рис. 3.29.

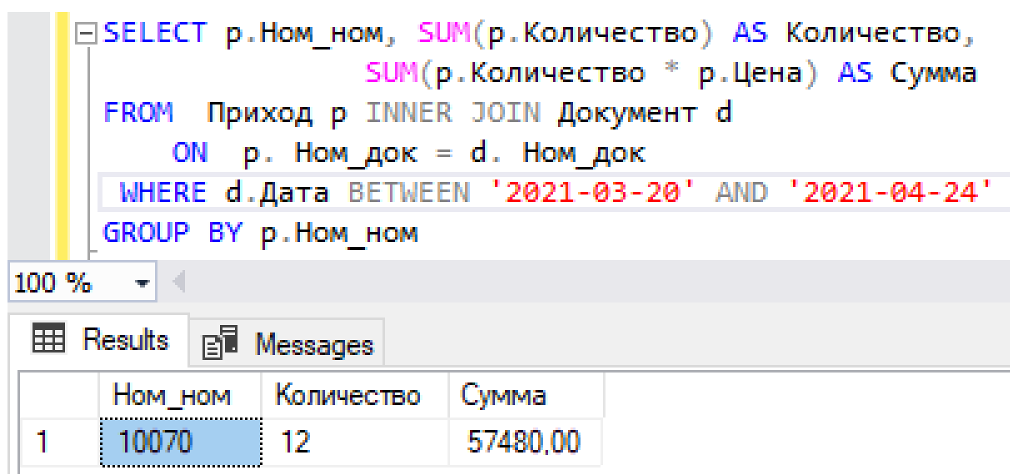



Рис. 3.29. Запрос, использующий соединение таблиц и группировку

При создании условий соединения необходимо иметь ввиду следующее:

- Если в запрос включены две таблицы, а условия соединения не указаны, каждая запись из одной таблицы связывается с каждой записью из другой, и результат имеет число записей, равное произведению числа записей во всех таблицах, участвующих в запросе. Такой запрос может занять много времени.
- Нельзя связывать две таблицы по столбцам со значениями NULL, так как пустое поле одной таблицы соответствует пустому полю другой таблицы.

### 3.2.2.2. Внешнее соединение

Внешнее соединение возвращает все строки из одной таблицы и только те строки из другой таблицы, для которых условие соединения принимает значение true. Строки второй таблицы, не удовлетворяющие условию соединения (т.е. имеющие значение false), получают значение NULL в результирующем наборе.

Рассмотрим три вида внешнего соединения**:** LEFT JOIN, RIGHT JOIN и FULL JOIN.

В левом соединении (LEFT JOIN) запрос возвращает все строки из левой таблицы (т.е. таблицы, стоящей *слева* от зарезервированного словосочетания LEFT JOIN) и только те из правой таблицы, которые удовлетворяют условию соединения. Если же в правой таблице не найдется строк, удовлетворяющих заданному условию, то в результате они замещаются значениями NULL.

Например, даны две таблицы Tabl1 и Tabl2

Tabl1

| X | Y |
|---|---|
| 0 | 5 |
| 1 | 6 |
| 3 | 10 |

Tabl2

| X | Z |
|---|---|
| 0 | 3 |
| 1 | 4 |
| 2 | 5 |

В результате запроса

SELECT Y, Z FROM Tabl1 LEFT JOIN Tabl2 ON Tabl1.X = Tabl2.X

получим на экране следующую таблицу

| Y | Z |
|---|---|
| 5 | 3 |
| 6 | 4 |
| 10 | NULL |

Правое соединение (RIGHT JOIN) создает соединение, в котором выбираются все записи из правой таблицы, а также записи из левой таблицы, значения поля связи которого совпадают со значениями поля связи правой таблицы. Если же в левой таблице не найдется строк, удовлетворяющих заданному условию, то в результате они замещаются значениями NULL.

Для таблиц Tabl1 и Tabl2 сделаем правое соединение

SELECT Y, Z FROM Tabl1 RIGHT JOIN Tabl2 ON Tabl1.X = Tabl2.X

В результате получим таблицу

| Y | Z |
|---|---|
| 5 | 3 |
| 6 | 4 |
| NULL | 5 |

Использование в запросе опции FULL JOIN создает соединение, в котором выбираются все записи из правой и левой таблицы. Например, полное объединение таблиц Tabl1 и Tabl2

SELECT Y, Z FROM Tabl1 FULL JOIN Tabl2 ON Tabl1.X = Tabl2.X

сформирует таблицу

| Y | Z |
|---|---|
| 5 | 3 |
| 6 | 4 |
| 10 | NULL |
| NULL | 5 |

#### 3.2.2.3. Объединение выборок

Оператор UNION объединяет результаты выполнения одной команды SELECT с результатами другой команды. Он удобен, если требуется просмотреть аналогичные данные из разных таблиц. Его синтаксис:

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

По умолчанию оператор UNION удаляет повторяющиеся записи. Предложение ALL запрещает удалять повторяющиеся записи из результата. Правила использования оператора UNION:

- Все команды SELECT должны возвращать одинаковое количество столбцов в результате выполнения запроса.
- Столбцы должны иметь одинаковые типы данных и размерность.
- Только последняя команда SELECT может иметь предложение ORDER BY.

Пример, получить данные о товарах в таблицах Приход и Склад

```
SELECT Ном_ном, Цена
FROM Приход
    UNION ALL
 SELECT Ном_ном, Цена
 FROM Склад
 Order BY 1
```

Результат выполнения запроса приведен на рис. 3.30

```
SELECT Ном_ном, Цена
FROM Приход
    UNION ALL
 SELECT Ном_ном, Цена
 FROM Склад
 Order BY 1
```

100 %

Results | Messages

| | Ном_ном | Цена |
|---|---|---|
| 1 | 10010 | 6520,00 |
| 2 | 10010 | 6520,00 |
| 3 | 10010 | 6520,00 |
| 4 | 10020 | 3990,00 |
| 5 | 10020 | 3990,00 |
| 6 | 10030 | 5200,00 |
| 7 | 10030 | 5200,00 |
| 8 | 10040 | 6990,00 |
| 9 | 10040 | 6990,00 |
| 10 | 10050 | 9820,00 |
| 11 | 10050 | 9820,00 |
| 12 | 10060 | 5820,00 |
| 13 | 10060 | 5820,00 |
| 14 | 10070 | 4790,00 |
| 15 | 10070 | 4790,00 |
| 16 | 10070 | 4790,00 |
| 17 | 10080 | 16790,00 |
| 18 | 10080 | 16790,00 |
| 19 | 10090 | 47520,00 |

Рис. 3.30. Запрос с оператором UNION

В данном примере рассмотрим две таблицы:

АВТОР (Фамилия, Имя, Район);

СЛУЖАЩИЙ (Фамилия, Имя, Район, Должность).

В результате следующего запроса будут найдены все авторы и служащие, живущие в Октябрьском районе.

```
SELECT Имя, Фамилия, Район, ‘Автор’
FROM АВТОР
WHERE Район = ‘Октябрьский’
UNION
SELECT Имя, Фамилия, Район, Должность
FROM СЛУЖАЩИЙ
WHERE Район = ‘Октябрьский’
ORDER BY 1
```

Обратите внимание, что поскольку в примере указывается еще и должность сотрудников, в первый запрос пришлось включить еще один столбец, без которого было бы получено сообщение об ошибке.

Оператор UNION можно использовать в качестве оператора IF для отображения различных значений одного поля в зависимости от значений в других полях. Без оператора UNION для получения аналогичного результата потребовалось бы выполнение нескольких запросов.

Пусть требуется получить список товаров, указав для каждого из них процентное снижение цены и новую цену. При этом цена товаров до 5000 руб. снижается на 20%, цена товаров между 5000 руб. и 30000 руб. снижается на 10%, цена товаров больше 30000 руб. снижается на 30%.

```
SELECT ‘на 20%’, Наименование AS ‘Наименование’, Цена AS ‘Старая цена’, Цена * 0.8 AS ‘Новая цена’
FROM Склад
WHERE Цена < 5000
UNION
SELECT ‘на 10%’, Наименование AS ‘Наименование’, Цена AS ‘Старая цена’, Цена * 0.9 AS ‘Новая цена’
FROM Склад
WHERE (Цена >=5000) AND (Цена <= 30000)
UNION
SELECT ‘на 30%’, Наименование AS ‘Наименование’, Цена AS ‘Старая цена’, Цена * 0.7 AS ‘Новая цена’
FROM Склад
WHERE Цена > 30000
```

Результат выполнения запроса приведен на рис. 3.31.

## Вопросы для самопроверки

1. Какая операция используется в языке SQL для выборки данных из нескольких таблиц?
2. Как осуществляется внутреннее соединение таблиц?
3. Какие способы записи многотабличных запросов Вы знаете?
4. Что получится в результате запроса, если не указать условие соединения?
5. Сколько условий соединения должно присутствовать в запросе?
6. Какие виды внешнего соединения Вы знаете?
7. Для чего в запросах используется оператор UNION?

```
SELECT [Снижается]='на 20%', Наименование, Цена AS 'Старая цена', Цена * 0.8 AS 'Новая цена'
FROM Склад
WHERE Цена < 5000
UNION
SELECT [Снижается]='на 10%', Наименование, Цена AS 'Старая цена', Цена * 0.9 AS 'Новая цена'
FROM Склад
WHERE (Цена >=5000) AND (Цена <= 30000)
UNION
SELECT [Снижается]='на 30%', Наименование, Цена AS 'Старая цена', Цена * 0.7 AS 'Новая цена'
FROM Склад
WHERE Цена > 30000
```

100 %

Results | Messages

| | Снижается | Наименование | Старая цена | Новая цена |
|---|---|---|---|---|
| 1 | на 10% | Мат. плата BioStar | 7632,00 | 6868.80000 |
| 2 | на 10% | Монитор ASUS 21.5" | 5820,00 | 5238.00000 |
| 3 | на 10% | Монитор Samsung 28" | 16790,00 | 15111.00000 |
| 4 | на 10% | Монитор Xiaomi 23.8" | 9820,00 | 8838.00000 |
| 5 | на 10% | Принтер HP | 6990,00 | 6291.00000 |
| 6 | на 10% | Принтер Pantum | 5200,00 | 4680.00000 |
| 7 | на 10% | Принтер XEROX | 6520,00 | 5868.00000 |
| 8 | на 20% | Мат. плата INTEL | 3408,00 | 2726.40000 |
| 9 | на 20% | Монитор AOC 18.5" | 4790,00 | 3832.00000 |
| 10 | на 20% | Принтер Canon | 3990,00 | 3192.00000 |
| 11 | на 30% | Мат. плата ASUS | 47520,00 | 33264.00000 |
| 12 | на 30% | Процессор Intel i7 | 34320,00 | 24024.00000 |
| 13 | на 30% | Процессор Intel Xeon | 48560,00 | 33992.00000 |

Рис. 3.31. Оператор UNION как аналог оператора IF

8. Какие правила использования оператора UNION Вы знаете?

9. Как объединить результаты двух или более команд SELECT без исключения повторяющихся строк?

## Практические задания

1. Даны две таблицы:

СТУДЕНТЫ( ФИО, Дата рождения, Номер группы, Номер зачетной книжки)

ЭКЗАМЕНЫ(Номер зачетной книжки, Дисциплина, Оценка, Зачет).

Напишите запрос, который выводит список студентов, имеющих задолженности (имеет хотя бы одну двойку или нет зачета).

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 запрос для определения списка продавцов, у которых делал покупку Петров.
- Создайте объединение из двух запросов, которое показало бы имена, города и рейтинги всех заказчиков. Те из них, которые имеют поле рейтинг =200 и более, должны, кроме того, иметь слова – « Высокий рейтинг», а остальные должны иметь слова «Низкий рейтинг».

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 запрос для вывода названия и численности населения городов, получавших груз весом более 1000 кГ.
- Какие города в базе данных имеют наибольшую и наименьшую численность населения ?

4. Дана база 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 запрос для получения списка клиентов, проживающих в настоящее время в гостинице в одноместных номерах.

### 3.2.3. Подзапросы

Для создания более сложных запросов можно использовать вложенные запросы (подзапросы).

Подзапросом называется команда SELECT, которая находится внутри другой команды языка манипулирования данными.

Именно возможность вложения команд SQL друг в друга является причиной, по которой SQL получил свое название — Structured Query Language (структурированный язык запросов). Каждая вклю-

чающая команда - следующий по старшинству уровень в подзапросе - представляет собой внешний уровень для внутреннего подзапроса.

Это достаточно сложная и интересная тема, поскольку существует два типа обработки подзапроса (коррелированный и некоррелированный) и три возможности соединения подзапроса с внешним предложением. Но давайте рассмотрим все по порядку.

Подзапросы можно разделить на следующие группы в зависимости от возвращаемых результатов:

- Скалярные – это подзапросы, возвращающие единственное значение.
- Векторные – подзапросы, возвращающие от 0 до нескольких однотипных элементов (список элементов).
- Табличные – это подзапросы, возвращающие таблицу.

Сначала рассмотрим использование подзапросов в команде SELECT. Подзапросы могут располагаться в разных частях этой команды:

в предложении SELECT – скалярные;

в предложении FROM – табличные некоррелированные;

в предложении WHERE – любые;

в предложении HAVING – любые.

Достаточно упрощенный синтаксис подзапроса, иллюстрирующий вложение подзапроса в предложении WHERE команды SELECT, выглядит следующим образом:

```
SELECT   [DISTINCT] <список_выбора_запроса>
 FROM <список_таблиц>
WHERE <выражение>   [NOT]   IN /   оператор_сравнения
[ANY   /   ALL] / [NOT]   EXISTS
( SELECT   [DISTINCT]   <список_выбора_подзапроса>
FROM <список_таблиц >
WHERE  <условия>)
```

Обращаем внимание, что текст подзапроса заключается в круглые скобки.

Как же работают подзапросы? Очень просто — они возвращают результаты внутреннего запроса во внешнюю команду и имеют две основные формы: некоррелированную (noncorrelated) и коррелированную (correlated).

В чем различие этих форм? Коррелированные подзапросы содержат ссылки на значения полей в запросе верхнего уровня, а некоррелированные – не содержат. Поэтому некоррелированный подзапрос выполняется автономно от внешней команды. После чего внешний запрос выполняет то или иное действие, основываясь на полученных результатах выполнения внутреннего запроса. Таким образом, некоррелированный подзапрос вычисляется один раз для запроса верхнего уровня.

Коррелированный подзапрос не может выполняться как независимый запрос, поскольку он содержит условия, зависящие от значений полей в основном запросе. Коррелированная форма подзапроса представляет собой следующее действие — внешняя команда SQL предоставляет значения для внутреннего подзапроса, которые будут использоваться при его выполнении. После этого результаты выполнения подзапроса возвращаются во внешний запрос. В процессе выполнения коррелированный подзапрос вычисляется для каждой строки запроса верхнего уровня.

#### 3.2.3.1. Скалярные подзапросы

Скалярные подзапросы возвращают единственное значение. Они начинаются с простого (немодифицированного) оператора сравнения =, <>, >, >=, <, или <=.

Общая форма таких запросов имеет вид:

Начало команды SELECT, INSERT, UPDATE, DELETE или подзапроса

WHERE <выражение> <оператор _сравнения> (подзапрос)

[Окончание команды SELECT, INSERT, UPDATE, DELETE или подзапроса]

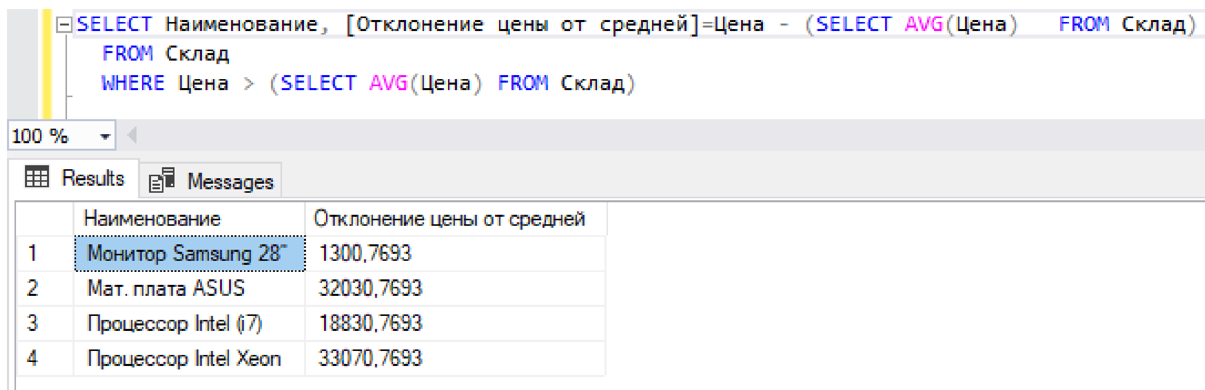
Такими подзапросами достаточно сложно пользоваться, поскольку для этого пользователь должны очень хорошо знать табличные данные и суть решаемой задачи, чтобы быть уверенными, что подзапрос выдаст единственное значение.

По этой причине в подзапросы с оператором сравнения часто включают агрегатные функции, поскольку они гарантированно возвращают единственное значение.

Например, требуется найти названия товаров, цена на которые превышает среднюю цену всех товаров, и отклонение цены от средней. Для решения такой задачи можно использовать следующий запрос:

```
SELECT Наименование, [Отклонение от средней цены] = Цена-(SELECT AVG(Цена)   FROM Склад)
FROM Склад
WHERE Цена > (SELECT AVG(Цена) FROM Склад)
```

Результат выполнения запроса приведен на рис. 3.32




| | Наименование | Отклонение цены от средней |
|---|---|---|
| 1 | Монитор Samsung 28" | 1300,7693 |
| 2 | Мат. плата ASUS | 32030,7693 |
| 3 | Процессор Intel (i7) | 18830,7693 |
| 4 | Процессор Intel Xeon | 33070,7693 |

Рис. 3.32. Запрос с некоррелированным скалярным подзапросом

Во вложенном подзапросе определяется средняя цена товаров на складе. Во внешнем запросе она используется как для вычисления отклонения цены от среднего уровня, так и для определения отклонения цены от средней цены всех товаров.

С операторами сравнения можно использовать и коррелированные подзапросы.

Для того чтобы найти все типы товаров, максимальная цена которых, по меньшей мере в полтора раза превышает величину средней цены для этого типа товаров, можно воспользоваться следующим запросом:

```
SELECT pl.Тип
FROM Приход  p1
GROUP BY  pl.Тип
HAVING  MAX(pl. Цена)   >
     (SELECT AVG1.5 * p2.Цена)
      FROM   Приход  p2
     WHERE pl.Тип = p2.Тип)
```

Результат выполнения запроса приведен на рис. 3.33

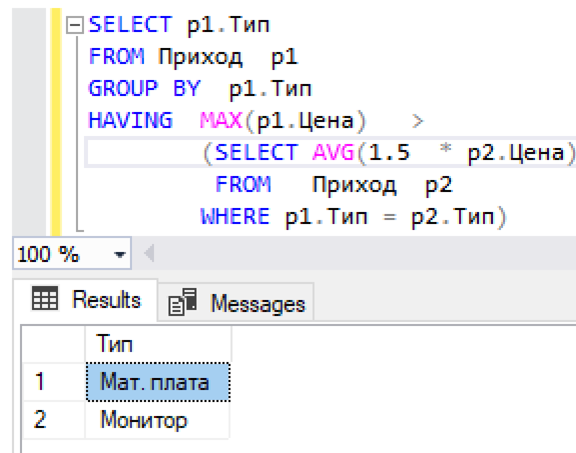



Рис. 3.33. Запрос с коррелированным скалярным подзапросом

Обратите внимание, что коррелированный подзапрос входит в предложение HAVING внешнего запроса. Следует добавить, что подзапрос в этом случае выполняется по одному разу для каждой группы, определенной во внешнем запросе, т. е. по одному разу для каждого типа товара.

### 3.2.3.2. Векторные подзапросы

Векторные подзапросы либо не возвращают ни одного значения, либо возвращают несколько значений. Такие подзапросы начинаются с операторов IN, NOT IN или оператора сравнения с ключевыми словами ANY или ALL.

#### 3.2.3.2.1. Подзапросы, начинающиеся с оператора IN

Такие подзапросы имеют следующую общую форму:

Начало команды SELECT, INSERT, UPDATE, DELETE или подзапроса

WHERE <выражение> [NOT] IN (подзапрос)

[Окончание команды SELECT, INSERT, UPDATE, DELETE или подзапроса]

Результатом выполнения такого внутреннего подзапроса является список, не включающий ни одного или включающий несколько значений. После того как он возвратит результаты, к их обработке приступит внешний запрос.

Оператор IN используется для отбора во внешнем запросе только тех записей, которые содержат значения, совпадающие с одним из отобранных внутренним запросом.

Рассмотрим примеры, с помощью которых постараемся разобраться, как выполняются коррелированные и некоррелированные запросы.

Пример: необходимо узнать поставщиков, которые поставляли товар 25.11.2020

Вариант 1. Запрос с некоррелированным подзапросом:

```
SELECT Название
FROM Поставщик
WHERE Код_постав IN (SELECT Код_постав
                     FROM Документ
                     WHERE Дата = ’25.11.2020’
```

Результат выполнения запроса приведен на рис. 3.34

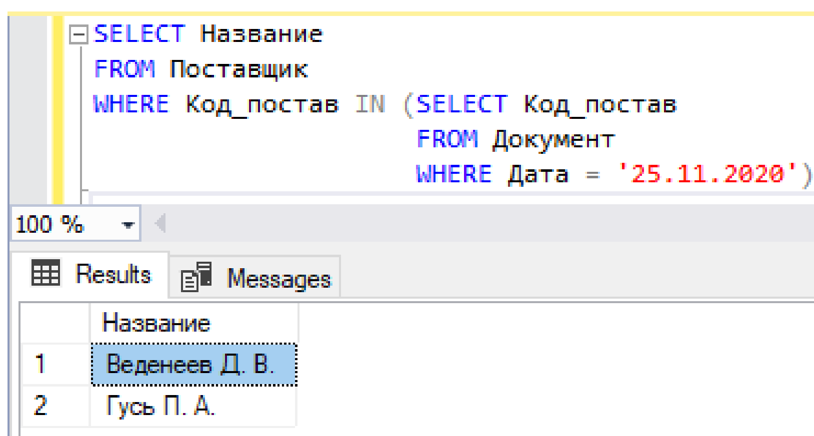



Рис. 3.34. Запрос с некоррелированным подзапросом и оператором IN

С концептуальной точки зрения, весь запрос реализуются за два шага. Сначала внутренний запрос

```
SELECT Код_постав
FROM Документ
WHERE Дата = '25.11.2020'
```

возвращает коды тех поставщиков, которые поставляли товар 25.11.2020 (например: 120 и 220).

Затем эти значения передаются во внешний запрос, который ищет в таблице Поставщик названия поставщиков, соответствующие найденным кодам:

```
SELECT Название
FROM Поставщик
WHERE Код_постав IN (120, 220)
```

Например, результатом этого запроса будет:

Веденеев Д.В.

Гусь П.А.

Вариант 2. Запрос с коррелированным подзапросом, позволяющий узнать поставщиков, которые поставляли товар 15.03.2020

```
SELECT Название
FROM Поставщик A
WHERE '25.11.2020' IN (SELECT Дата
                      FROM Документ
                      WHERE A.Код_постав = Код_постав)
```

В коррелированном запросе внутренний подзапрос не может быть реализован отдельно: он для своего выполнения должен получить данные из внешнего запроса и выполняется последовательно для каждой строки во внешнем запросе. Так для рассматриваемого примера внутренний запрос будет выполняться столько раз, сколько строк содержит внешняя таблица Поставщик.

Как же выполняется обработка в этом случае? Внешний запрос отыскивает первое имя в таблице Поставщик. В нашей учебной базе это Петров А.Н., имеющий код равный 100. Теперь внутренний запрос выполняет операцию соединения, чтобы найти требуемые строки в таблице Документ. Предположим, что таких строк обнаружено две, в которых значение поля Дата равно соответственно 13.01.2020 и 20.07.2020.

После этого, внутренний запрос возвращает результат во внешний запрос, где полученные значения Документ.Дата сравниваются с константой '25.11.2020'. Следовательно, Петров А.Н. в результат запроса не попадет.

Затем снова начинает работать подзапрос, но на этот раз с кодом поставщика второй строки из таблицы Поставщик. Он находит одну строку с датой 20.10.2020 отличной от 25.11.2020. Соответственно внешний запрос не найдет поставщика, удовлетворяющему критерию, и подзапрос начинает выполняться в третий раз.

Для третьей строки таблицы Поставщик со значением Код_постав, равным 120 (поставщик Веденеев Д.В.), подзапрос находит две строки. Одна из них относится к дате '25.11.2020', но этого

достаточно — этот поставщик попадает в результирующий список. После этого запрос продолжится выполняться для каждой строки внешнего запроса.

Подобным образом будут обработаны все записи таблицы Поставщик.

Несмотря на всю разницу выполнения некоррелированного и коррелированного запросов, результаты выполнения обоих запросов будут выглядеть совершенно одинаково (рис. 3.34).

Таким образом, порядок выполнения команды SELECT с коррелированным подзапросом выглядит следующим образом:

1. Выбирается строка из таблицы, имя которой указано во внешнем запросе.

2. Выполняется подзапрос и полученное значение применяется для анализа этой строки в условии предложения WHERE внешнего запроса.

3. По результату оценки этого условия принимается решение о включении или не включении строки в состав выходных данных.

4. Процедура повторяется для следующей строки таблицы внешнего запроса.

Вариант 3. Этот же запрос можно сформулировать как запрос на соединение таблиц.

```
SELECT DISTINCT Название
FROM Поставщик p, Документ  d
WHERE p.Код_постав = d. Код_постав  AND  Дата = '25.11.2020'
```

Ключевое слово DISTINCT добавлено в список выбора, чтобы названия каждого поставщика отображались только один раз. Результат выполнения этого запроса будет таким же, как и в вариантах 1 и 2.

Оператор NOT IN используется для отбора во внешнем запросе только тех записей, которые содержат значения, не совпадающие ни с одним из отобранных внутренним запросом.

Например, запрос

```
SELECT Название
FROM Поставщик
WHERE Код_постав NOT IN (SELECT Код_постав
                         FROM Документ
                         WHERE Дата = '25.11.2020')
```

отберет только тех поставщиков, дата поставки которых не совпадает с датой 25.11.2020.

#### 3.2.3.2.2. Подзапросы, включающие ключевые слова ANY или ALL

Еще один вид подзапросов, которые либо не возвращают ни одного значения, либо возвращают несколько значений, использует оператор сравнения, модифицированный ключевыми словами ANY или ALL. Все они имеют следующую общую форму:

Начало команды SELECT, INSERT, UPDATE, DELETE или подзапроса

WHERE <выражение> <оператор_сравнения> [ANY / ALL] (подзапрос)

[Окончание команды SELECT, INSERT, UPDATE, DELETE или подзапроса]

Рассмотрим, для чего нужны ключевые слова ANY и ALL.

*Проверка ANY*

В проверке ANY, для того чтобы сравнить проверяемое значение со столбцом данных, отобранным внутренним запросом, используется один из шести операторов сравнения (=, <>, <, <=, >, >=). Проверяемое значение поочередно сравнивается с *каждым* элементом, содержащимся в столбце.

В стандарте ANSI/ISO для языка SQL содержатся подробные правила, определяющие результаты проверки ANY, когда проверяемое значение сравнивается со столбцом результатов внутреннего запроса:

♦ Если операция сравнения имеет значение TRUE *хотя бы для одного* значения в столбце, то проверка ANY возвращает значение TRUE (имеется некоторое значение, полученное внутренним запросом, для которого условие сравнения выполняется).

♦ Если операция сравнения имеет значение FALSE для всех значений в столбце, то проверка ANY возвращает значение FALSE (можно утверждать, что ни для одного значения, возвращенного внутренним запросом, условие сравнения не выполняется).

♦ Если внутренний запрос возвращает результат в виде пу-

стого столбца, то проверка ANY возвращает значение FALSE (в результате выполнения внутреннего запроса не получено ни одного значения, для которого выполнялось бы условие сравнения).

♦ Если операция сравнения не имеет значение TRUE ни для одного значения в столбце, но в нем присутствуют одно или несколько значений NULL, то проверка ANY возвращает результат NULL. В этой ситуации невозможно с определенностью утверждать, существует ли полученное внутренним запросом значение, для которого выполняется условие сравнения.

Например: выбрать принтеры, цена которых больше цены любого монитора

```
SELECT Наименование, Приход.Цена
FROM Приход, Склад
WHERE Приход.Ном_ном = Склад. Ном_ном
           AND Тип = 'Принтер'
           AND Цена > ANY (SELECT Цена
                                   FROM Приход
                                   WHERE Тип = 'Монитор')
```

Внутренний запрос находит все мониторы и возвращает столбец, содержащий цену этих мониторов. Внешний запрос по очереди проверяет все строки таблицы Приход. Предложение WHERE внешнего запроса сравнивает цену принтера с ценой каждого монитора, отобранного внутренним запросом. Если есть ***хотя бы один*** монитор, который стоит меньше, чем заданное проверяемое значение, то проверка > ANY возвращает значение TRUE, а наименование принтера заносится в таблицу результатов запроса. Если таких принтеров нет, название принтера в таблицу результатов запроса не попадает. В учебной базе данных содержаться мониторы с ценами 9820,00 руб., 5820,00 руб., 4790,00 руб., 16790,00 руб. и принтеры с ценами 6520,00 руб., 3990,00 руб., 5200,00 руб., 6990,00 руб. Следовательно, принтер с ценой 3990,00 руб. в итоговую таблицу не попадет.

Результат выполнения запроса приведен на рис. 3.35.

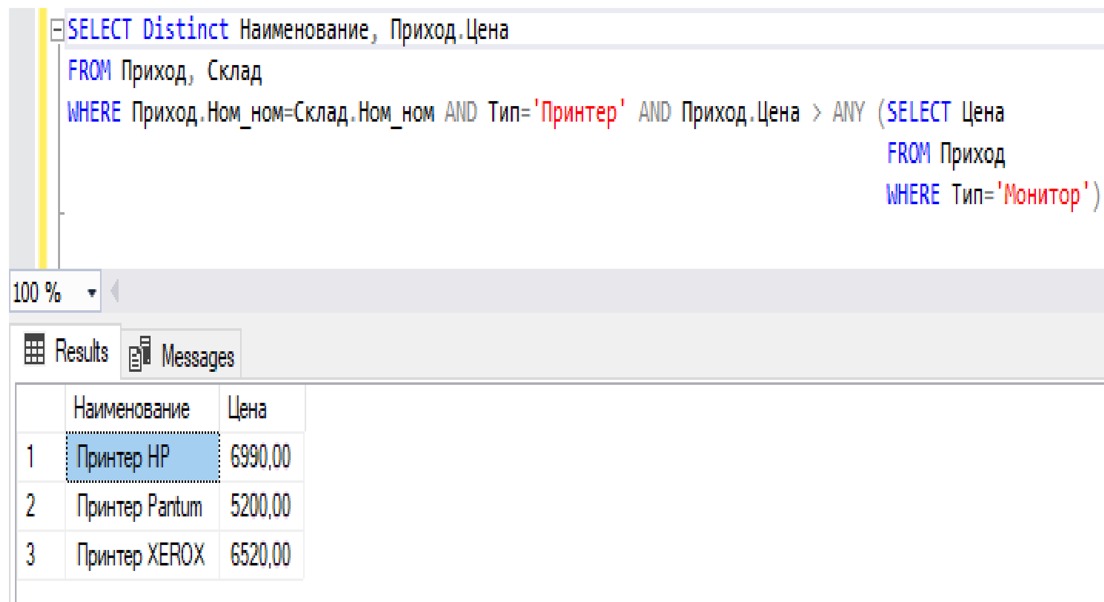



Рис. 3.35. Запрос с некоррелированным подзапросом
и проверкой >ANY

Надо отметить, что оператор ANY не полностью однозначен.

Создадим запрос, чтобы найти все поступления на склад, сумма которых меньше любой суммы поступления до 14.11.2020

```
SELECT *, Цена * Количество AS 'Сумма'
FROM Приход
WHERE Цена * Количество < ANY (SELECT Цена * Количество
                               FROM Приход A, Документ B
                               WHERE A.Ном_док = B.Ном_док
                                     AND Дата < '14.11.2020')
```

Внутренний запрос

```
SELECT Цена * Количество
FROM Приход A, Документ B
WHERE A.Ном_док = B.Ном_док
      AND Дата < '14.11.2020'
```

выводит следующие значения 65200, 32600, 39900 и 41600. Спрашивается: с какой суммой попадут товары в результирующий запрос?

Результат выполнения запроса приведен на рис. 3.36.

Как следует из рис. 3.36, в результирующий запрос попали все товары с суммой меньшей 65200.

```
SELECT *, Цена*Количество AS 'Сумма'
FROM Приход
WHERE Цена*Количество < ANY (SELECT Цена*Количество
                              FROM Приход A,Документ B
                              WHERE A.Ном_док = B.Ном_док
                                    AND B.Дата < '14.11.2020')
```

100 %

Results | Messages

| | Ном_док | Ном_ном | Тип | Цена | Количество | Сумма |
|---|---|---|---|---|---|---|
| 1 | 1200 | 10010 | Принтер | 6520,00 | 5 | 32600,00 |
| 2 | 1300 | 10020 | Принтер | 3990,00 | 10 | 39900,00 |
| 3 | 1500 | 10030 | Принтер | 5200,00 | 8 | 41600,00 |
| 4 | 1710 | 10040 | Принтер | 6990,00 | 7 | 48930,00 |
| 5 | 1850 | 10050 | Монитор | 9820,00 | 6 | 58920,00 |
| 6 | 1960 | 10060 | Монитор | 5820,00 | 9 | 52380,00 |
| 7 | 2500 | 10070 | Монитор | 4790,00 | 8 | 38320,00 |
| 8 | 2790 | 10070 | Монитор | 4790,00 | 4 | 19160,00 |

Рис. 3.36. Запрос с некоррелированным подзапросом
и проверкой <ANY

Для модификации операторов сравнения используются следующие операторы:

< ANY – означает: меньше, чем любое полученное число; эквивалентно < для самого большего полученного числа. Поэтому < ANY (1,2,3) означает меньше 3.

< = ANY – означает: меньше или равно любому полученному числу; эквивалентно операции < = для самого большего полученного числа.

> ANY – означает: больше, чем наименьшее выбранное значение. Поэтому > ANY (1,2,3) означает больше 1.

> = ANY – означает: больше или равно для самого меньшего полученного числа.

= ANY – полностью эквивалентен оператору IN.

<>ANY не эквивалентно NOT IN и существенно отличается от оператора NOT IN:

- <>ANY (a, b, c) — <> *a* ИЛИ <> b ИЛИ <> c
- NOT IN (a,b,c) — <> a И <> b И <> c

*Проверка ALL*

В проверке ALL используется один из шести операторов (=, <>, <, <=, >, >=) для сравнения проверяемого значения со столбцом данных, отобранным внутренним запросом. Проверяемое значение поочередно сравнивается с *каждым* элементом, содержащимся в столбце.

В стандарте ANSI/ISO для языка SQL содержатся подробные правила, определяющие результаты проверки ALL, когда проверяемое значение сравнивается со столбцом результатов внутреннего запроса:

♦ Если операция сравнения дает результат TRUE для *каждого* значения в столбце, то проверка ALL возвращает значение TRUE. Условие сравнения выполняется для каждого значения, возвращенного внутренним запросом.

♦ Если операция сравнения дает результат FALSE для *какого-нибудь* значения в столбце, то проверка ALL возвращает значение FALSE. В этом случае можно утверждать, что условие сравнения выполняется не для каждого значения, возвращенного внутренним запросом.

♦ Если внутренний запрос возвращает результат в виде пустого столбца, то проверка ALL возвращает значение TRUE. Считается, что условие сравнения выполняется, даже если результаты внутренним запроса отсутствуют.

♦ Если операция сравнения не дает результат FALSE ни для одного значения в столбце, но в нем присутствуют одно или несколько значений NULL, то проверка ALL возвращает значение NULL. В этой ситуации нельзя с определенностью утверждать, для всех ли значений, возвращенных внутренним запросом, справедливо условие сравнения.

Вот пример запроса с оператором ALL: вывести те товары, цена которых больше чем цена каждого монитора

```
SELECT *
FROM Приход
WHERE Цена > ALL (SELECT Цена
                  FROM Приход
                  WHERE Тип = 'Монитор')
```

Предложение WHERE внешнего запроса сравнивает значения цены каждого полученного товара со всеми ценами, полученными в результате выполнения внутреннего запроса. Если цены всех мониторов меньше проверяемого значения, то предикат >ALL возвращает значение TRUE и данный товар попадает в запрос.

Внутренний запрос определяет значения цен всех мониторов. Таблица Приход содержит пять мониторов и цены у них 9820 руб., 5820 руб., 4790 руб., 16790 руб. и 15000 руб. соответственно. Внешний запрос, используя полученные во внутреннем запросе данные, находит товары с ценой большей, чем у любого из мониторов. Например, самая высокая цена монитора – 16790 рублей. Следовательно, выбираются только те товары, у которых цена выше 16790 рублей.

Результат выполнения запроса приведен на рис. 3.37.

```
SELECT *
FROM Приход
WHERE Цена > ALL (SELECT Цена
                  FROM Приход
                  WHERE Тип = 'Монитор')
```

100 %

Results | Messages

|  | Ном_док | Ном_ном | Тип | Цена | Количество |
|---|---|---|---|---|---|
| 1 | 7420 | 10090 | Мат. плата | 47520,00 | 8 |
| 2 | 9680 | 10120 | Процессор | 34320,00 | 7 |
| 3 | 7420 | 10130 | Процессор | 48560,00 | 5 |

Рис. 3.37. Запрос с некоррелированным подзапросом и проверкой <ALL

В проверке ALL используются следующие операторы:

> ALL – означает: больше каждого значения элементов результирующего множества, что равносильно больше максимальной величины. Например, > ALL (1,2,3) означает больше, чем 3.

< ALL – меньше каждого значения элементов результирующего множества. Что равносильно меньше минимальной величины. Например, < ALL (1,2,3) означает меньше, чем 1.

= ALL – означает: равно всем полученным значениям.

Ключевое слово ALL используется в основном с неравенствами, чем с равенствами, так как значение может быть "равным для всех" результатом подзапроса, только если все результаты идентичны.

Посмотрите следующий запрос:

```
SELECT *
FROM Приход
WHERE Цена = ALL (SELECT Цена
                  FROM Приход
                  WHERE Тип = 'Монитор')
```

Эта команда допустима, но с данными таблицы Приход, мы не получим никакого вывода (рис. 3.38). Только в единственном случае вывод будет выдан этим запросом – если цены всех мониторов окажутся одинаковыми.

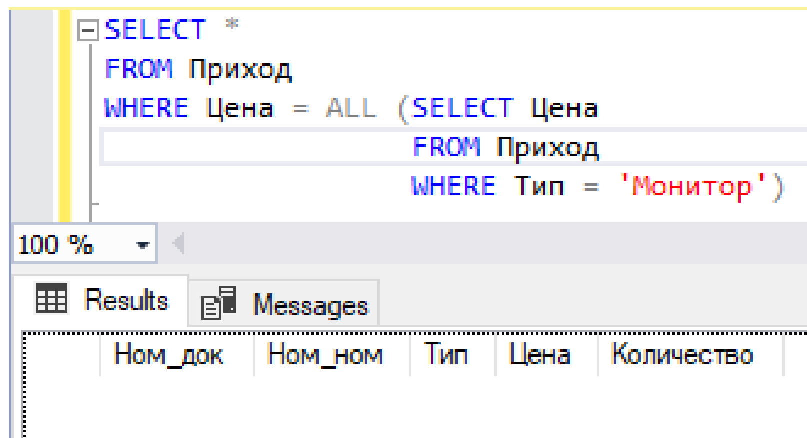



Рис. 3.38. Запрос с некоррелированным подзапросом и проверкой =ALL

Однако ALL может более эффективно использоваться с неравенствами, то есть с оператором <>. В SQL, выражение <> ALL в действительности соответствует " *не равен любому* " результату подзапроса. Другими словами, предикат верен, если данное значение не найдено среди результатов подзапроса. Следовательно, наш предыдущий пример противоположен по смыслу этому примеру:

```
SELECT *
FROM Приход
WHERE Цена <> ALL (SELECT Цена
                   FROM Приход
                   WHERE Тип = 'Монитор')
```

Внутренний подзапрос выбирает цены всех мониторов. Например, он выводит набор из пяти значений: 9820 руб., 5820 руб., 4790 руб., 16790 руб. и 15000 руб. Затем, основной запрос, выбирает все строки, с ценой, не совпадающей ни с одной из них.

Результат выполнения запроса приведен на рис. 3.39.

```
SELECT *
FROM Приход
WHERE Цена <> ALL (SELECT Цена
                   FROM Приход
                   WHERE Тип = 'Монитор')
```

100 %

Results | Messages

| | Ном_док | Ном_ном | Тип | Цена | Количество |
|---|---|---|---|---|---|
| 1 | 1000 | 10010 | Принтер | 6520,00 | 10 |
| 2 | 1200 | 10010 | Принтер | 6520,00 | 5 |
| 3 | 1300 | 10020 | Принтер | 3990,00 | 10 |
| 4 | 1500 | 10030 | Принтер | 5200,00 | 8 |
| 5 | 1710 | 10040 | Принтер | 6990,00 | 7 |
| 6 | 7420 | 10090 | Мат. плата | 47520,00 | 8 |
| 7 | 8750 | 10100 | Мат. плата | 7632,00 | 26 |
| 8 | 8750 | 10110 | Мат. плата | 3408,00 | 35 |
| 9 | 9680 | 10120 | Процессор | 34320,00 | 7 |
| 10 | 7420 | 10130 | Процессор | 48560,00 | 5 |

Рис. 3.39. Запрос с некоррелированным подзапросом и проверкой <>ALL

Можно сформулировать тот же самый запрос, используя, оператор NOT IN:

```
SELECT *
FROM Приход
WHERE Цена NOT IN (SELECT Цена
                   FROM Приход
                   WHERE Тип = 'Монитор')
```

Можно также использовать оператор ANY:

```
SELECT *
FROM Приход
 WHERE NOT Цена = ANY (SELECT Цена
                       FROM Приход
                      WHERE Тип = 'Монитор')
```

Результат будет одинаков для всех трех запросов (рис. 3.39).

Одно значительное различие между ключевыми словами ALL и ANY это способ действия в ситуации, когда внутренний запрос (подзапрос) не возвращает никаких значений. В этом случае проверка ALL - автоматически верна, а проверка ANY автоматически неправильна. Это означает, что в случае отсутствия в таблице Приход мониторов, следующий запрос

```
SELECT *
FROM Приход
WHERE Цена > ANY (SELECT Цена
                  FROM Приход
                  WHERE Тип = 'Монитор')
```

не произведет никакого вывода, в то время как запрос -

```
SELECT *
FROM Приход
WHERE Цена > ALL (SELECT Цена
                  FROM Приход
                  WHERE Тип = 'Монитор')
```

выведет всю таблицу Приход.

#### 3.2.3.2.3. Проверка на существование

В результате проверки на существование (оператор EXISTS) можно выяснить, содержится ли в таблице результатов внутреннего запроса хотя бы одна строка.

EXISTS - это оператор, который производит значение TRUE, если в таблице результатов внутреннего запроса содержится, хотя бы одна строка, и FALSE, если таблица результатов внутреннего запроса не содержит ни одной строки.

Как видим, этот оператор производит значение логического типа. Это означает, что оператор EXISTS может работать автономно или в комбинации с другими логическими выражениями, использующими логические операторы AND, OR и NOT.

Например, запрос выводит некоторые данные из таблицы Поставщик, если один или более поставщиков в этой таблице находятся в Московской области

```
SELECT Код_постав, Название, Регион
FROM Поставщик
WHERE EXISTS ( SELECT *
               FROM Поставщик
               WHERE Регион =  'Моск.обл.' )
```

Результат выполнения запроса приведен на рис. 3.40.

```
SELECT Код_постав, Название, Регион
FROM Поставщик
WHERE EXISTS ( SELECT *
               FROM Поставщик
               WHERE Регион =  'Московск.обл.' )
```

100 %

Results | Messages

| | Код_постав | Название | Регион |
|---|---|---|---|
| 1 | 100 | Петров А. Н. | Владимир.обл. |
| 2 | 110 | Васильев К. Р. | Владимир.обл. |
| 3 | 120 | Веденеев Д. В. | Владимир.обл. |
| 4 | 130 | Логинова Н. Д. | Костром.обл. |
| 5 | 140 | Сталь Д. П. | Воронеж.обл. |
| 6 | 150 | Сабаев Т. Р. | Иванов.обл. |
| 7 | 160 | Ромашкина М. П. | Воронеж.обл. |
| 8 | 170 | Григорьев Л. М. | Ивановская обл. |
| 9 | 180 | Ветров М. Д. | Костром.обл. |
| 10 | 190 | Филатов П. С. | Московск.обл. |
| 11 | 200 | Акулова М. П. | Московск.обл. |
| 12 | 210 | Малышев Г. И. | Московск.обл. |
| 13 | 220 | Гусь П. А. | Тверская обл. |
| 14 | 230 | Комарова Н. В. | Нижегород.обл. |
| 15 | 240 | Шереметьев А. Р. | Рязанская обл. |

Рис. 3.40. Запрос с некоррелированным подзапросом
и оператором EXISTS

Внутренний запрос выбирает все данные для всех поставщиков в Московской области. Оператор EXISTS в предложении WHERE внешнего запроса отмечает, что некоторый вывод был произведен подзапросом, и принимает значение TRUE. Подзапрос (некоррелированный) был выполнен только один раз для всего внешнего запроса, и, следовательно, имеет одно значение для всех строк таблицы внешнего запроса. Поэтому все строки таблицы Поставщик будут выведе-

ны. В том случае, если в таблице Поставщик отсутствуют поставщики из Московской области, то запрос не вернет ни одной строки.

Обратите внимание на то, что оператор EXISTS возвращает значения TRUE или FALSE и не возвращает никаких данных из таблицы. По этой причине список выбора такого подзапроса часто состоит из одной звездочки *. Здесь нет необходимости указывать названия столбцов, поскольку осуществляется просто проверка существования строк, удовлетворяющих условиям, указанным в подзапросе. Однако можно и явно указать список выбора, следуя обычным правилам. Следует отметить еще одну особенность подзапроса проверки на существование - перед оператором EXISTS не должно быть названий столбцов, констант или других выражений.

Оператор EXISTS можно использовать с коррелированными подзапросами.

В коррелированном подзапросе, оператор EXISTS оценивается отдельно для каждой строки таблицы, имя которой указано во внешнем запросе. Это дает возможность использовать EXISTS как предикат, который генерирует различные ответы для каждой строки таблицы указанной во внешнем запросе. Например, определить, какие товары поставлялись более одного раза

```
SELECT DISTINCT Ном_ном
FROM Приход A
WHERE EXISTS  (SELECT *
               FROM Приход B
               WHERE A.Ном_док <> B.Ном_док
               AND A.Ном_ном = B.Ном_ном
```

Результат выполнения запроса приведен на рис. 3.41.

Для каждой строки внешнего запроса, внутренний запрос находит строки, которые совпадают со значением поля номенклатурный номер товара Ном_ном, но со значением поля номер документа Ном_док, соответствующему другому документу. Если любые такие строки найдены внутренним запросом, это означает, что имеются два разных документа, по которым поступал один и тот же товар. Поэтому оператор EXISTS верен для текущей строки, и значение поля Ном_ном таблицы, указанной во внешнем запросе, будет выведено.

Если бы опция DISTINCT не была указана, то один и тот же товар будет выведен для каждого документа.

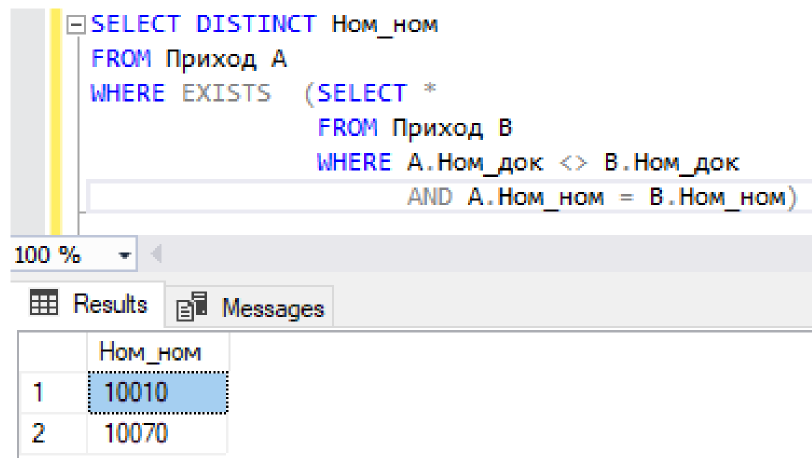



Рис. 3.41. Запрос с коррелированным подзапросом
и оператором EXISTS

В запросах может использоваться оператор NOT EXISTS.

Для оператора NOT EXISTS используются правила обработки, обратные по отношению к ключевому слову EXISTS.

Например, один из способов, которым мы могли бы найти все товары, поставлявшиеся только один раз, будет состоять в том, чтобы инвертировать наш предыдущий пример.

```
SELECT DISTINCT Ном_ном
FROM Приход A
WHERE NOT EXISTS  (SELECT *
                   FROM Приход B
                   WHERE A.Ном_док <> B.Ном_док
                   AND A.Ном_ном = B.Ном_ном
```

Если требуется в запросе вывести больше информации о товарах, а не только их номера, то можно сделать это, соединив, например, таблицу Приход с таблицей Документ

```
SELECT DISTINCT Ном_ном, Дата
FROM Приход A, Документ D
WHERE EXISTS  (SELECT *
               FROM Приход B
               WHERE A.Ном_док <> B.Ном_док
               AND A.Ном_ном = B.Ном_ном)
         AND A. Ном_док = D.Ном_док
```

Результат выполнения запроса приведен на рис. 3.42.

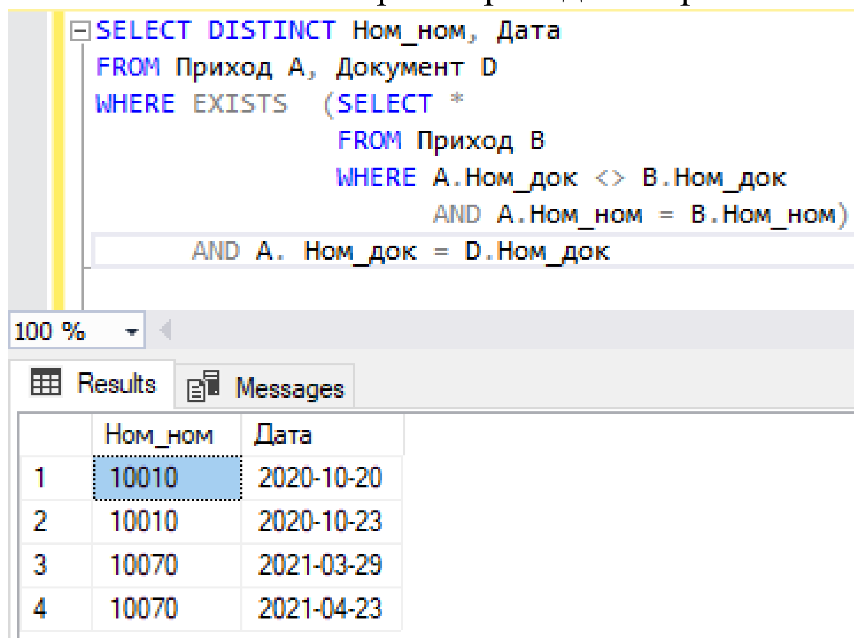



Рис. 3.42. Запрос с комбинацией соединения и подзапроса с оператором EXISTS

Внутренний запрос здесь выглядит, как и в предыдущем примере. Внешний запрос - это соединение таблицы Приход с таблицей Документ. Новое предложение основного предиката (AND A. Ном_док = D.Ном_док), естественно, оценивается на том же самом уровне, что и предложение EXISTS.

Из-за логического оператора AND, оба условия в предложении WHERE внешнего запроса должны быть верны для того, чтобы весь предикат был верен. Следовательно, результаты подзапроса имеют смысл только в тех случаях, когда вторая часть запроса верна, а соединение - выполняемо. Таким образом, комбинация соединения и подзапроса может стать очень мощным способом обработки данных.

### 3.2.3.3. Подзапросы в предложении FROM

Подзапросы – это мощное и гибкое средство создания и комбинирования таблиц результатов запроса. Чтобы сделать подзапросы еще более универсальными, стандарт SQL-92 разрешает использовать их практически везде, где в запросах допускаются ссылки на таблицы. В частности, запрос может указываться вместо имени таблицы в предложении FROM.

Пример: вывести названия и общую сумму поставок для всех поставщиков.

SELECT Название AS ‘Название поставщика’, Общая_поставка

FROM Поставщик A, (SELECT Код_постав, SUM(Цена * Количество) AS Общая_поставка

FROM Приход, Документ

WHERE Приход. Ном_док = Документ. Ном_док

GROUP BY Код_постав) B

WHERE A. Код_постав = B. Код_постав

Результат выполнения запроса приведен на рис. 3.43.

```
SELECT Название AS 'Название поставщика', Общая_поставка
FROM Поставщик A, (SELECT Код_постав, SUM(Цена * Количество) AS Общая_поставка
                   FROM Приход, Документ
                   WHERE Приход. Ном_док = Документ. Ном_док
                   GROUP BY Код_постав)  B
WHERE  A. Код_постав = B. Код_постав
```

100 %

Results | Messages

| | Название поставщика | Общая_поставка |
|---|---|---|
| 1 | Васильев К. Р. | 65200,00 |
| 2 | Веденеев Д. В. | 84980,00 |
| 3 | Сталь Д. П, | 59060,00 |
| 4 | Ромашкина М. П. | 281840,00 |
| 5 | Филатов П. С. | 87250,00 |
| 6 | Гусь П. А. | 58920,00 |
| 7 | Комарова Н. В. | 401662,00 |
| 8 | Шереметьев А. Р. | 622960,00 |

Рис. 3.43. Результат запроса с подзапросом в предложении FROM

Если первая спецификация в предложении FROM – это, как обычно, имя таблицы, то вторая спецификация – вовсе не имя таблицы, а полноценный запрос. Фактически последний может быть намного сложнее.

Когда запрос включается в предложение FROM, СУБД сначала выполняет этот запрос и создает временную таблицу его результатов, а затем выполняет внешний запрос, используя в нем созданную таблицу в качестве источника данных точно так же, как она использовала бы обычную таблицу, хранящуюся в базе данных. В нашем примере содержимым временной таблицы является информация, извлеченная из таблиц Приход и Документ, и к этой информации присоединяется таблица Поставщик для получения названий поставщиков.

Этот же запрос можно записать и другим способом. Например, соединить три таблицы Поставщик, Приход и Документ с группировкой результатов. В чем и состоит прелесть SQL. Он дает пользователям несколько способов получения одного и того же результата.

### 3.2.3.4. Правила формирования подзапросов

Теперь, когда у вас есть представление о подзапросах, суммируем некоторые правила, по которым они должны строиться. В основном они относятся к списку выбора подзапроса с некоторыми дополнительными ограничениями на функции, которые в них можно использовать:

- Список выбора внутреннего запроса, начинающийся с оператора сравнения или с IN, может включать только одно выражение или имя столбца. При этом столбец, который указывается в предложении WHERE внешнего запроса, должен быть совместимым со столбцом, имя которого задается в списке выбора подзапроса.
- Список выбора подзапроса, начинающийся с ключевого слова EXISTS, почти всегда включает звездочку (*). Связано это с тем, что в данном случае нет необходимости приводить имена столбцов, поскольку выполняется лишь проверка на существование (или отсутствие) любых строк, которые удовлетворяют заданным критериям.
- Внутренние запросы, начинающиеся простым оператором сравнения (после которого нет ключевого слова ANY или ALL), не могут включать предложения GROUP BY и HAVING.
- Типы данных ntext, text и image не могут быть использованы в списке выбора вложенных запросов.
- Подзапросы не могут манипулировать своими результатами внутри себя. Другими словами, подзапрос не может включать предложение ORDER BY

### Вопросы для самопроверки

1. Что такое подзапрос?
2. Почему возникает необходимость создания подзапросов?
3. Какие группы подзапросов Вы знаете?
4. В каких предложениях команды SELECT могут располагаться подзапросы?
5. Что такое скалярный подзапрос?

6. Какие подзапросы возвращают множество значений?

7. В чем различие при выполнении коррелированных и некоррелированных запросов?

8. С какого оператора начинается подзапрос проверки на существование?

9. Каковы правила составления подзапросов?

**Практические задания**

1. 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 запрос с подзапросом, который показывал всех продавцов, имеющих максимальные комиссионные.
- Напишите запрос с подзапросом, который бы выбирал все покупки с суммой больше, чем любая для заказчиков в Москве.
- Напишите запрос с подзапросом, который бы показывал каждого продавца с многочисленными заказчиками.

2. Дана база данных для работников ГИБДД:

ВОДИТЕЛЬ (номер удостоверения, ФИО, адрес)

АВТОМОБИЛЬ (регистрационный номер, марка, цвет, номер водительского удостоверения)

СОВЕРШАЕТ НАРУШЕНИЕ (номер водительского удостоверения, код нарушения, дата, место, время)

НАРУШЕНИЕ (код нарушения, наименование нарушения, мера пресечения)

ОБСТОЯТЕЛЬСТВО (дата, время, место, погода)

Напишите запрос с подзапросом, который выводит водителей, имеющих более одного автомобиля?

3. Дана база данных, состоящая из четырех таблиц:

КЛИЕНТ (Код_клиента, Фамилия, Имя, Отчество, Город);

ПРОЖИВАЕТ(Код_клиента, Номер, Дата_прибытия, Дата_убытия);

НОМЕР (Номер, Число мест, Этаж);

ТИП НОМЕРА (Число мест, Цена)

Напишите запрос с подзапросом, который перечислит фамилии клиентов, которые в текущем году останавливались в одноместных номерах.

### 3.2.4. Оконные функции T-SQL

#### 3.2.4.1. Описание оконных функций

Оконные функции (window functions) это функции, применяемые к набору строк связанных с текущей строкой.

Эти функции основаны на принципе языка SQL - принципе работы с окнами (windowing). Основа этого принципа - возможность выполнять различные вычисления с набором строк, или окном, и возвращать одно значение.

Оконные функции в основном используются для решения аналитических задач, например, для вычисления каких-то статистических значений (нарастающие итоги, скользящие средние, промежуточные итоги и так далее).

Конечно, все что могут оконные функции, можно реализовать и без них, например, с помощью агрегатных функциях AVG, SUM, COUNT и др. Однако оконные функции обладают большим преимуществом перед агрегатными функциями: оконные функции не приводят к группированию строк в одну строку вывода, строки сохраняют значения своих атрибутов, а агрегированное значение добавляется к каждой строке.

В стандарте языка SQL предусмотрена поддержка нескольких типов оконных функций:

- агрегатные,
- ранжирующие,
- сдвига,
- аналитические (или распределения).

В одной команде SELECT с одним предложением FROM можно использовать несколько оконных функций.

Впервые поддержка оконных функций появилась в версии Microsoft SQL Server 2005, в которой была реализованы базовая функциональность. В Microsoft SQL Server 2012 функционал оконных функций был расширен, и теперь он с лёгкостью решает много задач, для решения которых до этого приходилось писать сложный код.

Определение оконной функции указывается в предложении OVER:

Имя оконной функции (<столбец для вычислений>) OVER

(

[PARTITION BY <столбец для секционирования >]

[ORDER BY <столбец для упорядочения [ASC | DESC]>

[ROWS или RANGE <выражение для фильтрации строк в пределах кадра>]]

)

Предложение OVER определяет набор строк по отношению к текущей строке (окно), предоставляемый как входные данные этапа обработки запроса. Оно содержит три ключевых элемента - секционирование, упорядочение и кадрирование. Задача этих трех элементов — фильтровать строки в окне.

Элемент секционирования реализован как предложение PARTITION BY и поддерживается всеми оконными функциями. Предложение PARTITION BY не является обязательным, но дополняет OVER и позволяет ограничить окно только теми строками, у которых те же атрибуты секционирования, что и в текущей строке. Например, если в функции присутствует предложение PARTITION BY Val и значение атрибута val в текущей строке равно 100, окно, связанное с текущей строкой, обеспечит выбор из результирующего набора всех строк, у которых значение атрибута val равно 100. Если значение атрибута val текущей строки равно 150, в окно войдут все строки со значением атрибута val равным 150. Если предложение PARTITION BY не указано, функция будет обрабатывать все строки результирующего набора.

Элемент упорядочения реализован как предложение ORDER BY. Параметры этого предложения ASC и DESC определяют порядок обработки строк в секции по возрастанию или по убыванию значений. По умолчанию записи сортируются по возрастанию. Все оконные функции поддерживают элемент упорядочения.

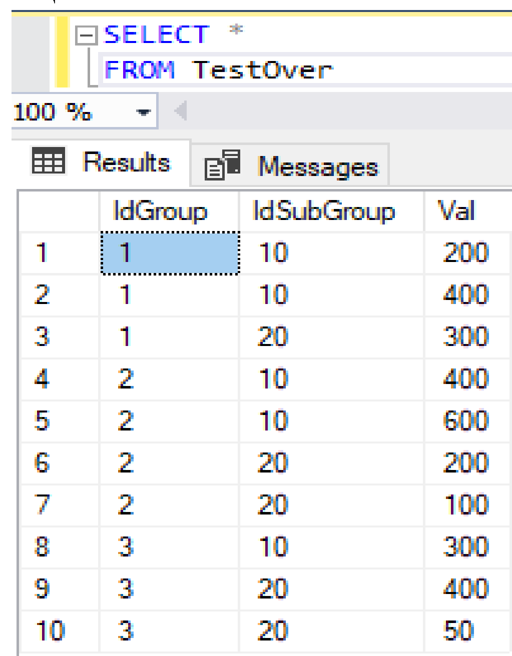
Кадрирование применяется как к агрегатным функциям, так и к трем функциям сдвига: FIRST_VALUE, LAST_VALUE и NTILE. Для кадрирования в стандарте SQL используются предложения ROWS и RANGE. Предложение ROWS ограничивает строки в окне, указывая фиксированное количество строк, предшествующих или следующих

за текущей строкой. Предложение RANGE логически ограничивает строки за счет указания диапазона значений в отношении к значению текущей строки. Оба предложения ROWS и RANGE используются вместе с предложением ORDER BY.

Для того, чтобы лучше понять как работают оконные функции, рассмотрим несколько примеров. Для демонстрации работы оконных функций будем использовать тестовую таблицу с именем TestOver, представленную на рис. 3.44.

Таблица TestOver содержит три столбца. Как видно на рис. 3.44 столбец IdGroup имеет три группы значений, а столбец IdSubGroup - две подгруппы с разным количеством элементов в подгруппе.

Рассмотрение примеров начнем с запросов, использующих агрегатные оконные функции.




| | IdGroup | IdSubGroup | Val |
|---|---|---|---|
| 1 | 1 | 10 | 200 |
| 2 | 1 | 10 | 400 |
| 3 | 1 | 20 | 300 |
| 4 | 2 | 10 | 400 |
| 5 | 2 | 10 | 600 |
| 6 | 2 | 20 | 200 |
| 7 | 2 | 20 | 100 |
| 8 | 3 | 10 | 300 |
| 9 | 3 | 20 | 400 |
| 10 | 3 | 20 | 50 |

Рис. 3.44. Таблица TestOver

### 3.2.4.2. Агрегатные оконные функции

Агрегатные функции – это функции SUM, AVG, MIN, MAX, COUNT, которые выполняют на наборе данных вычисления и возвращают результирующее значение

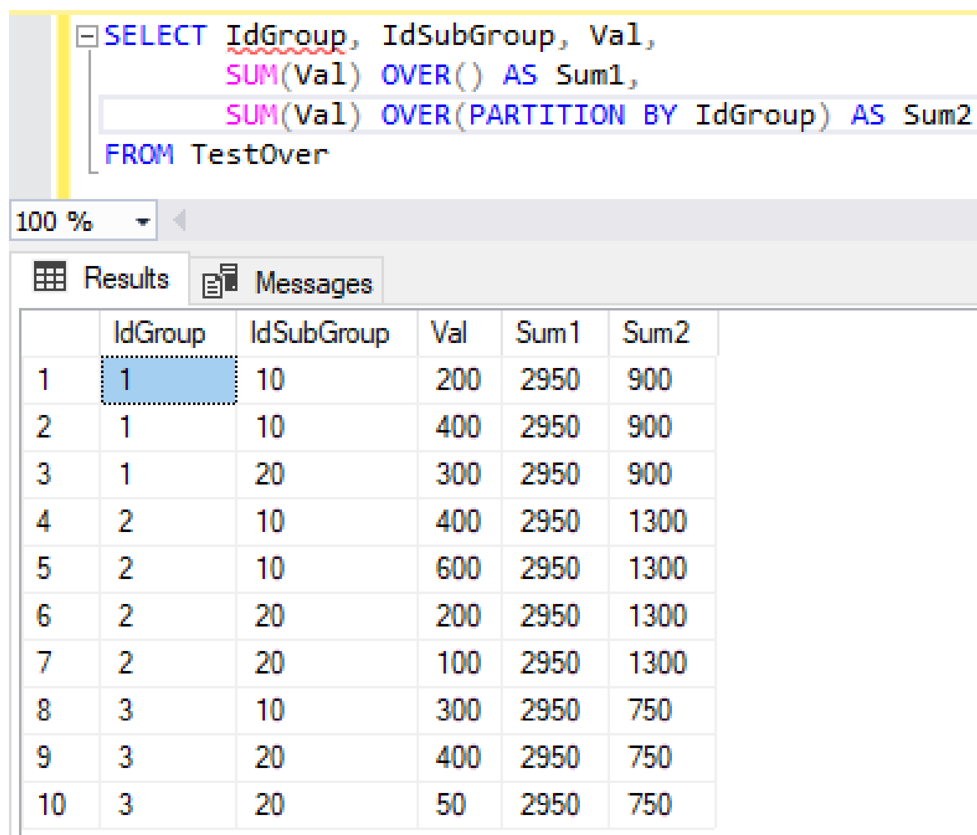
Обычно агрегатные функции используются в запросах в сочетании с предложением GROUP BY. Но их также можно использовать и с предложением OVER. В последнем случае они будут вычислять значения в определённом окне (наборе данных) для каждой текущей строки.

*Использование в агрегатных оконных функциях элемента секционирования (предложение PARTITION BY)*

Давайте посмотрим, как работает предложение OVER с предложением PARTITION BY, на примере функции суммирования:

```
 SELECT IdGroup, IdSubGroup, Val,
    SUM(Val) OVER() AS Sum1,
    SUM(Val) OVER(PARTITION BY IdGroup) AS Sum2
FROM TestOver
```

Результат выполнения команды SELECT с предложением PARTITION BY представлен на рис. 3.45.



```
SELECT IdGroup, IdSubGroup, Val,
       SUM(Val) OVER() AS Sum1,
       SUM(Val) OVER(PARTITION BY IdGroup) AS Sum2
FROM TestOver
```

100 %

Results | Messages

| | IdGroup | IdSubGroup | Val | Sum1 | Sum2 |
|---|---|---|---|---|---|
| 1 | 1 | 10 | 200 | 2950 | 900 |
| 2 | 1 | 10 | 400 | 2950 | 900 |
| 3 | 1 | 20 | 300 | 2950 | 900 |
| 4 | 2 | 10 | 400 | 2950 | 1300 |
| 5 | 2 | 10 | 600 | 2950 | 1300 |
| 6 | 2 | 20 | 200 | 2950 | 1300 |
| 7 | 2 | 20 | 100 | 2950 | 1300 |
| 8 | 3 | 10 | 300 | 2950 | 750 |
| 9 | 3 | 20 | 400 | 2950 | 750 |
| 10 | 3 | 20 | 50 | 2950 | 750 |

Рис. 3.45. Результат выполнения команды SELECT с агрегатной оконной функцией, содержащей предложение PARTITION BY

Обратите внимание на новые столбцы с именами Sum1 и Sum2, которые появился в результирующей таблице. Давайте разберемся со значениями, которые содержат эти столбцы.

Первое предложение OVER() не содержит элементов секционирования, упорядочения и кадрирования. В таком варианте окном будет весь набор данных и никакого упорядочивания не применяется.

Для каждой строки выводится одно и то же значение 2950. Это общая сумма всех значений столбца с именем Val.

Второе предложение OVER содержит предложение PARTITION BY**,** определяющее столбец, по которому будет производиться секционирование данных, и он является ключевым в разбиении набора строк на окна. Предложение PARTITION BY секционировало строки по столбцу IdGroup. Теперь для каждой секции будет применяться оконная функция и рассчитываться своя сумма значений. Можно создавать секции по нескольким столбцам. Тогда в предложении PARTITION BY нужно писать поля для секции через запятую, например, PARTITION BY IdGroup, Val.

*Использование в агрегатных оконных функциях элемента упорядочения (предложение ORDER BY)*

Вместе с предложением PARTITION BY в оконной функции может применяться предложение ORDER BY, которое определяет порядок упорядочения внутри секции. В этом случае оконная функция будет обрабатывать данные согласно этому порядку. Рассмотрим следующую команду

```
SELECT IdGroup, IdSubGroup, Val,
SUM(Val) OVER(PARTITION BY IdGroup ORDER BY Val) AS
Sum3
FROM TestOver
```

В данном примере в оконной функции применяется агрегатная функция SUM по столбцу Val, окно секционируется по столбцу IdGroup, а строки секции упорядочиваются (по возрастанию) по столбцу Val.

Результат выполнения этой команды представлен на рис. 3.46.

По умолчанию, если не задано предложение ORDER BY внутри предложения OVER, границами окна являются все строки. Если предложение ORDER BY задано, то границей для текущей строки будут все предшествующие строки и текущая строка. Таким образом, мы указали, что хотим видеть для каждого значения атрибута Val сумму со всеми предыдущими значениями. Такое суммирование часто называют нарастающий итог или накопительный итог.

```
SELECT IdGroup, IdSubGroup, Val,
       SUM(Val) OVER(PARTITION BY IdGroup ORDER BY Val) AS Sum3
FROM TestOver
```

100 %

Results | Messages

| | IdGroup | IdSubGroup | Val | Sum3 |
|---|---|---|---|---|
| 1 | 1 | 10 | 200 | 200 |
| 2 | 1 | 20 | 300 | 500 |
| 3 | 1 | 10 | 400 | 900 |
| 4 | 2 | 20 | 100 | 100 |
| 5 | 2 | 20 | 200 | 300 |
| 6 | 2 | 10 | 400 | 700 |
| 7 | 2 | 10 | 600 | 1300 |
| 8 | 3 | 20 | 50 | 50 |
| 9 | 3 | 10 | 300 | 350 |
| 10 | 3 | 20 | 400 | 750 |

Рис. 3.46. Результат выполнения команды SELECT с агрегатной оконной функцией, содержащей предложения PARTITION BY и ORDER BY

Если при описании окна секционирование отсутствует, т.е. предложение PARTITION BY не указано, а имеется только предложение ORDER BY, то окном будет весь набор данных.

Рассмотрим, как изменится нарастающий итог, в зависимости от упорядочения. Изменим предыдущую команду, добавив в предложение ORDER BY столбец IdSubGroup:

SELECT IdGroup, IdSubGroup, Val,
SUM(Val) OVER(PARTITION BY IdGroup ORDER BY IdSubGroup, Val) AS Sum4
FROM TestOver

Результат выполнения этой команды приведен на рис. 3.47.

Как следует из рис. 3.47, в столбце Sum4 последние в секции значения сходятся с предыдущим примером, но сумма для всех остальных строк отличается. Поэтому важно четко понимать, что вы хотите получить в итоге.

```sql
SELECT IdGroup, IdSubGroup, Val,
       SUM(Val) OVER(PARTITION BY IdGroup ORDER BY IdSubGroup, Val) AS Sum4
FROM TestOver
```

100 %

Results | Messages

| | IdGroup | IdSubGroup | Val | Sum4 |
|---|---|---|---|---|
| 1 | 1 | 10 | 200 | 200 |
| 2 | 1 | 10 | 400 | 600 |
| 3 | 1 | 20 | 300 | 900 |
| 4 | 2 | 10 | 400 | 400 |
| 5 | 2 | 10 | 600 | 1000 |
| 6 | 2 | 20 | 100 | 1100 |
| 7 | 2 | 20 | 200 | 1300 |
| 8 | 3 | 10 | 300 | 300 |
| 9 | 3 | 20 | 50 | 350 |
| 10 | 3 | 20 | 400 | 750 |

Рис. 3.47. Результат выполнения команды SELECT с агрегатной оконной функцией, содержащей предложение ORDER BY IdSubGroup, Val

*Использование в агрегатных оконных функциях элемента кадрирования (предложения* ROWS и RANGE)

Теперь рассмотрим, как использовать в оконных функциях элемент кадрирования. При наличии упорядочения кадрирование определяет нижнюю и верхнюю границы в секции окна, так что фильтр проходят только строки между этими двумя границами. Для кадрирования используются два предложения ROWS и RANGE.

*Предложение ROWS*

Предложения ROWS означает, что границы кадра могут задаваться в виде смещения числа строк или разницы от текущей строки. Предложение ROWS содержит следующее выражение для ограничения строк в пределах кадра:

ROWS BETWEEN UNBOUNDED PRECEDING |
<n> PRECENDING |
<n> FOLLOWING |
CURRENT ROW
AND
UNBOUNDED FOLLOWING |
<n> PRECENDING |
<n> FOLLOWING |
CURRENT ROW

где CURRENT ROW – параметр указывает, что окно начинается или заканчивается на текущей строке, он может быть задан как начальная или как конечная точка;

UNBOUNDED FOLLOWING –параметр указывает, что окно заканчивается на последней строке секции, т.е. в окно входят все записи после текущей строки. Этот параметр может быть указан только как верхняя граница окна;

UNBOUNDED PRECEDING – параметр указывает, что окно начинается с первой строки секции, т.е. в окно входят все записи до текущей строки. Данный параметр используется только как нижняя граница окна;

<n > PRECEDING – параметр определяет число строк перед текущей строкой;

<n> FOLLOWING – параметр определяет число строк после текущей строки. Если FOLLOWING используется как начальная точка окна, то конечная точка должна быть также указана с помощью FOLLOWING.

Предшествующие и последующие строки определяются на основании порядка, заданного в предложении ORDER BY.

Для формирования оконного кадра, создаваемого для каждой строки, можно использовать комбинацию этих параметров для достижения желаемого результата, например:

ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING

В этом случае в кадр попадут текущая и одна следующая запись.

Предложение ROWS поддерживает лаконичную форму. Если не указать верхнюю границу, предполагается, что это CURRENT ROW (то есть текущая строка). Поэтому описания

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

и

ROWS UNBOUNDED PRECEDING

эквивалентны друг другу

Рассмотрим следующий запрос:

```
SELECT IdGroup, IdSubGroup, Val,
SUM(Val) OVER(PARTITION BY IdGroup ORDER BY IdSub-
Group, Val ROWS BETWEEN CURRENT ROW AND 1 FOL-
LOWING) AS Sum5
FROM TestOver
```

В результате эта команда сформирует итоговую таблицу, представленную на рис. 3.48.

```
SELECT IdGroup, IdSubGroup, Val,
       SUM(Val) OVER(PARTITION BY IdGroup ORDER BY IdSubGroup,
                     Val ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING) AS Sum5
FROM TestOver
```

100 %

Results | Messages

| | IdGroup | IdSubGroup | Val | Sum5 |
|---|---|---|---|---|
| 1 | 1 | 10 | 200 | 600 |
| 2 | 1 | 10 | 400 | 700 |
| 3 | 1 | 20 | 300 | 300 |
| 4 | 2 | 10 | 400 | 1000 |
| 5 | 2 | 10 | 600 | 700 |
| 6 | 2 | 20 | 100 | 300 |
| 7 | 2 | 20 | 200 | 200 |
| 8 | 3 | 10 | 300 | 350 |
| 9 | 3 | 20 | 50 | 450 |
| 10 | 3 | 20 | 400 | 400 |

Рис. 3.48. Результат выполнения команды SELECT с агрегатной оконной функцией SUM, содержащей предложение ROWS

Здесь, значения столбца Sum5 рассчитывается как сумма текущей и следующей строки в кадре. А последняя в кадре строка имеет то же значение, что и в столбце Val. Первое значение столбца Sum5 равное 600 рассчитано сложением 200 и 400. Для следующего значения ситуация аналогичная. А последняя в кадре сумма имеет значение 300, потому что в текущей строке значение атрибута Val больше не с чем складывать.

В качестве второго примера использования предложения ROWS создадим три оконные функции с тремя определениями кадра:

```
SELECT IdGroup, IdSubGroup, Val,
  MAX(Val) OVER(PARTITION BY IdGroup
                ORDER BY IdSubGroup, Val
                ROWS BETWEEN 1 PRECEDING
                AND 1 PRECEDING) AS Max1,
  MAX(Val) OVER(PARTITION BY IdGroup
                ORDER BY IdSubGroup, Val
                ROWS BETWEEN 1 FOLLOWING
                AND 1 FOLLOWING)) AS Max2,
```

```
SUM(Val) OVER(PARTITION BY IdGroup
                          ORDER BY IdSubGroup, Val
                          ROWS BETWEEN 1 PRECEDING
                          AND 1 FOLLOWING)) AS Sum6
FROM TestOver
```

При вычислении атрибута Max1 определяется кадр, состоящий из строк между 1 предыдущей и 1 предыдущей. Это означает, что кадр содержит только предыдущую строку в секции. Агрегат MAX, применяемый здесь к столбцу Val, излишен, потому что в кадре будет максимум одна строка. Максимальным значением будет значение в этой строке или NULL, если в кадре строк нет (то есть, если текущая строка является первой в секции). На рис. 3.49 приведен результат выполнения команды SELECT.

```
SELECT IdGroup, IdSubGroup, Val,
       MAX(Val) OVER(PARTITION BY IdGroup
                     ORDER BY IdSubGroup, Val
                     ROWS BETWEEN 1 PRECEDING
                          AND 1 PRECEDING) AS Max1,
       MAX(Val) OVER(PARTITION BY IdGroup
                     ORDER BY IdSubGroup, Val
                     ROWS BETWEEN 1 FOLLOWING
                          AND 1 FOLLOWING) AS Max2,
       SUM(Val) OVER(PARTITION BY IdGroup
                     ORDER BY IdSubGroup, Val
                     ROWS BETWEEN 1 PRECEDING
                          AND 1 FOLLOWING) AS Sum6
FROM TestOver
```

100 %

Results | Messages

| | IdGroup | IdSubGroup | Val | Max1 | Max2 | Sum6 |
|---|---|---|---|---|---|---|
| 1 | 1 | 10 | 200 | NULL | 400 | 600 |
| 2 | 1 | 10 | 400 | 200 | 300 | 900 |
| 3 | 1 | 20 | 300 | 400 | NULL | 700 |
| 4 | 2 | 10 | 400 | NULL | 600 | 1000 |
| 5 | 2 | 10 | 600 | 400 | 100 | 1100 |
| 6 | 2 | 20 | 100 | 600 | 200 | 900 |
| 7 | 2 | 20 | 200 | 100 | NULL | 300 |
| 8 | 3 | 10 | 300 | NULL | 50 | 350 |
| 9 | 3 | 20 | 50 | 300 | 400 | 750 |
| 10 | 3 | 20 | 400 | 50 | NULL | 450 |

Рис. 3.49. Результат выполнения команды SELECT с агрегатной оконной функцией MAX, содержащей предложение ROWS

Обратите внимание, что у первой строки в секции нет соответствующей предыдущей, поэтому значение Max1 в первой строке секции равно NULL.

Аналогично при вычислении атрибута Max2 определяется кадр, состоящий из строк между 1 последующей и 1 последующей, то есть имеется в виду только следующая строка. Агрегат MAX(Val) возвращает значение из предыдущей строки (рис. 3.49).

Так как за последней строкой в секции никаких других строк нет, значение Max2 в последней строке секции равно NULL.

При вычислении атрибута Sum6 определяется кадр, состоящий из строк между одной предыдущей и одной следующей, то есть численность строк в кадре может достигать трех.

Как и в предыдущих примерах, нет строки, предшествующей первой строке, и строки, последующей за последней. Функция SUM корректно суммирует количество строк в кадре.

*Предложение RANGE*

Язык SQL позволяет определять оконный кадр с использованием предложения RANGE. В отличие от предложения ROWS, он работает не с физическими строками, а с диапазоном строк в предложении ORDER BY. Это означает, что строки с одинаковыми значениями в столбце предложения ORDER BY будут считаться как одна текущая строка для параметра CURRENT ROW**.** А в предложении ROWS текущая строка – это одна  текущая строка набора данных.

Вот перечень возможностей для верхней и нижней границ в кадре:

```
RANGE BETWEEN UNBOUNDED PRECEDING |
                        CURRENT ROW
    AND
                        UNBOUNDED FOLLOWING |
                       CURRENT ROW
```

Как и ROWS, предложение RANGE также поддерживает лаконичную форму.

В качестве примера рассмотрим запрос, в котором описаны две оконные функции

```
SELECT IdGroup, IdSubGroup, Val,
  SUM(Val) OVER(PARTITION BY IdGroup
                  ORDER BY IdSubGroup
                  RANGE CURRENT ROW) AS Sum7,
  SUM(Val) OVER(PARTITION BY IdGroup
                  ORDER BY IdSubGroup
                  RANGE BETWEEN UNBOUNDED
                 PRECEDING AND CURRENT ROW) AS Sum8,
FROM TestOver
```

Результирующая таблица этого запроса приведена на рис. 3.50.

```
SELECT IdGroup, IdSubGroup, Val,
       SUM(Val) OVER(PARTITION BY IdGroup
                     ORDER BY IdSubGroup
                     RANGE CURRENT ROW) AS Sum7,
       SUM(Val) OVER(PARTITION BY IdGroup
                     ORDER BY IdSubGroup
                     RANGE BETWEEN UNBOUNDED
                     PRECEDING AND CURRENT ROW) AS Sum8
FROM TestOver
```

100 %

Results | Messages

| | IdGroup | IdSubGroup | Val | Sum7 | Sum8 |
|---|---|---|---|---|---|
| 1 | 1 | 10 | 200 | 600 | 600 |
| 2 | 1 | 10 | 400 | 600 | 600 |
| 3 | 1 | 20 | 300 | 300 | 900 |
| 4 | 2 | 10 | 400 | 1000 | 1000 |
| 5 | 2 | 10 | 600 | 1000 | 1000 |
| 6 | 2 | 20 | 200 | 300 | 1300 |
| 7 | 2 | 20 | 100 | 300 | 1300 |
| 8 | 3 | 10 | 300 | 300 | 300 |
| 9 | 3 | 20 | 400 | 450 | 750 |
| 10 | 3 | 20 | 50 | 450 | 750 |

Рис. 3.50. Результат выполнения команды SELECT с агрегатной оконной функцией, содержащей предложение RANGE

В первой оконной функции предложение RANGE настроено на текущую строку. Но для предложения RANGE текущая строка, это все строки, соответствующие одному значению упорядочивания. Упорядочивание в данном примере осуществляется по столбцу IdSubGroup. Первые две строки первого кадра имеют значение атрибута IdSubGroup равное 10. Следовательно, оба эти значения удовлетворяют ограничению RANGE CURRENT ROW. Поэтому значение атрибута Sum7 для каждой из этих строк равно общей сумме по ним

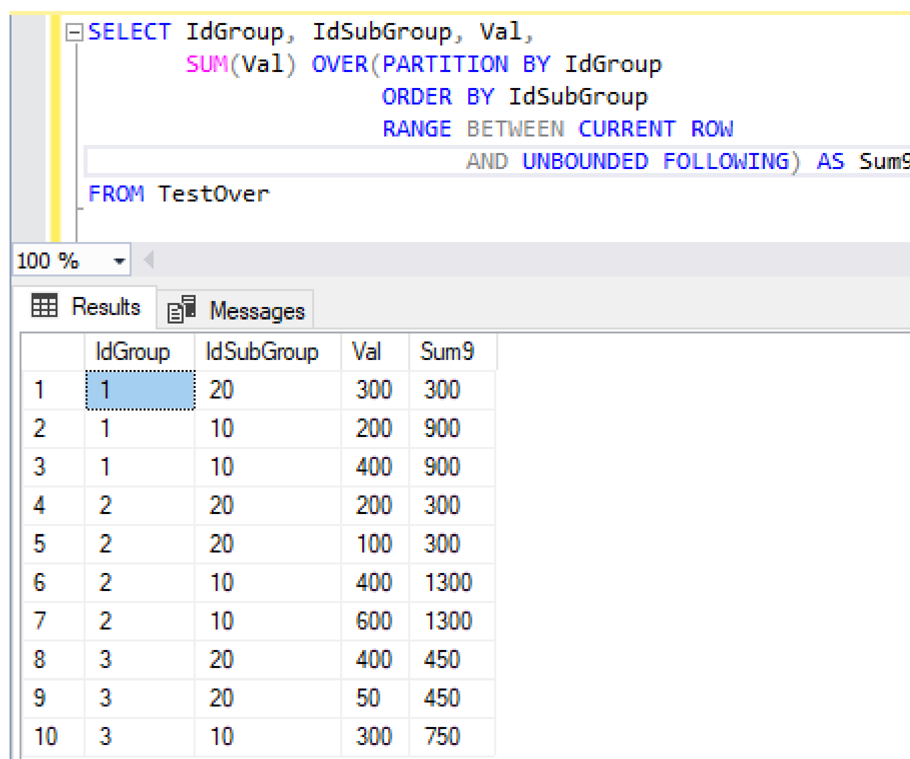
600. Так как во второй кадр входит только одна строка (третья строка) значение атрибута  Sum6 будет равно 300.

Во второй оконной функции кадр задан по всем предыдущим строкам и текущей в секции. Для первой и второй строки это правило работает как в первой оконной функции этого примера, а для третьей как сумма атрибута Val предыдущих строк с текущей.

В следующем примере оконная функция содержит предложение RANGE, которое настроено так, что в кадр входят текущая строка и все записи после текущей строки:

```
SELECT IdGroup, IdSubGroup, Val,
  SUM(Val) OVER(PARTITION BY IdGroup
                ORDER BY IdSubGroup
                RANGE BETWEEN CURRENT ROW
               AND UNBOUNDED FOLLOWING) AS Sum9
FROM TestOver
```

Результат запроса представлен на рис. 3.51.




| | IdGroup | IdSubGroup | Val | Sum9 |
|---|---|---|---|---|
| 1 | 1 | 20 | 300 | 300 |
| 2 | 1 | 10 | 200 | 900 |
| 3 | 1 | 10 | 400 | 900 |
| 4 | 2 | 20 | 200 | 300 |
| 5 | 2 | 20 | 100 | 300 |
| 6 | 2 | 10 | 400 | 1300 |
| 7 | 2 | 10 | 600 | 1300 |
| 8 | 3 | 20 | 400 | 450 |
| 9 | 3 | 20 | 50 | 450 |
| 10 | 3 | 10 | 300 | 750 |

Рис. 3.51. Результат выполнения команды SELECT с агрегатной оконной функцией, содержащей кадр, в который входят текущая строка и все записи после текущей строки

В оконной функции заданное ограничение кадра позволяет получить сумму из текущей строки и всех последующих в рамках одного окна. Так как вторая и третья строка у нас в одной подгруппе, то эти значения и есть текущая строка (Current Row). Поэтому значения атрибута Val этих строк просуммированы сразу.

Обратите внимание на порядок строк в таблице на рис. 3.51. Дело в том, что предложение ORDER BY в описание окна не определяет порядок вывода в результирующей таблице. Если нужно гарантировать упорядочение итоговой таблицы, необходимо добавить в команду SELECT предложение ORDER BY. Например так:

```
SELECT IdGroup, IdSubGroup, Val,
                    SUM(Val) OVER(PARTITION BY IdGroup
                     ORDER BY IdSubGroup
                     RANGE BETWEEN CURRENT ROW
                    AND UNBOUNDED FOLLOWING) AS Sum8
FROM TestOver
ORDER BY IdGroup, IdSubGroup
```

На рис. 3.52 приведена таблица, полученная в результате выполнения последней команды SELECT.

```
SELECT IdGroup, IdSubGroup, Val,
       SUM(Val) OVER(PARTITION BY IdGroup
                     ORDER BY IdSubGroup
                     RANGE BETWEEN CURRENT ROW
                           AND UNBOUNDED FOLLOWING) AS Sum8
FROM TestOver
ORDER BY IdGroup, IdSubGroup
```

100 %

Results | Messages

| | IdGroup | IdSubGroup | Val | Sum8 |
|---|---|---|---|---|
| 1 | 1 | 10 | 200 | 900 |
| 2 | 1 | 10 | 400 | 900 |
| 3 | 1 | 20 | 300 | 300 |
| 4 | 2 | 10 | 400 | 1300 |
| 5 | 2 | 10 | 600 | 1300 |
| 6 | 2 | 20 | 200 | 300 |
| 7 | 2 | 20 | 100 | 300 |
| 8 | 3 | 10 | 300 | 750 |
| 9 | 3 | 20 | 400 | 450 |
| 10 | 3 | 20 | 50 | 450 |

Рис. 3.52. Результат выполнения команды SELECT с агрегатной оконной функцией и сортировкой

### 3.2.4.3. Оконные функции ранжирования

Функции ранжирования – это функции, которые ранжируют значение для каждой строки в группе. Например, их можно использовать для того, чтобы пронумеровать строки по группам или выставить ранг и составить рейтинг.

В Microsoft SQL Server существуют следующие ранжирующие функции:

- ROW_NUMBER –функция вычисляет последовательные номера строк, начиная с 1, в соответствующей секции окна и в соответствии с заданным упорядочением окна;
- RANK — функция возвращает ранг каждой строки в зависимости от значения заданного столбца. В случае нахождения одинаковых значений в столбце, возвращает одинаковый ранг с пропуском следующего;
- DENSE_RANK — функция возвращает «уплотненный» ранг каждой строки. Но в отличие от функции RANK, она для одинаковых значений возвращает ранг, не пропуская следующий;
- NTILE – это функция, которая возвращает результирующий набор, разделённый на группы по определенному столбцу.

Все четыре функции ранжирования поддерживают необязательное предложение секционирования (PARTITION BY) и обязательное предложение упорядочения окна (ORDER BY). Если предложение секционирования окна отсутствует, весь результирующий набор базового запроса считается одной секцией. Что касается предложения упорядочения окна, то оно обеспечивает упорядочение при вычислениях. Понятно, что ранжирование строк без определения упорядочения не имеет смысл.

Рассмотрим примеры использования ранжирующих оконных функций. В первом примере запрос пронумерует все строки таблицы TestOver в соответствии со значениями столбца Val в порядке убывания:

```
SELECT IdGroup, IdSubGroup, Val,
        ROW_NUMBER() OVER (ORDER BY Val DESC) AS
        Rownumber1
FROM TestOver
```

На рис. 3.53 приведен результат запроса.

```
SELECT IdGroup, IdSubGroup, Val,
        ROW_NUMBER() OVER (ORDER BY Val DESC) AS
           Rownumber1
FROM TestOver
```

100 %

Results | Messages

| | IdGroup | IdSubGroup | Val | Rownumber1 |
|---|---|---|---|---|
| 1 | 2 | 10 | 600 | 1 |
| 2 | 3 | 20 | 400 | 2 |
| 3 | 1 | 10 | 400 | 3 |
| 4 | 2 | 10 | 400 | 4 |
| 5 | 1 | 20 | 300 | 5 |
| 6 | 3 | 10 | 300 | 6 |
| 7 | 1 | 10 | 200 | 7 |
| 8 | 2 | 20 | 200 | 8 |
| 9 | 2 | 20 | 100 | 9 |
| 10 | 3 | 20 | 50 | 10 |

Рис. 3.53. Результат выполнения команды SELECT с оконной функцией ранжирования ROW_NUMBER

Так как в запросе SELECT нет предложения ORDER BY, сортировка итоговой таблицы не гарантируется. Если нужно, чтобы сортировка итоговой таблицы запроса выполнялась на основе номера строки, можно использовать псевдоним оконной функции в предложении ORDER BY команды SELECT, например:

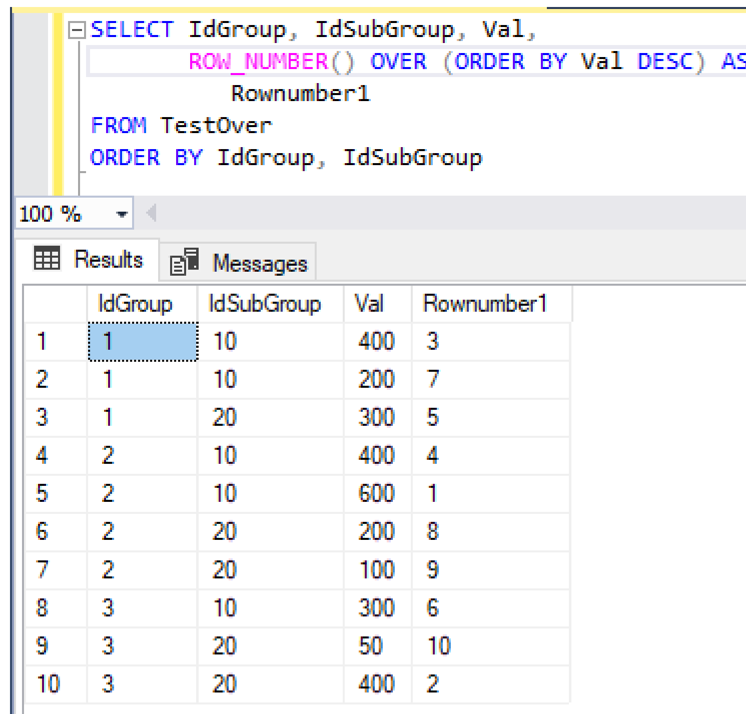
```
SELECT IdGroup, IdSubGroup, Val,
        ROW_NUMBER() OVER (ORDER BY Val DESC) AS
        Rownumber1
FROM TestOver
ORDER BY Rownumber1
```

Результат этого запроса совпадает с предыдущим запросом (рис. 3.53).

Если необходимо упорядочение итоговой таблицы, которое отличается от упорядочения окна, то можно использовать, например, следующий запрос:

```
SELECT IdGroup, IdSubGroup, Val,
        ROW_NUMBER() OVER (ORDER BY Val DESC) AS
        Rownumber1
FROM TestOver
ORDER BY IdGroup, IdSubGroup
```

На рис. 3.54 приведены результаты этого запроса.



```
SELECT IdGroup, IdSubGroup, Val,
        ROW_NUMBER() OVER (ORDER BY Val DESC) AS
          Rownumber1
FROM TestOver
ORDER BY IdGroup, IdSubGroup
```

| | IdGroup | IdSubGroup | Val | Rownumber1 |
|---|---|---|---|---|
| 1 | 1 | 10 | 400 | 3 |
| 2 | 1 | 10 | 200 | 7 |
| 3 | 1 | 20 | 300 | 5 |
| 4 | 2 | 10 | 400 | 4 |
| 5 | 2 | 10 | 600 | 1 |
| 6 | 2 | 20 | 200 | 8 |
| 7 | 2 | 20 | 100 | 9 |
| 8 | 3 | 10 | 300 | 6 |
| 9 | 3 | 20 | 50 | 10 |
| 10 | 3 | 20 | 400 | 2 |

Рис. 3.54. Результат выполнения команды SELECT с оконной функцией ранжирования ROW_NUMBER и сортировкой

Следующий запрос вычисляет ранги и «уплотненные» ранги строк. При этом используется секционирование окна по умолчанию и упорядочение по атрибуту Val по убыванию (DESC):

```
SELECT IdGroup, IdSubGroup, Val,
   RANK()      OVER(ORDER BY Val DESC) AS Rank1,
   DENSE_RANK() OVER(ORDER BY Val DESC) AS
   DenseRank
FROM TestOver
```

Результаты запроса представлены на рис. 3.55.

Все строки с одним номером подгруппы, например 300, получили одинаковый «неплотный» ранг 5 и «плотный» ранг 3. Ранг 5 означает, что есть четыре строки с большими значениями атрибута Val (упорядочение ведется по убыванию), а «плотный» ранг 3 означает, что есть два значения атрибута Val в таблице (600 и 400), больше, чем 300.

```
SELECT IdGroup, IdSubGroup, Val,
       RANK() OVER(ORDER BY Val DESC) AS Rank1,
       DENSE_RANK() OVER(ORDER BY Val DESC) AS
              DenseRank
FROM TestOver
```

100 %

Results | Messages

| | IdGroup | IdSubGroup | Val | Rank1 | DenseRank |
|---|---|---|---|---|---|
| 1 | 2 | 10 | 600 | 1 | 1 |
| 2 | 3 | 20 | 400 | 2 | 2 |
| 3 | 1 | 10 | 400 | 2 | 2 |
| 4 | 2 | 10 | 400 | 2 | 2 |
| 5 | 1 | 20 | 300 | 5 | 3 |
| 6 | 3 | 10 | 300 | 5 | 3 |
| 7 | 1 | 10 | 200 | 7 | 4 |
| 8 | 2 | 20 | 200 | 7 | 4 |
| 9 | 2 | 20 | 100 | 9 | 5 |
| 10 | 3 | 20 | 50 | 10 | 6 |

Рис. 3.55. Результат выполнения команды SELECT с оконными функциями ранжирования RANK() и DENSE_RANK()

Рассмотрим использование элемента секционирования в функциях ранжирования, например:

```
SELECT IdGroup, IdSubGroup, Val,
        ROW_NUMBER() OVER (PARTITION BY
       IdSubGroup ORDER BY IdGroup) AS
       Rownumber2,
        RANK() OVER (PARTITION BY IdSubGroup
        ORDER BY Val) AS Rank2
FROM TestOver
```

Обе оконные функции имеют секционирование по атрибуту IdSubGroup. Первая оконная функция ROW_NUMBER() нумерует строки в каждой подгруппе, при этом используется упорядочивание по столбцу IdGroup. Во второй функции RANK() выставляется ранг каждой записи в подгруппе на основе значения атрибута Val.

На рис. 3.56 представлена результирующая таблица выполнения команды из рассмотренного примера.

```
SELECT IdGroup, IdSubGroup, Val,
        ROW_NUMBER() OVER (PARTITION BY IdSubGroup
                           ORDER BY IdGroup) AS
                Rownumber2,
        RANK() OVER (PARTITION BY IdSubGroup
                     ORDER BY Val) AS Rank2
FROM TestOver
```

100 %

Results | Messages

| | IdGroup | IdSubGroup | Val | Rownumber2 | Rank2 |
|---|---|---|---|---|---|
| 1 | 1 | 10 | 200 | 1 | 1 |
| 2 | 3 | 10 | 300 | 5 | 2 |
| 3 | 1 | 10 | 400 | 2 | 3 |
| 4 | 2 | 10 | 400 | 3 | 3 |
| 5 | 2 | 10 | 600 | 4 | 5 |
| 6 | 3 | 20 | 50 | 5 | 1 |
| 7 | 2 | 20 | 100 | 3 | 2 |
| 8 | 2 | 20 | 200 | 2 | 3 |
| 9 | 1 | 20 | 300 | 1 | 4 |
| 10 | 3 | 20 | 400 | 4 | 5 |

Рис. 3.56. Результат выполнения команды SELECT с оконными функциями ранжирования ROW_NUMBER() и RANK()

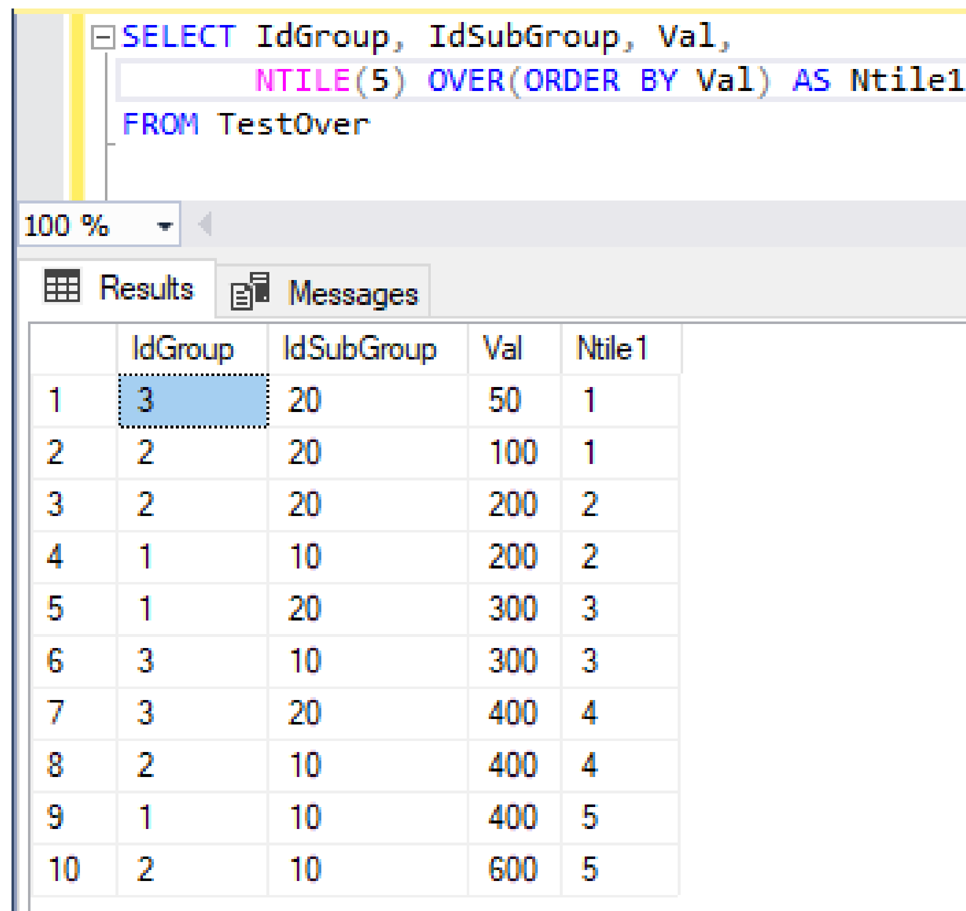
Функция NTILE позволяет разбивать строки в секции окна на примерно равные по размеру подгруппы (tiles) в соответствии с заданным числом подгрупп и упорядочением окна. Допустим, что нужно разбить строки таблицы TestOver на 5 подгрупп одинакового размера на основе упорядочения по атрибуту Val. В таблице 10 строк, значит размер каждой подгруппы будет составлять 2 строки. Поэтому первым 2 строкам будет назначен номер группы 1, следующим 2 строкам — номер подгруппы 2 и т. д. Вот запрос, вычисляющий номера подгрупп:

```
SELECT IdGroup, IdSubGroup, Val,
        NTILE(5) OVER(ORDER BY Val) AS Ntile1
FROM TestOver
```

На рис. 3.57 представлен результат выполнения этого запроса.

Ранее при описании функции NTILE использовалось слово «примерно», потому что число строк, полученное в базовом запросе, может не делиться нацело на число подгрупп. Допустим, вы хотите разбить строки таблицы TestOver на 4 подгруппы. При делении 10 на 4 получаем частное 2 и остаток 2. Это означает, что базовая размер-

ность подгрупп будет две строки, но часть подгрупп получать дополнительную строку.



```
SELECT IdGroup, IdSubGroup, Val,
       NTILE(5) OVER(ORDER BY Val) AS Ntile1
FROM TestOver
```

100 %

Results | Messages

|  | IdGroup | IdSubGroup | Val | Ntile1 |
|---|---|---|---|---|
| 1 | 3 | 20 | 50 | 1 |
| 2 | 2 | 20 | 100 | 1 |
| 3 | 2 | 20 | 200 | 2 |
| 4 | 1 | 10 | 200 | 2 |
| 5 | 1 | 20 | 300 | 3 |
| 6 | 3 | 10 | 300 | 3 |
| 7 | 3 | 20 | 400 | 4 |
| 8 | 2 | 10 | 400 | 4 |
| 9 | 1 | 10 | 400 | 5 |
| 10 | 2 | 10 | 600 | 5 |

Рис. 3.57. Результирующая таблица выполнения команды SELECT с оконной функцией ранжирования NTILE, разбивающая данные на 5 подгрупп

Функция NTILE не пытается распределять дополнительные строки среди подгрупп с равным расстоянием между подгруппами — она просто добавляет по одному ряду в первые подгруппы, пока не распределит остаток. При наличии остатка равного 2 размерность первых 2 подгрупп будет на единицу больше базовой размерности. Поэтому первые 2 подгруппы будут содержать 3 строки, а последние 2 подгруппы — 2 строки, как показано в следующем запросе:

```
SELECT IdGroup, IdSubGroup, Val,
       NTILE(4) OVER(ORDER BY Val) AS Ntile2
FROM TestOver
```

Результаты запроса приведены на рис. 3.58

```
SELECT IdGroup, IdSubGroup, Val,
       NTILE(4) OVER(ORDER BY Val) AS Ntile2
FROM TestOver
```

100 %

Results | Messages

| | IdGroup | IdSubGroup | Val | Ntile2 |
|---|---|---|---|---|
| 1 | 3 | 20 | 50 | 1 |
| 2 | 2 | 20 | 100 | 1 |
| 3 | 2 | 20 | 200 | 1 |
| 4 | 1 | 10 | 200 | 2 |
| 5 | 1 | 20 | 300 | 2 |
| 6 | 3 | 10 | 300 | 2 |
| 7 | 3 | 20 | 400 | 3 |
| 8 | 2 | 10 | 400 | 3 |
| 9 | 1 | 10 | 400 | 4 |
| 10 | 2 | 10 | 600 | 4 |

Рис. 3.58. Результирующая таблица выполнения команды SELECT с оконной функцией ранжирования NTILE, разбивающая данные на 4 подгруппы

#### 3.2.4.4. Оконные функции смещения

Функции смещения – это функции, которые позволяют перемещаться и, соответственно, обращаться к разным строкам в наборе данных (окне) относительно текущей строки или просто обращаться к значениям в начале или в конце окна. Эти функции появились в Microsoft SQL Server 2012.

К функциям смещения в SQL относятся:

- LEAD – функция обращается к данным из следующей строки набора данных. Ее можно использовать, например, для того чтобы сравнить значение указанного атрибута в текущей строке со значением этого атрибута в следующей строке. Функция имеет три параметра: столбец, значение которого необходимо вернуть (обязательный параметр), количество строк для смещения (по умолчанию 1), значение, которое необходимо вернуть, если после смещения возвращается значение NULL;
- LAG – функция обращается к данным из предыдущей строки набора данных. В данном случае функцию можно использовать для того, чтобы сравнить значение указанного атрибута в текущей строке со значением этого атрибута в предыдущей строке. Функция имеет три параметра: столбец, значение которого необходимо вернуть (обязательный параметр), количество строк для смещения (по умолчанию

1), значение, которое необходимо вернуть, если после смещения возвращается значение NULL;

- FIRST_VALUE — функция возвращает первое значение из набора данных. В качестве параметра функции принимается столбец, значение которого необходимо вернуть;
- LAST_VALUE — функция возвращает последнее значение из набора данных. В качестве параметра функции принимается столбец, значение которого необходимо вернуть.

Оконные функции смещения делятся на две категории. Первая категория - функции, смещение которых указывается по отношению к текущей строке. К этой группе относятся функции LAG и LEAD. В функциях второй категории смещение указывается по отношению к началу или концу окна. В эту группу входят функции FIRST_VALUE, LAST_VALUE.

Функции первой категории (LAG и LEAD) поддерживают предложение секционирования, а также упорядочения окна. Функции из второй категории (FIRST_VALUE и LAST_VALUE) помимо предложения секционирования и упорядочения окна поддерживают предложение оконного кадра.

Рассмотрим запрос, в котором используются оконные функции смещения LEAD и LAG:

```
SELECT IdGroup, IdSubGroup, Val,
        LEAD(Val) OVER (PARTITION BY
        IdGroup ORDER BY IdSubGroup) AS Lead,
        LAG(Val) OVER (PARTITION BY
        IdGroup ORDER BY IdSubGroup) AS Lag
FROM TestOver
```

На рис. 3.59 приведена итоговая таблица запроса с оконными функциями смещения LEAD и LAG.

Так как смещение явно не задано, по умолчанию предполагается смещение на единицу. Так как данные в функции секционируются по IdGroup, каждая секция будет содержать только те строки, у которых одинаковое значение атрибута IdGroup (в нашем примере три секции). Что касается упорядочения окон, то понятия «предыдущий» и «следующий» определяются упорядочением по значениям атрибута IdSubGroup. Заметьте, что в результатах запроса LAG возвращает NULL для первой строки оконной секции, потому что перед первой

строкой других строк нет. Аналогично LEAD возвращает NULL для последней строки каждой секции.

```
SELECT IdGroup, IdSubGroup, Val,
       LEAD(Val) OVER (PARTITION BY IdGroup
                       ORDER BY IdSubGroup) AS Lead,
       LAG(Val) OVER (PARTITION BY IdGroup
                      ORDER BY IdSubGroup) AS Lag
FROM TestOver
```

100 %

Results | Messages

| | IdGroup | IdSubGroup | Val | Lead | Lag |
|---|---|---|---|---|---|
| 1 | 1 | 10 | 200 | 400 | NULL |
| 2 | 1 | 10 | 400 | 300 | 200 |
| 3 | 1 | 20 | 300 | NULL | 400 |
| 4 | 2 | 10 | 400 | 600 | NULL |
| 5 | 2 | 10 | 600 | 200 | 400 |
| 6 | 2 | 20 | 200 | 100 | 600 |
| 7 | 2 | 20 | 100 | NULL | 200 |
| 8 | 3 | 10 | 300 | 400 | NULL |
| 9 | 3 | 20 | 400 | 50 | 300 |
| 10 | 3 | 20 | 50 | NULL | 400 |

Рис. 3.59. Итоговая таблица запроса с оконными функциями смещения LEAD и LAG

Если необходимо смещение, отличное от единицы, его нужно указать в качестве второго аргумента функций LAG и LEAD.

Оконные функции LAG и LEAD по умолчанию возвращают NULL, если по заданному смещению нет строки. Если нужно возвращать другое значение, можно указать его в качестве третьего аргумента функции. Например, LAG(val, 3, 0.00) возвращает «0.00», если по смещению 3 перед текущей строкой нет строки.

Функции FIRST_VALUE и LAST_VALUE возвращают запрошенные значения соответственно из первой и последней строки в окне.

Рассмотрим запрос, демонстрирующий, как возвращать значения из первой и последней строки каждого окна:

```
SELECT IdGroup, IdSubGroup, Val,
        FIRST_VALUE(Val) OVER (PARTITION BY
            IdGroup    ORDER BY IdGroup) AS FirstValue,
        LAST_VALUE (Val) OVER (PARTITION BY IdGroup
            ORDER BY IdGroup ) AS LastValue
FROM TestOver
```

Результат запроса приведен на рис. 3.60.

```
SELECT IdGroup, IdSubGroup, Val,
       FIRST_VALUE(Val) OVER (PARTITION BY IdGroup
                              ORDER BY IdGroup) AS FirstValue,
       LAST_VALUE (Val) OVER (PARTITION BY IdGroup
                              ORDER BY IdGroup ) AS LastValue
FROM TestOver
```

| | IdGroup | IdSubGroup | Val | FirstValue | LastValue |
|---|---|---|---|---|---|
| 1 | 1 | 10 | 200 | 200 | 300 |
| 2 | 1 | 10 | 400 | 200 | 300 |
| 3 | 1 | 20 | 300 | 200 | 300 |
| 4 | 2 | 10 | 400 | 400 | 100 |
| 5 | 2 | 10 | 600 | 400 | 100 |
| 6 | 2 | 20 | 200 | 400 | 100 |
| 7 | 2 | 20 | 100 | 400 | 100 |
| 8 | 3 | 10 | 300 | 300 | 50 |
| 9 | 3 | 20 | 400 | 300 | 50 |
| 10 | 3 | 20 | 50 | 300 | 50 |

Рис. 3.60. Итоговая таблица запроса с оконными функциями смещения FIRST_VALUE и LAST_VALUE

Оконную функцию можно использовать в выражениях. В следующем примере запрос возвращает, разницу между текущим значением атрибута Val и значением этого атрибута в первой и последней строке:

```
SELECT IdGroup, IdSubGroup, Val,
        Val- FIRST_VALUE(Val) OVER (PARTITION BY
            IdGroup    ORDER BY IdGroup) AS DiffFirst,
        Val- LAST_VALUE (Val) OVER (PARTITION BY
            IdGroup   ORDER BY  IdGroup  ) AS DiffLast
FROM TestOver
```

На рис. 3.61 представлена итоговая таблица этого запроса.

```
SELECT IdGroup, IdSubGroup, Val,
       Val- FIRST_VALUE(Val) OVER (PARTITION BY IdGroup
                                   ORDER BY IdGroup) AS DiffFirst,
       Val- LAST_VALUE (Val) OVER (PARTITION BY IdGroup
                                   ORDER BY  IdGroup   ) AS DiffLast
FROM TestOver
```

100 %

Results | Messages

| | IdGroup | IdSubGroup | Val | DiffFirst | DiffLast |
|---|---|---|---|---|---|
| 1 | 1 | 10 | 200 | 0 | -100 |
| 2 | 1 | 10 | 400 | 200 | 100 |
| 3 | 1 | 20 | 300 | 100 | 0 |
| 4 | 2 | 10 | 400 | 0 | 300 |
| 5 | 2 | 10 | 600 | 200 | 500 |
| 6 | 2 | 20 | 200 | -200 | 100 |
| 7 | 2 | 20 | 100 | -300 | 0 |
| 8 | 3 | 10 | 300 | 0 | 250 |
| 9 | 3 | 20 | 400 | 100 | 350 |
| 10 | 3 | 20 | 50 | -250 | 0 |

Рис. 3.61. Итоговая таблица запроса с оконными функциями смещения FIRST_VALUE и LAST_VALUE, входящими в выражения

### 3.2.4.5. Аналитические оконные функции

Аналитические оконные функции, или функции распределения (distribution function), предоставляют информацию о распределении данных. Эти функции очень специфичны и в основном используются для статистического анализа, к ним относятся:

- CUME_DIST — вычисляет и возвращает интегральное распределение значений в наборе данных. Иными словами, она определяет относительное положение значения в наборе;
- PERCENT_RANK — вычисляет и возвращает относительный ранг строки в наборе данных;
- PERCENTILE_CONT — вычисляет процентиль на основе постоянного распределения значения столбца. В качестве параметра принимает процентиль, который необходимо вычислить;
- PERCENTILE_DISC — вычисляет определенный процентиль для отсортированных значений в наборе данных. В качестве параметра принимает процентиль, который необходимо вычислить.

Функции PERCENT_RANK и CUME_DIST относятся к функциям распределения рангов. Функции PERCENTILE_CONT и PERCENTILE_DISC являются функциями обратного распределения.

*Функции распределения рангов*

Оконные функции распределения рангов поддерживают необязательное предложение секционирования и обязательное предложение упорядочения окна.

Рассмотрим пример запроса, использующего две аналитические функции PERCENTILE_CONT и PERCENTILE_DISC, которые вычисляют относительный ранг строки в секции окна, выраженный как дробное число от нуля до единицы:

```
SELECT IdGroup, IdSubGroup, Val,
        PERCENT_RANK() OVER (PARTITION BY
            IdGroup ORDER BY Val) AS PercentRank,
        CUME_DIST() OVER (PARTITION BY
            IdGroup ORDER BY Val) AS CumeDist
 FROM TestOver
```

Результаты запроса приведены на рис. 3.62.

```
SELECT IdGroup, IdSubGroup, Val,
        PERCENT_RANK() OVER (PARTITION BY IdGroup
                             ORDER BY Val) AS PercentRank,
        CUME_DIST() OVER (PARTITION BY IdGroup
                          ORDER BY Val) AS CumeDist
  FROM TestOver
```

100 %

Results | Messages

| | IdGroup | IdSubGroup | Val | PercentRank | CumeDist |
|---|---|---|---|---|---|
| 1 | 1 | 10 | 200 | 0 | 0,333333333333333 |
| 2 | 1 | 20 | 300 | 0,5 | 0,666666666666667 |
| 3 | 1 | 10 | 400 | 1 | 1 |
| 4 | 2 | 20 | 100 | 0 | 0,25 |
| 5 | 2 | 20 | 200 | 0,333333333333333 | 0,5 |
| 6 | 2 | 10 | 400 | 0,666666666666667 | 0,75 |
| 7 | 2 | 10 | 600 | 1 | 1 |
| 8 | 3 | 20 | 50 | 0 | 0,333333333333333 |
| 9 | 3 | 10 | 300 | 0,5 | 0,666666666666667 |
| 10 | 3 | 20 | 400 | 1 | 1 |

Рис. 3.62. Результирующая таблица выполнения команды SELECT с аналитическими оконными функциями PERCENT_RANK и CUME_DIST

Функции PERCENT_RANK и CUME_DIST выполняют вычисления немного по-разному. Чтобы понять смысл этих вычислений надо знать статистический анализ. Проще говоря процентильный ранг (функция PERCENT_RANK) в данном примере можно считать долей записей, у которых значение атрибута Val меньше значения этого атрибута в текущей строке, а интегральное распределение (функция CUME_DIST ) — долей записей, у которых значение атрибута Val меньше или равно этому атрибуту в текущей строке.

Значения, которые возвращают эти функции, многие воспринимает как процент.

*Функции обратного распределения*

Функции обратного распределения, более известные под именем процентилей, выполняют вычисление, которое можно считать обратным к функциям распределения рангов. Как известно, функции распределения рангов (PERCENT_RANK и CUME_DIST) вычисляют относительный ранг текущей строки в секции окна, который выражается числом от 0 до 1 (процентом). Функции обратного распределения принимают в качестве входных данных процент и возвращают значение из группы, соответствующее этому проценту. Грубо говоря, если на вход поступает процент P и упорядочение в группе основано, например, на атрибуте Val, возвращенный процентиль является значением Val, для которого доля значений Val, которые меньше процентиля, равна P. Наверное самый известный процентиль — 0,5 (50-й процентиль), более известный как медиана. К примеру, если группа состоит из значений 5, 7, 11, 101, 759, то процентиль 0,5 для нее равен 11.

Рассмотрим пример запроса, использующего две аналитические функции PERCENTILE_DISC и PERCENTILE_CONT

```
SELECT IdGroup, IdSubGroup, Val,
    PERCENTILE_DISC(0.5) WITHIN GROUP(ORDER BY Val)
        OVER (PARTITION BY IdGroup) AS PercentRank,
   PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY Val)
        OVER (PARTITION BY  IdGroup) AS PercentileCont
 FROM TestOver
```

Результаты выполнения этого запроса приведены на рис. 3.63.

```sql
SELECT IdGroup, IdSubGroup, Val,
       PERCENTILE_DISC(0.5) WITHIN GROUP(ORDER BY Val)
                 OVER (PARTITION BY IdGroup) AS PercentRank,
       PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY Val)
                OVER (PARTITION BY  IdGroup) AS PercentileCont
  FROM TestOver
```

100 %

Results | Messages

| | IdGroup | IdSubGroup | Val | PercentRank | PercentileCont |
|---|---|---|---|---|---|
| 1 | 1 | 10 | 200 | 300 | 300 |
| 2 | 1 | 20 | 300 | 300 | 300 |
| 3 | 1 | 10 | 400 | 300 | 300 |
| 4 | 2 | 20 | 100 | 200 | 300 |
| 5 | 2 | 20 | 200 | 200 | 300 |
| 6 | 2 | 10 | 400 | 200 | 300 |
| 7 | 2 | 10 | 600 | 200 | 300 |
| 8 | 3 | 20 | 50 | 300 | 300 |
| 9 | 3 | 10 | 300 | 300 | 300 |
| 10 | 3 | 20 | 400 | 300 | 300 |

Рис. 3.63. Результирующая таблица выполнения команды SELECT с аналитическими оконными функциями PERCENTILE_DISC и PERCENTILE_CONT

Функция PERCENTILE_DISC (DISC означает «discrete distribution model», то есть «модель дискретного распределения») возвращает первое значение в группе, интегральное распределение которого больше или равно входному значению, при этом предполагается, что группа трактуется как секция окна с тем же упорядочением, которое определено в группе. Посмотрите на запрос, результаты которого представлены на рис. 3.63, где вычисляется процентильный ранг и интегральное распределение. В данном случае функция

PERCENTILE_DISC(0.5) WITHIN GROUP(ORDER BY Val)
OVER(PARTITION BY IdGroup)

вернет результат 300 для секции со значением атрибута IdGroup равным 1, потому что этот результат относится к интегральному распределению 0.666666666666667, а это первое значение, большее или равное входному числу 0.5.

Оконные функции обратного распределения поддерживают необязательное предложение секционирования окна. В функциях обратного распределения также важно упорядочение, но оно не входит в определение окна, но для этого применяется отдельное предложение WITHIN GROUP.

## Вопросы для самопроверки

1. Какой принцип языка SQL лежит в основе оконных функций?
2. Какие типы оконных функций существуют в языке SQL?
3. Какова структура описания оконной функции?
4. Какие ключевые элементы входят в предложение OVER?
5. Для чего в предложении OVER используется предложение PARTITION BY?
6. Для чего в предложении OVER используется предложение ORDER BY?
7. Для чего в предложении OVER используется предложения ROWS и RANGE?
8. Какие ранжирующие функции существуют в Microsoft SQL Server?
9. Какие оконные функции смещения существуют в Microsoft SQL Server?
10. Какие аналитические оконные функции существуют в Microsoft SQL Server?

## Практические задания

Дана база данных, состоящая из четырех таблиц:

ПРОДАВЕЦ (Код_продавца, ФИО_продавца, город_продавца, комиссионные_продавца, руководитель, план продаж);

ЗАКАЗЧИК (Код_заказчика, ФИО_заказчика, город_заказчика, рейтинг заказчика, сумма_кредита); более высокий рейтинг указываетна большее предпочтение;

ЗАКАЗ (номер_заказа, сумма_заказа, дата_заказа, код_продавца, код_заказчика);

ТОВАР (Код_товара, Наименование_товара, цена, количество).

1. Напишите запрос с оконной функцией, который выбрал бы общую сумму заказа для каждого заказчика.
2. Напишите запрос с оконной функцией, который выбрал бы высший рейтинг в каждом городе.
3. Напишите запрос с оконной функцией, который вычисляет с нарастающим итогом сумму заказа для каждого продавца.
4. Напишите запрос с оконной функцией, который ранжирует товар по цене.

5. Напишите запрос с оконной функцией, который возвращает сумму текущего заказа для каждого заказчика, а также сумму заказа предыдущего и последующего заказов этого же заказчика.

### 3.3. Команды модификации данных

В этом разделе рассматриваются команды, которые управляют значениями, представленными в таблице. Значения могут быть помещены и удалены тремя командами языка манипулирования данными DML:

INSERT – команда добавления;

DELETE – команда удаления;

UPDATE – команда обновления.

#### 3.3.1. Добавление записей в таблицу

Для добавления записей используется команда INSERT, имеющая следующий формат:

INSERT INTO <Имя таблицы> [(<Список полей>)]
VALUES (<Список выражений>)

Команда INSERT служит для добавления записей в конец существующей таблицы, используя выражения, перечисленные после слова VALUES.

В предложении VALUES указываются выражения, порождающие значения полей новой записи таблицы. Выражение может включать имена полей таблицы, вызовы функций, определенных в данной СУБД, константы, знак операций сцепления строк или знаки операций. Типы значений выражений должны соответствовать типам полей таблицы. Строки и даты должны заключаться в одинарные кавычки.

Необязательная часть команды <Список полей> может быть опущена. Тогда подразумеваются все столбцы таблицы, причем в порядке, установленном при создании таблицы.

Так, например, чтобы ввести строку в таблицу Поставщик нового поставщика, можно использовать следующую команду:

INSERT INTO Поставщик
VALUES (33,'Алферов','Моск.обл.',65)

На рис. 3.64 представлена таблица Поставщик после вставки новой записи.

| | Код_постав | Название | Регион | Рейтинг |
|---|---|---|---|---|
| ▶ | 33 | Алферов | Московск.обл. | 65 |
| | 40 | Маслов | Владимир.обл. | 50 |
| | 100 | Петров А. Н. | Владимир.обл. | 80 |
| | 110 | Васильев К. Р. | Владимир.обл. | 75 |
| | 120 | Веденеев Д. В. | Владимир.обл. | 94 |
| | 130 | Логинова Н. Д. | Костром.обл. | 86 |
| | 140 | Сталь Д. П, | Воронеж.обл. | 65 |
| | 150 | Сабаев Т. Р. | Иванов.обл. | 90 |
| | 160 | Ромашкина М.... | Воронеж.обл. | 70 |
| | 170 | Григорьев Д. М. | Ивановская обл. | 76 |

Рис. 3.64. Таблица Поставщик после вставки новой строки

Таблица (в нашем случае - Поставщик), должна быть предварительно определена, а каждое значение в предложении значений, должно совпадать с типом данных столбца, в который оно вставляется.

Если порядок значений не соответствует тому порядку полей, который был установлен при создании таблицы, то после имени таблицы в скобках приводится список полей в соответствии со списком значений. Это позволяет вставлять имена в любом порядке.

Например, следующая команда дополняет таблицу Поставщик новой записью (рис. 3.65):

INSERT INTO Поставщик (Регион, Название, Код_постав, Рейтинг)

VALUES ('Владимир.обл.', 'Егоров', 2001, 76)

| | | | | |
|---|---|---|---|---|
| | 200 | Акулова М. П. | Московск.обл. | 80 |
| | 210 | Малышев Г. И. | Московск.обл. | 64 |
| | 220 | Гусь П. А. | Тверская обл. | 88 |
| | 230 | Комарова Н. В. | Нижегород.обл. | 100 |
| | 240 | Шереметьев А.... | Рязанская обл. | 96 |
| ▶ | 2001 | Егоров | Владимир.обл. | 76 |
| * | NULL | NULL | NULL | NULL |

Рис. 3.65. Таблица Поставщик после вставки строки ('Владимир.обл.', 'Егоров', 2001, 76)

Какие поля должны быть заданы при вводе данных? Это определяется тем, как описаны эти поля при создании соответствующей таблицы. Отметим, что если поле имеет признак обязательный (NOT NULL) при описании таблицы, то команда INSERT должна обяза-

тельно содержать данные для ввода в это поле каждой строки. Поэтому если в таблице все поля обязательные, то каждая вводимая строка должна содержать полный перечень вводимых значений, а указание имен полей в этом случае необязательно. В противном случае, если имеется хотя бы одно необязательное поле и вы не вводите в него значений, задание списка имен полей — обязательно.

Если в списке полей отсутствует какое-либо поле таблицы, то ему будет присвоено значение NULL или значение по умолчанию (DEFAULT), если эти ограничения определены при создании таблицы.

Предположим, что еще не известно значение поля Регион для Меньшова. Тогда новую строку можно вставить в таблицу Поставщик одной из следующих команд:

```
INSERT INTO Поставщик (Код_постав, Название, Рейтинг)
VALUES (33,'Меньшов', 74)
```

или

```
INSERT INTO Поставщик
VALUES (33,'Мньшов', NULL, 74)
```

Эти команды будут выполнены, т.к. поле Регион допускает использование значения NULL (приложение 3). Результат выполнения команды представлен на рис. 3.66.

| | Код_постав | Название | Регион | Рейтинг |
|---|---|---|---|---|
| | 33 | Алферов | Московск.обл. | 65 |
| ▶ | 35 | Меньшов | *NULL* | 74 |
| | 40 | Маслов | Владимир.обл. | 50 |
| | 100 | Петров А. Н. | Владимир.обл. | 80 |
| | 110 | Васильев К. Р. | Владимир.обл. | 75 |
| | 120 | Веденеев Д. В. | Владимир.обл. | 94 |

Рис. 3.66. Таблица Поставщик после вставки строки со значением NULL

Таким же образом можно присваивать значения полям, для которых при создании таблицы установлено значение по умолчанию (DEFAULT).

Например, команда

```
INSERT INTO Поставщик (Код_постав, Название, Регион)
VALUES (21, 'Мальцев В.Б.', 'Иванов.обл.')
```

добавит в таблицу Поставщик нового поставщика Мальцева из Ивановской области, имеющего код равный 21 и рейтинг равный 100 (рис. 3.67).

```
SELECT *
FROM Поставщик
```

100 %

Results | Messages

| | Код_постав | Название | Регион | Рейтинг |
|---|---|---|---|---|
| 1 | 21 | Мальцев В. Б. | Иванов.обл. | 100 |
| 2 | 33 | Алферов | Московск.обл. | 65 |
| 3 | 40 | Маслов | Владимир.обл. | 25 |
| 4 | 53 | Кутузов | NULL | 89 |
| 5 | 100 | Ягудин | Московск.обл. | 80 |
| 6 | 110 | Васильев К. Р. | Владимир.обл. | 37 |
| 7 | 120 | Веденеев Д. В. | Владимир.обл. | 49 |
| 8 | 130 | Логинова Н. Д. | Костром.обл. | 86 |
| 9 | 140 | Сталь Д. П, | Воронеж.обл. | 65 |
| 10 | 160 | Ромашкина М. П. | Воронеж.обл. | 70 |
| 11 | 180 | Ветров М. Д. | Костром.обл. | 68 |

Рис. 3.67. Таблица Поставщик после вставки строки со значениями (21, 'Мальцев В.Б.', 'Иванов.обл.')

Обратите внимание, что столбец Рейтинг в списке полей команды INSERT отсутствует. Это значит, что для этого поля автоматически установлено значение по умолчанию и оно равно 100 (приложение 3).

Следует отметить, что начиная с SQL Server 2008 с помощью одной команды INSERT можно вставлять несколько записей. Например, команда

```
INSERT INTO Поставщик (Код_постав, Название, Регион)
VALUES (2002, 'Аникина', 'Владимир.обл.'),
       (2003, 'Сидоров', 'Иванов.обл.')
```

добавит в таблицу Поставщик две записи (рис. 3.68).

| | | | | |
|---|---|---|---|---|
| | 240 | Шереметьев А.... | Рязанская обл. | 96 |
| | 2001 | Егоров | Владимир.обл. | 76 |
| | 2002 | Аникина | Владимир.обл. | 100 |
| ▶ | 2003 | Сидоров | Иванов.обл. | 100 |
| * | NULL | NULL | NULL | NULL |

22 of 22

Рис. 3.68. Таблица Поставщик после вставки двух новых строк

Заметим, что при вставке строки в таблицу проверяются все ограничения, наложенные на данную таблицу. Это могут быть ограничения первичного ключа или уникального индекса, проверочные ограничения типа CHECK, ограничения ссылочной целостности. В случае нарушения какого-либо ограничения вставка строки будет отклонена.

### 3.3.2. Удаление записей из таблицы

Записи из таблицы удаляются командой удаления - DELETE. Эта команда позволяет удалить одну или несколько строк из таблицы в соответствии с условиями, которые задаются для удаляемых строк. Формат команды выглядит следующим образом:

```
DELETE FROM <Имя таблицы или представления>
[WHERE <Условие>]
```

Команда удаления может удалять только введенные строки, а не индивидуальные значения полей.

Если условие не указано, то будут удалены все записи без предупреждения и без запроса на подтверждение!

Однако это не означает, что удаляется вся таблица. Структура таблицы остается. Например, чтобы удалить все содержание таблицы Поставщик, вы можете использовать следующую команду:

```
DELETE FROM Поставщик
```

Теперь, когда таблица пуста, ее можно окончательно удалить командой DROP TABLE.

Обычно, требуется удалять только некоторые определенные строки из таблицы. Чтобы определить какие строки будут удалены, используется условие предложения WHERE. Например, удалить из таблицы Поставщик Меньшова, имеющего код равный 35, можно командой

```
DELETE FROM Поставщик
WHERE Код_постав = 35
```

В данном примере в условии использовалось поле Код_постав вместо поля Название потому, что это поле является первичным ключом, что гарантирует удаление только одной строки.

Можно также использовать DELETE с условием, которое бы выбирало группу строк, как показано в этом примере:

DELETE FROM Поставщик
WHERE Регион = 'Иванов.обл'

В результате выполнения этой команды будут удалены все поставщики из Ивановской области (рис. 3.69).

| | Код_постав | Название | Регион | Рейтинг |
|---|---|---|---|---|
| ▶ | 33 | Алферов | Московск.обл. | 65 |
| | 40 | Маслов | Владимир.обл. | 50 |
| | 53 | Кутузов | NULL | 89 |
| | 100 | Петров А. Н. | Владимир.обл. | 80 |
| | 110 | Васильев К. Р. | Владимир.обл. | 75 |
| | 120 | Веденеев Д. В. | Владимир.обл. | 94 |
| | 130 | Логинова Н. Д. | Костром.обл. | 86 |
| | 140 | Сталь Д. П, | Воронеж.обл. | 65 |
| | 160 | Ромашкина М.... | Воронеж.обл. | 70 |
| | 180 | Ветров М. Д. | Костром.обл. | 68 |
| | 190 | Филатов П. С. | Московск.обл. | 85 |
| | 200 | Акулова М. П. | Московск.обл. | 80 |
| | 210 | Малышев Г. И. | Московск.обл. | 64 |
| | 220 | Гусь П. А. | Тверская обл. | 88 |
| | 230 | Комарова Н. В. | Нижегород.обл. | 100 |
| | 240 | Шереметьев А.... | Рязанская обл. | 96 |
| | 2001 | Егоров | Владимир.обл. | 76 |
| | 2002 | Аникина | Владимир.обл. | 100 |
| * | NULL | NULL | NULL | NULL |

Рис. 3.69. Таблица Поставщик после удаления поставщиков из Ивановской области

### 3.3.3. Изменение значений полей таблицы

В SQL для изменения существующих данных в таблице используется команда UPDATE, которая имеет следующий синтаксис:

UPDATE <Имя таблицы>
SET Столбец1 = Выражение1
[, Столбец2 = Выражение2,…]
[WHERE <Условие>]

Команда изменяет в указанной таблице значения указанных полей тех записей, которые удовлетворяют заданному условию отбора (WHERE <условие>). Если условие не указано, изменения применяются ко всем записям таблицы. Например, изменить рейтинги всех поставщиков на 200 можно с помощью команды

```
UPDATE Поставщик
SET Рейтинг = 200
```

Конечно, редко требуется указывать все строки таблицы для изменения единственного значения. Поэтому команда UPDATE как правило содержит условие. Вот как, например можно выполнить изменение для поставщика с кодом равным 120:

```
UPDATE Поставщик
SET Рейтинг = 98
WHERE Код_постав = 120
```

С помощью одной команды могут быть заданы значения для любого количества столбцов.

Предположим, что поставщик Петров (код поставщика 100) ушел на пенсию, и мы хотим переназначить его код новому поставщику Ягудину из Московской области:

```
UPDATE Поставщик
SET Название = 'Ягудин', Регион = 'Московск.обл.'
 WHERE Код_постав = 100
```

Результат запроса представлен на рис. 3.70.

| | Код_постав | Название | Регион | Рейтинг |
|---|---|---|---|---|
| | 33 | Алферов | Московск.обл. | 65 |
| | 40 | Маслов | Владимир.обл. | 50 |
| | 53 | Кутузов | *NULL* | 89 |
| ▶ | 100 | Ягудин | Московск.обл. | 80 |
| | 110 | Васильев К. Р. | Владимир.обл. | 75 |
| | 120 | Веденеев Д. В. | Владимир.обл. | 98 |

Рис. 3.70. Таблица Поставщик после выполнения команды UPDATE

В предложении SET команды UPDATE допускается использовать выражения.

Предположим, что надо уменьшить в два раза рейтинг всем поставщикам во Владимирской области:

```
UPDATE Поставщик
SET Рейтинг  = Рейтинг * 0.5
WHERE Регион = 'Владимир.обл.'
```

Результат запроса представлен на рис. 3.71.

|  | Код_постав | Название | Регион | Рейтинг |
|---|---|---|---|---|
|  | 33 | Алферов | Московск.обл. | 65 |
|  | 40 | Маслов | Владимир.обл. | 25 |
|  | 53 | Кутузов | NULL | 89 |
|  | 100 | Ягудин | Московск.обл. | 80 |
|  | 110 | Васильев К. Р. | Владимир.обл. | 37 |
|  | 120 | Веденеев Д. В. | Владимир.обл. | 49 |
|  | 130 | Логинова Н. Д. | Костром.обл. | 86 |
|  | 140 | Сталь Д. П, | Воронеж.обл. | 65 |
|  | 160 | Ромашкина М.... | Воронеж.обл. | 70 |
|  | 180 | Ветров М. Д. | Костром.обл. | 68 |
|  | 190 | Филатов П. С. | Московск.обл. | 85 |
|  | 200 | Акулова М. П. | Московск.обл. | 80 |
|  | 210 | Малышев Г. И. | Московск.обл. | 64 |
|  | 220 | Гусь П. А. | Тверская обл. | 88 |
|  | 230 | Комарова Н. В. | Нижегород.обл. | 100 |
|  | 240 | Шереметьев А.... | Рязанская обл. | 96 |
|  | 2001 | Егоров | Владимир.обл. | 38 |
| ▶ | 2002 | Аникина | Владимир.обл. | 50 |
| * | NULL | NULL | NULL | NULL |

Рис. 3.71. Таблица Поставщик после выполнения команды UPDATE

Предложение SET - это не предикат, поэтому оно может вводить неопределенные значения NULL. Так что, если требуется установить все рейтинги поставщиков в Московской области в NULL, то можно воспользоваться следующей командой:

```
UPDATE Поставщик
SET Рейтинг = NULL
WHERE Рейтинг = 'Моск.обл.'
```

### 3.3.4. Использование подзапросов с командами модификации

Подзапросы могут использоваться и с командами языка манипулирования данными (DML).

Важный принцип, который надо соблюдать при работе с командами модификации, состоит в том, что нельзя в предложении FROM любого подзапроса, указывать таблицу, к которой ссылаетесь с помощью основной команды. Это относится ко всем трем командам модификации.

*Использование подзапросов с командой INSERT*

Команда INSERT использует данные, возвращаемые подзапросом, для помещения их в другую таблицу.

Базовый синтаксис соответствующей команды должен быть следующим:

```
INSERT INTO <Имя таблицы> [(<Список полей>)]
<команда SELECT>
```

Эта команда используется для ввода в заданную таблицу новых строк, отобранных из другой таблицы с помощью команды SELECT.

Рассмотрим пример использования этой команды. Пусть таблица Владимир имеет структуру, полностью совпадающую со структурой таблицы Поставщик. Запрос, позволяющий заполнить таблицу Владимир записями из таблицы Поставщик обо всех поставщиках из Владимирской области, выглядит следующим образом:

```
INSERT INTO Владимир
SELECT *
FROM Поставщик
WHERE Регион = 'Владимир.обл.'
```

Эта команда выбирает все строки из таблицы Поставщик со значениями Регион = 'Владимир.обл.' и помещает в таблицу Владимир. Чтобы это работало, таблица Владимир должна отвечать следующим условиям:

- она должна уже быть создана;
- она должна иметь четыре столбца, которые совпадают с таблицей Поставщик в терминах типа данных; то есть первый, второй, и так далее, столбцы каждой таблицы, должны иметь одинаковый тип данных.

Использование в подзапросе символа «*» является в данном случае оправданным, так как порядок следования столбцов является одинаковым для обеих таблиц. Если бы это было не так, следовало бы применить список столбцов либо в команде INSERT, либо в подзапросе, либо в обоих местах, который приводил бы в соответствие порядок следования столбцов.

Данные таблицы Владимир представлены на рис. 3.72.

```
SELECT *
FROM Владимир
```

100 %

Results | Messages

| | Код_постав | Название | Регион | Рейтинг |
|---|---|---|---|---|
| 1 | 40 | Маслов | Владимир.обл. | 25 |
| 2 | 110 | Васильев К. Р. | Владимир.обл. | 37 |
| 3 | 120 | Веденеев Д. В. | Владимир.обл. | 49 |
| 4 | 2001 | Егоров | Владимир.обл. | 38 |
| 5 | 2002 | Аникина | Владимир.обл. | 50 |

Рис. 3.72. Данные таблицы Владимир

Предположим, например, необходимо сформировать новую таблицу с именем Итоги_дня, которая будет следить за общим количеством сумм поступлений за каждый день. Заполнить таблицу Итоги_дня надо информацией представленной в таблице Приход. Для этого воспользуемся следующей командой:

```
INSERT INTO Итоги_дня (Дата, Сумма)
SELECT Дата, Sum(Цена * Количество)
FROM Документ, Приход
WHERE Документ.Ном_док = Приход.Ном_док
GROUP BY Дата
```

Данные таблицы Владимир представлены на рис. 3.73.

```
SELECT *
FROM Итоги_дня
```

100 %

Results | Messages

| | Дата | Сумма |
|---|---|---|
| 1 | 2020-10-20 | 65200,00 |
| 2 | 2020-10-23 | 32600,00 |
| 3 | 2020-11-05 | 39900,00 |
| 4 | 2020-11-07 | 41600,00 |
| 5 | 2020-11-14 | 48930,00 |
| 6 | 2020-11-25 | 111300,00 |
| 7 | 2021-03-29 | 38320,00 |
| 8 | 2021-04-23 | 19160,00 |
| 9 | 2021-05-16 | 83950,00 |
| 10 | 2021-05-17 | 622960,00 |
| 11 | 2021-06-11 | 317712,00 |
| 12 | 2021-07-04 | 240240,00 |

Рис. 3.73. Данные таблицы Итоги_дня

*Использование подзапросов с командой DELETE*

Использование подзапроса в предложение WHERE команды DELETE дает возможность задавать некоторые довольно сложные условия, чтобы установить, какие строки будут удаляться.

Как отмечалось выше при работе с командами модификации нельзя ссылаться к таблице, из которой вы будете удалять строки в предложении FROM подзапроса. Однако, в условии подзапроса можно сослаться на текущую строку этой таблицы.

Пример, удалить все поступления товара за 25 октября 2020 года от поставщика Гусь П.А.

```
DELETE FROM Приход
WHERE EXISTS
        ( SELECT *
          FROM Документ d, Поставщик p
          WHERE Дата = ’25.11.2020’
                  AND Название = ‘Гусь П.А.’
                  AND  Приход._Ном_док = d._Ном_док
                 AND d. Код_постав = p.Код_постав)
```

Обратите внимание, что в предложении WHERE внутреннего запроса имеется ссылка (Приход.Ном_док) к таблице Приход. Это означает, что весь подзапрос будет выполняться отдельно для каждой строки таблицы Приход, т.е. подзапрос является коррелированным.

На рис. 3.74, а приведены данные таблицы Приход до удаления, а на рис. 3.74, б – после удаления.

DESKTOP-GI90AK0\...а БД - dbo.Приход | SQLQuery1.sql - DE...-GI90AK0\User (61))*

| Ном_док | Ном_ном | Тип | Цена | Количество |
|---|---|---|---|---|
| 1000 | 10010 | Принтер | 6520,0000 | 10 |
| 1200 | 10010 | Принтер | 6520,0000 | 5 |
| 1300 | 10020 | Принтер | 3990,0000 | 10 |
| 1500 | 10030 | Принтер | 5200,0000 | 8 |
| 1710 | 10040 | Принтер | 6990,0000 | 7 |
| 1850 | 10050 | Монитор | 9820,0000 | 6 |
| 2500 | 10070 | Монитор | 4790,0000 | 8 |
| 2790 | 10070 | Монитор | 4790,0000 | 4 |
| 6300 | 10080 | Монитор | 16790,0000 | 5 |
| 7420 | 10090 | Мат. плата | 47520,0000 | 8 |
| 8750 | 10100 | Мат. плата | 7632,0000 | 26 |
| 8750 | 10110 | Мат. плата | 3408,0000 | 35 |
| 9680 | 10120 | Процессор | 34320,0000 | 7 |
| 7420 | 10130 | Процессор | 48560,0000 | 5 |

*а*

DESKTOP-GI90AK0\...а БД - dbo.Приход | SQLQuery1.sql - DE...-GI90AK0\User (61))*

| Ном_док | Ном_ном | Тип | Цена | Количество |
|---|---|---|---|---|
| 1000 | 10010 | Принтер | 6520,0000 | 10 |
| 1200 | 10010 | Принтер | 6520,0000 | 5 |
| 1300 | 10020 | Принтер | 3990,0000 | 10 |
| 1500 | 10030 | Принтер | 5200,0000 | 8 |
| 1710 | 10040 | Принтер | 6990,0000 | 7 |
| 2500 | 10070 | Монитор | 4790,0000 | 8 |
| 2790 | 10070 | Монитор | 4790,0000 | 4 |
| 6300 | 10080 | Монитор | 16790,0000 | 5 |
| 7420 | 10090 | Мат. плата | 47520,0000 | 8 |
| 8750 | 10100 | Мат. плата | 7632,0000 | 26 |
| 8750 | 10110 | Мат. плата | 3408,0000 | 35 |
| 9680 | 10120 | Процессор | 34320,0000 | 7 |
| 7420 | 10130 | Процессор | 48560,0000 | 5 |
| NULL | NULL | NULL | NULL | NULL |

*б*

Рис. 3.74. Результат выполнения команды DELETE

Transact-SQL расширяет синтаксис команды DELETE, вводя дополнительное предложение FROM.

При помощи этого предложения можно выполнять соединения таблиц, что логически заменяет использование подзапросов в предложении WHERE для идентификации удаляемых строк. Предыдущий пример можно записать с помощью дополнительного предложения FROM следующим образом:

```
DELETE FROM Приход
FROM Документ d, Поставщик p
WHERE Дата = ’25.11.2020’
        AND Название = ‘Гусь П.А.’
        AND  Приход._Ном_док = d._Ном_док
        AND d. Код_постав = p.Код_постав
```

*Использование подзапросов с командой UPDATE*

Команда UPDATE использует подзапросы в предложении WHERE .

Рассмотрим пример, поясняющий использование команды UPDATE с подзапросом: увеличить рейтинг поставщика на 100, если он не менее 2 раз поставлял товар:

```
UPDATE Поставщик
SET Рейтинг  =  Рейтинг + 100
WHERE 2 < = ( SELECT COUNT (Ном_док)
                FROM Документ d
               WHERE Поставщик.Код_постав = d.Код_постав)
```

Данные таблицы Поставщик после выполнения команды UPDATE  представлены на рис. 3.75.

**Вопросы для самопроверки**

1. Какие команды SQL используются:
   а) для вставки строки в таблицу;
   б) удаления строки из таблицы;
   в) обновления полей таблицы?

2. В каком предложении команды INSERT указываются вставляемые в таблицу значения?

3. Какое соответствие должно быть между списком столбцов и списком значений в команде INSERT?

| Код_постав | Название | Регион | Рейтинг |
|---|---|---|---|
| 21 | Мальцев В. Б. | Иванов.обл. | 100 |
| 33 | Алферов | Московск.обл. | 65 |
| 40 | Маслов | Владимир.обл. | 25 |
| 53 | Кутузов | NULL | 89 |
| 100 | Ягудин | Московск.обл. | 80 |
| 110 | Васильев К. Р. | Владимир.обл. | 37 |
| 120 | Веденеев Д. В. | Владимир.обл. | 149 |
| 130 | Логинова Н. Д. | Костром.обл. | 86 |
| 140 | Сталь Д. П, | Воронеж.обл. | 165 |
| 160 | Ромашкина М.... | Воронеж.обл. | 170 |
| 180 | Ветров М. Д. | Костром.обл. | 68 |
| 190 | Филатов П. С. | Московск.обл. | 185 |
| 200 | Акулова М. П. | Московск.обл. | 80 |
| 210 | Малышев Г. И. | Московск.обл. | 64 |
| 220 | Гусь П. А. | Тверская обл. | 88 |
| 230 | Комарова Н. В. | Нижегород.обл. | 200 |
| 240 | Шереметьев А.... | Рязанская обл. | 96 |
| 2001 | Егоров | Владимир.обл. | 38 |
| 2002 | Аникина | Владимир.обл. | 50 |
| NULL | NULL | NULL | NULL |

Рис. 3.75. Данные таблицы Поставщик после выполнения команды UPDATE

4. Какие служебные слова могут использоваться в команде DELETE?

5. С помощью какого предложения команды DELETE может указываться удаляемая строка?

6. Какие строки будут удалены командой DELETE, если не указать условие?

7. Какие служебные слова могут использоваться в команде UPDATE?

8. В каком предложении команды UPDATE указываются вставляемые в таблицу значения?

9. Какие строки будут изменены командой UPDATE, если не указать условие?

10. Для чего используются подзапросы в командах INSERT INTO, DELETE FROM и UPDATE?

**Практические задания**

1. Дана база данных, состоящая из четырех таблиц:

КЛИЕНТ (Код_клиента, Фамилия, Имя, Отчество, Город*);*
ПРОЖИВАЕТ(Код_клиента, Номер, Дата_прибытия, Дата_убытия);
НОМЕР (Номер, Число мест, Этаж);
ТИП НОМЕРА (Число мест, Цена)

- Добавьте клиента Курочкина Афанасия Егоровича, прибывшего из Калуги, в таблицу КЛИЕНТ.
- Увеличьте цену проживания в гостинице на 15%.
- Удалите из базы данных клиента Иванова Ивана Ивановича из Костромы.

2. Дана база данных КЛИЕНТЫ:

КЛИЕНТ (код клиента, наименование, годовой доход, тип заказчика)
ОТГРУЗКА (номер отгрузки, код клиента, вес, номер грузовика, название города, дата)
ВОДИТЕЛЬ (номер отгрузки, имя водителя)
ГОРОД (название, число жителей)

Удалите из базы данных все города с населением до 5000 человек. Не забудьте обновить таблицу ОТГРУЗКА.

3. 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 команду, которая бы удалила все покупки заказчика Ершова из таблицы Покупка
- Напишите команду, которая бы удаляла всех заказчиков, не имеющих текущих покупок.
- Напишите команду, которая бы увеличила на двадцать процентов комиссионные всех продавцов, имеющую сумму покупок выше, чем 100000 руб.

## 3.4. Представления

### 3.4.1. Введение в представления

Таблицы, с которыми мы имели дело раньше, называются - *базовыми таблицами*. Эти таблицы определяют структуру базы данных и содержат данные, которые постоянно находятся на устройстве хранения информации.

Однако имеется другой вид таблиц: - представления (VIEW, обзор, взгляд).

Представления являются объектами базы данных, информация в которых не хранится постоянно, как в базовых таблицах, а формируется динамически при обращении к ним.

Для пользователя базы данных представление выглядит подобно обычной таблице, состоящей из строк и столбцов. Однако, в отличие от таблицы, представление как совокупность значений в базе данных реально не существует. Строки и столбцы данных, которые пользователь видит с помощью представления, являются результатами запроса, лежащего в его основе.

Представление – это хранимый запрос, создаваемый на основе команды SELECT.

При создании представление получает имя и его определение сохраняется в базе данных.

Поэтому часто их называют *виртуальными* таблицами, поскольку такая таблица не существует как независимый объект в базе данных.

Запрос, определяющий представление, выполняется тогда, когда к представлению происходит обращение с другим запросом, например, SELECT, UPDATE и т.д.

Представления - подобны окнам, через которые можно просматривать информацию, которая фактически хранится в базовых таблицах.

Назначение представлений:

1. Хранение сложных запросов.

2. Представление данных в виде, удобном пользователю. Они показывают каждому пользователю структуру хранимых данных в наиболее подходящем для него виде.

3. С помощью представлений можно ограничить доступ к данным, разрешая пользователям видеть только некоторые из строк и столбцов таблицы.

### 3.4.2. Создание представлений

Создание представлений выполняется командой CREATE VIEW, базовый синтаксис которой выглядит следующим образом:

```
CREATE VIEW имя_представления [(столбец1, столбец2, ...)]
[WITH ENCRYPTION ]
AS
команда SELECT
[WITH CHECK OPTION]
```

Не трудно заметить, что создать представление значительно проще, чем создать таблицу, — не нужно указывать практически никаких дополнительных параметров, кроме имени таблицы или другого представления, на основе которого оно создается.

Рассмотрим параметры команды CREATE VIEW:

столбец — это своего рода псевдоним, используемый для столбца в представлении. Этот параметр является обязательным, когда столбец представления создается на базе арифметического выражения, функции, или когда несколько столбцов исходных таблиц или представлений имеют одинаковые имена. Имена можно указать в списке имён после имени представления, а можно указать псевдонимы для отдельных столбцов результата (например, select Naimen AS ‘Название товара’). Если не указывать имена столбцов, то в представлении они получат имена, которые указаны в команде SELECT.

WITH ENCRYPTION предписывает серверу шифровать SQL-код запроса. Это гарантирует невозможность его несанкционированного просмотра и использования. Этот аргумент применяется, если при определении представления необходимо скрыть имена исходных таблиц и столбцов, а также алгоритм объединения данных.

AS — ключевое слово, показывающее начало определения представления.

Команда SELECT — команда, определяющая собственно представление. Элементы этой команды ничем не отличаются от тех, которые мы рассматривали в разделе 3.2.

WITH CHECK OPTION — предписывает серверу исполнять проверку при модификации данных на соответствие условиям на значения (ограничения), которые были определены для таблиц, лежащих в основе создаваемого представления. Включение в команду этого параметра гарантирует, что сделанные изменения будут отображены в представлении. Если пользователь пытается выполнить изменения, приводящие к исключению строки из представления, при заданном аргументе WITH CHECK OPTION сервер выдаст сообщение об ошибке и все изменения будут отклонены.

Рассмотрим примеры использования представлений.

Как отмечалось выше, представления могут ограничивать доступ к строкам. Выбираемые представлением строки базовой таблицы задаются условием (предикатом) в предложении WHERE при описании представления. Доступ через представление возможен только к строкам, удовлетворяющим условию.

Пример: создадим представление, которое позволит просматривать поставщиков из Владимирской обл.:

```
CREATE VIEW Поставщики_Владимира
 AS
 SELECT *
 FROM Поставщики
 WHERE Регион = 'Владимир.обл.'
```

Теперь в базе данных хранится представление, называемое Поставщики_Владимира и к нему можно обращаться с помощью запросов так же, как и к любой другой таблице базы данных, например, выбрать из представления Поставщики_Владимира поставщиков, название которых начинается на букву ‘В’:

```
SELECT *
FROM Поставщики_Владимира
WHERE Название LIKE 'В%'
```

Результат выполнения запроса приведен на рис. 3.76.

```
SELECT *
FROM Поставщики_Владимира
WHERE Название LIKE 'В%'
```

100 %

Results | Messages

| | Код_постав | Название | Регион | Рейтинг |
|---|---|---|---|---|
| 1 | 110 | Васильев К. Р. | Владимир.обл. | 75 |
| 2 | 120 | Веденеев Д. В. | Владимир.обл. | 94 |

Рис. 3.76. Результат запроса, источником данных которого является представление Поставщики_Владимира

Преимущество использования представления, по сравнению с базовой таблицей, в том, что представление будет модифицировано автоматически всякий раз, когда таблица, лежащая в его основе, изменяется. Если добавить в таблицу Поставщик нового поставщика, живущего во Владимире, он автоматически появится в представлении.

Представления значительно расширяют управление данными. Это превосходный способ дать публичный доступ к некоторой, но не всей информации в таблице. Например, если требуется, чтобы поставщик был показан в таблице Поставщик, но при этом не были показаны рейтинги других поставщиков, можно создать следующее представление:

```
CREATE VIEW Поставщики
AS
SELECT Название, Регион, Код_постав
FROM Поставщик
```

Данное представление ограничивает число столбцов базовой таблицы, к которым возможен доступ.

Обращаем внимание, что в рассмотренных примерах, поля представлений имеют имена, совпадающие с именами полей базовой таблицы.

*Групповые представления*

Запрос, определяющий представление, может содержать предложение GROUP BY.

Пример: каждый день необходимо следить за количеством поставок, количеством поставщиков, средней суммой поставок, общей суммой поставок за каждый день. Чем конструировать каждый раз сложный запрос, можно просто создать следующее представление:

```
CREATE VIEW Итоги_за_день (Дата_поставки, Количество_поставок, Количество_ поставщиков, Средняя_сумма, Общая_сумма)
AS
SELECT Дата, COUNT(Документ.Ном_док), COUNT(DISTINCT Код_постав), AVG(Цена*Количество), SUM(Цена*Количество)
FROM Документ, Приход
WHERE Документ.Ном_док = Приход.Ном_док
GROUP BY Дата
```

Теперь вы сможете получить данные из представления Итоги_за_день, относящиеся, например, к 17.05.2021, с помощью простого запроса:

```
SELECT *
FROM Итоги_за_день
WHERE Дата_поставки = '17.05.2021'
```

Результат выполнения запроса приведен на рис. 3.77.

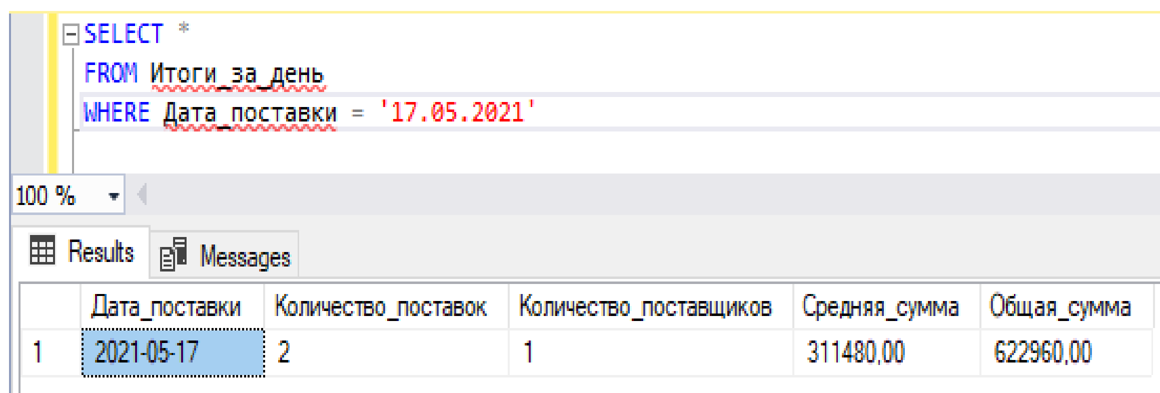



Рис. 3.77. Результат запроса, источником данных которого является представление Итоги_за_день

*Представления и соединения*

Представление может выводить информацию из любого числа базовых таблиц или из других представлений. Такое решение можно использовать при формировании сложных отчетов как промежуточный макет, скрывающий детали объединения большого количества исходных таблиц.

Например, создать представление, которое показывало бы тип товара, его наименование и фамилию поставщика, который его поставлял:

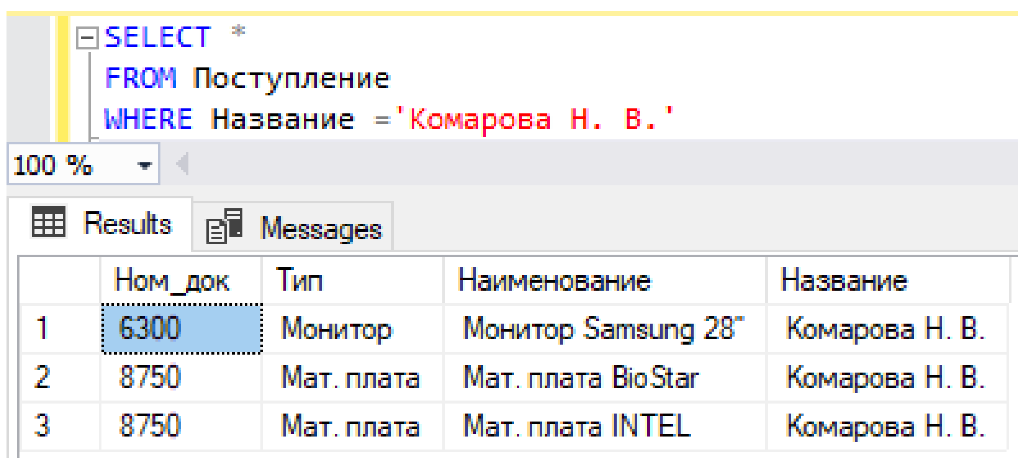
```
CREATE VIEW Поступление
AS
SELECT Приход.Ном_док, Тип, Наименование, Название
FROM Приход, Документ, Поставщик, Склад
WHERE Приход.Ном_док = Документ.Ном_док
AND Документ.Код_постав = Поставщик.Код_постав
AND Приход.Ном_ном = Склад.Ном_ном
```

Теперь можно выбрать все товары, поставленные конкретным поставщиком, или можно увидеть эту информацию для любого документа.

Например, чтобы увидеть все поставки Комаровой Н.В., требуется ввести следующий запрос:

```
SELECT *
FROM Поступление
WHERE Название = 'Комарова Н.В.'
```

Результат выполнения запроса приведен на рис. 3.78.




| | Ном_док | Тип | Наименование | Название |
|---|---|---|---|---|
| 1 | 6300 | Монитор | Монитор Samsung 28" | Комарова Н. В. |
| 2 | 8750 | Мат. плата | Мат. плата Bio Star | Комарова Н. В. |
| 3 | 8750 | Мат. плата | Мат. плата INTEL | Комарова Н. В. |

Рис. 3.78. Результат запроса, источником данных которого является представление Поступление

*Представления и подзапросы*

Представления могут также использовать и подзапросы.

Предположим, компания даст премию поставщикам, поставляющим самые дорогие товары в 2020г. Вы можете проследить эту информацию с помощью представления:

```
CREATE VIEW Премия
AS
SELECT A.Код_постав, Дата, Цена, Название
FROM Документ A, Приход B, Поставщик C
WHERE A.Код_постав = C.Код_постав
AND A.Ном_док = B.Ном_док
AND Цена = (SELECT MAX(Цена)
            FROM Приход, Документ
            WHERE YEAR (Дата) = 2020
           AND Документ.Ном_док = Приход.Ном_док)
```

Извлечение из этой таблицы поставщиков, которые будут получать премию, выполняется простым запросом:

```
SELECT *
FROM Премия
```

Результат выполнения запроса приведен на рис. 3.79.

```
SELECT *
FROM Премия
```

100 %

Results | Messages

| | Код_постав | Дата | Цена | Название |
|---|---|---|---|---|
| 1 | 220 | 2020-11-25 | 9820,00 | Гусь П. А. |

Рис. 3.79. Результат запроса, источником данных которого является представление Премия

### 3.4.3. Изменение значений базовой таблицы с помощью представлений

Один из наиболее трудных и неоднозначных аспектов представлений – это непосредственное их использование с командами модификации. Представление может изменяться командами модификации. Однако фактически модификация воздействует не на само представление, а на базовую таблицу.

Если команды модификации могут выполняться в представлении, представление будет *модифицируемым*. В противном случае оно предназначено только для чтения при запросе.

Представление является модифицируемым, если оно удовлетворяет следующим требованиям:

- Должен отсутствовать предикат DISTINCT, то есть повторяющиеся строки не должны исключаться из таблицы результатов запроса.
- Каждое имя в списке возвращаемых столбцов должно быть ссылкой на простой столбец. В этом списке не должны содержаться выражения, вычисляемые столбцы или агрегатные функции.
- В предложении FROM должна быть задана только одна таблица, которую можно обновлять, то есть у представления должна быть одна исходная таблица и пользователь должен иметь соответствующие права доступа к ней. Если исходная таблица сама является представлением, то оно также должно удовлетворять этим условиям.
- Предложение WHERE не должно содержать подзапросов.
- В запросе не должны содержаться предложения GROUP BY и HAVING.

Перечисленные требования базируются на принципе, запомнить который, пожалуй, легче, чем сами требования: представление разрешается обновлять в том случае, если система управления базами данных может для каждой строки представления найти исходную строку в исходной таблице, а для каждого обновляемого столбца представления — исходный столбец в исходной таблице. Если представление соответствует этим требованиям, то к нему могут применяться команды INSERT, UPDATE, DELETE.

Чтобы лучше понять смысл этих ограничений, рассмотрим примеры модифицируемых представлений и представлений только для чтение:

```
CREATE VIEW Дата_заказа (Дата, Количество_заказов)
AS
SELECT Дата, COUNT (*)
FROM Заказы
GROUP BY Дата
```

Это представление только для чтения из-за присутствия в нем агрегатной функции и GROUP BY.

```
CREATE VIEW Москва
AS
SELECT *
FROM Поставщик
WHERE Регион = 'Моск.обл.'
```

А это - представление модифицируемое.

```
CREATE VIEW Влад ( Код, Название_товара, Цена_с_НДС)
AS
SELECT Ном_ном, Наименование, Цена * 1.2
FROM Склад
```

Это представление только для чтение из-за выражения Цена * 1.2.

Модифицируемые и немодифицируемые представления создаются для различных целей.

С модифицируемыми представлениями в основном работают так же, как и с базовыми таблицами. Пользователи могут даже не знать, является ли объект, который они запрашивают, базовой таблицей или представлением. Таким образом, представление – это прежде всего средство для скрытия частей таблицы, не относящихся к потребностям данного пользователя.

Немодифицируемые представления позволяют получать и форматировать данные более рационально. Они создают целый набор сложных запросов, которые можно выполнить и повторить снова, сохраняя полученную информацию. Результаты этих запросов могут затем использоваться в других запросах, что позволит избежать сложных предикатов и снизить вероятность ошибочных действий.

Эти представления могут также иметь значение при решении задач защиты и безопасности данных. Например, можно предоставить некоторым пользователям возможность получения агрегатных данных (таких, как усредненное значение рейтинга поставщиков), не показывая конкретных значений рейтинга и, тем более, не позволяя их модифицировать.

*Добавление строк*

Рассмотрим пример использования представления в команде INSERT для добавления строк в таблицу, на которой это представление определено.

Создадим представление:

```
CREATE VIEW N1
AS
SELECT *
FROM Поставщик
WHERE Рейтинг = 80
```

Это представление модифицируемое. Оно просто ограничивает доступ к определенным строкам в таблице.

Вставим новую строку:

```
INSERT INTO N1
VALUES (40,'Маслов','Владимир.обл.', 50)
```

Это допустимая команда INSERT. Строка будет вставлена с помощью представления N1 в таблицу Поставщик (рис. 3.80).

| Код_постав | Название | Регион | Рейтинг |
|---|---|---|---|
| 40 | Маслов | Владимир.обл. | 50 |
| 100 | Петров А. Н. | Владимир.обл. | 80 |
| 110 | Васильев К. Р. | Владимир.обл. | 75 |
| 120 | Веденеев Д. В. | Владимир.обл. | 94 |
| 130 | Логинова Н. Д. | Костром.обл. | 86 |
| 140 | Сталь Д. П, | Воронеж.обл. | 65 |
| 150 | Сабаев Т. Р. | Иванов.обл. | 90 |
| 160 | Ромашкина М.... | Воронеж.обл. | 70 |
| 170 | Григорьев Л. М. | Ивановская обл. | 76 |
| 180 | Ветров М. Д. | Костром.обл. | 68 |
| 190 | Филатов П. С. | Московск.обл. | 85 |

Рис. 3.80. Результат вставки строки в таблицу Поставщик

Однако когда она появится там, она исчезнет из представления N1, поскольку значение рейтинга не равно 80 (рис. 3.81).

```
SELECT *
FROM N1
```

100 % Results Messages

| | Код_постав | Название | Регион | Рейтинг |
|---|---|---|---|---|
| 1 | 100 | Ягудин | Московск.обл. | 80 |
| 2 | 200 | Акулова М. П. | Московск.обл. | 80 |

Рис. 3.81. Данные представления N1

Иногда такой подход может стать проблемой, так как данные уже находятся в базовой таблице, но пользователь их не видит в представлении и не в состоянии выполнить их удаление или модификацию.

Для исключения подобных моментов служит опция WITH CHECK OPTION в определении представления. Фраза размещается в определении представления и все команды модификации будут подвергаться проверке.

Если в представление N1 добавить опцию WITH CHECK OPTION, то определение представления будет выглядеть следующим образом:

```
CREATE VIEW N2
AS
SELECT *
FROM Поставщик
WHERE Рейтинг = 50
WITH CHECK OPTION
```

Теперь, если выполнить запрос на вставку новой записи

```
INSERT INTO N3
VALUES (45,'Кокорин','Московск. обл.', 70),
```

будет выдано сообщение (рис. 3.82).

```
INSERT INTO N3
VALUES (45,'Кокорин А.А.', 'Московск.обл.', 70)
```

100 %

Messages

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

Completion time: 2020-12-05T11:50:38.4252130+03:00
```

Рис. 3.82. Сообщение о невозможности выполнить операцию вставки

### 3.4.4. Удаление представлений

Для удаления представлений из базы данных можно воспользоваться командой DROP VIEW языка Transact-SQL. Синтаксис команды DROP VIEW:

DROP VIEW имя__представления

В приведенном ниже примере удаляется представление с именем view1:

DROP VIEW view1

Удаление представления не влияет на таблицы, лежащие в его основе. Определение представления просто удаляется из базы данных.

#### Вопросы для самопроверки

1. Что такое представления?
2. В чем различие между представлениями и таблицами?
3. Каким целям служат представления?
4. Должно ли представление иметь одинаковое имя с таблицей, от которой порождено?
5. Возможно ли создание представления, включающего информацию из нескольких таблиц одновременно?
6. В чем различие между модифицируемыми и немодифицируемыми представлениями?

7. Каким требованиям должны удовлетворять модифицируемые представления?

8. С какой целью в команде CREATE VIEW используется опция WITH CHECK OPTION?

9. Какая команда используется для удаления представления из базы данных?

**Практические задания**

1. 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 представление, которое выводит список городов, в которых есть заказчики продавца Петрова.
- Напишите представление, в котором определяется число заказчиков, имеющих рейтинг > 100, в каждом городе.
- Напишите представление, которое бы показывало всех продавцов, имеющих самые высокие комиссионные.

2. Какое из этих представлений - модифицируемое? Объясните причину, по которой представление не является модифицируемым

№1
```
CREATE VIEW Dailyorders
  AS
  SELECT DISTINCT num, num, num, odate
  FROM Orders
```

№2
```
CREATE VIEW Custotals
    AS
    SELECT name, SUM (amt) AS 'Сумма'
    FROM Orders, Customers
    WHERE Orders.cnum = customer.cnum
    GROUP BY cname
```

№3
```
CREATE VIEW Thirdorders
    AS
    SELECT *
    FROM Dailyorders
    WHERE date = '10.03.2019'
```

# ЗАКЛЮЧЕНИЕ

Технология баз данных появилась почти полвека назад и за этот небольшой срок существенно изменила методы работы многих организаций и предприятий. В настоящее время сложно найти такую компанию, в которой не использовались бы информационные системы, основанные на базах данных.

Умение грамотно работать с базами данных – одно из ключевых требований к любому специалисту в области компьютерных технологий. Важную роль в освоении технологий баз данных играет понимание их теоретических основ. Учебное пособие нацелено именно на ознакомление студентов бакалавриата с фундаментальными понятиями баз данных и основано на материалах лекций, которые читаются автором в Институте информационных технологий и радиоэлектроники Владимирского государственного университета.

# БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Дейт, К. Дж. Введение в системы баз данных [Текст] : [перевод с английского] / К. Дж. Дейт. - 8-е изд. – М. ; СПб. : Диалектика, 2019. – 1327 с. – ISBN 978-5-907144-17-0.

2. Гарсиа-Молина, Гектор Системы баз данных : Полный курс / Гектор Гарсиа-Молина, Джеффри Д. Ульман, Дженнифер Уидом ; [Пер. с англ. и ред. А.С. Варакина]. – М. : Вильямс, 2003 (ГПП Печ. Двор). – 1083 с. – ISBN 5-8459-0384-X (в пер.)

3. Карпова И. П. Базы данных [Текст] : курс лекций и материалы для практических занятий : учебное пособие для студентов технических факультетов, изучающих автоматизированные информационные системы и системы управления базами данных / И. П. Карпова. – М. : Питер, 2013. – 240 с. – (Учебное пособие). ; ISBN 978-5-496-00546-3

4. Кузин А. В. Базы данных: учебное пособие для вузов по направлению "Информатика и вычислительная техника" / А. В. Кузин, С. В. Левонисова. – 4-е изд., стер. – М. : Академия, 2010. – 315 с.

5. Хомоненко, А. Д. Базы данных: учебник для высших учебных заведений / А. Д. Хомоненко, В. М. Цыганков, М. Г. Мальцев; под ред. А. Д. Хомоненко. – 6-е изд. доп. – СПб. : КОРОНА - Век, 2009. – 736 с.

6. Грофф, Джеймс Р. Энциклопедия SQL: наиболее полное и подробное руководство: пер. с англ. / Джеймс Р. Грофф, Пол Н. Вайнберг. – 3-е изд. – СПб. ; Киев : Питер : Изд. группа BHV. – с. 880-895. – ISBN 966-552-103-9 (BHV). – ISBN 5-88782-077-2 (Питер).

7. Справочник по Transact-SQL [Электронный ресурс] – Режим доступа:https://msdn.microsoft.com/ru/library/bb510741(v=sql.120).aspx (дата обращения: 31.05.2020).

8. ГОСТ 20886-85. Организация данных в системах обработки данных. Термины и определения. – М. : Стандартинформ, 2005.

# ПРИЛОЖЕНИЯ

## Приложение 1

## Типы данных в СУБД MS SQL Server

Типы данных в MS SQL Server делятся на следующие категории:

- Точные числа (таблица П1.1).
- Приблизительные числа (таблица П1.2).
- Символьные строки (таблица П1.3).
- Символьные строки в Unicode (таблица П1.4).
- Дата и время (таблица П1.5).
- Двоичные данные (таблица П1.6).
- Прочие типы данных (таблица П1.7).

Таблица П1.1 Точные числа

| **Наименование типа** | **Хранилище** | **Описание** |
|---|---|---|
| bit | 1 байт | Может принимать значения 0, 1 или NULL. Часто используется как тип данных Boolean. Строковые значения TRUE и FALSE можно преобразовать в значения данного типа: TRUE преобразуется в 1, а FALSE в 0. Тип данных bit имеет размер в один байт, но при наличии нескольких полей типа bit в таблице они все будут упакованы вместе. |
| tinyint | 1 байт | Целые числа от 0 до 255 |
| smallint | 2 байта | от $-2^{15}$ (-32 768) до $2^{15}$-1 (32 767). |
| int | 4 байта | от $–2^{31}$ (–2 147 483 648) до $2^{31}$-1 (2 147 483 647). |
| bigint | 8 байт | от $-2^{63}$ (-9 223 372 036 854 775 808) до $2^{63}$-1 (9 223 372 036 854 775 807). |

| Наименование типа | Хранилище | Описание |
|---|---|---|
| numeric (p, s) и decimal (p, s) | Точность: от 1 до 9 = 5 байт; от 10 до 19 = 9 байт; от 20 до 28 = 13 байт; от 29 до 38 = 17 байт. | Тип числовых данных с фиксированной точностью и масштабом. numeric и decimal функционально эквивалентны. p (точность) — максимальное количество десятичных разрядов числа, которые будут храниться (как слева, так и справа от десятичной запятой). Точность может быть значением в диапазоне от 1 до 38, по умолчанию 18. s (масштаб) — максимальное количество десятичных разрядов числа справа от десятичной запятой. Максимальное число цифр слева от десятичной запятой определяется как p — s (точность — масштаб). Масштаб может быть значение от 0 до p, по умолчанию 0. Максимальный размер хранилища зависит от точности. Тип данных numeric и decimal может принимать значение от $-10^{38}+1$ до $10^{38}-1$. |
| smallmoney | 4 байта | Тип данных для хранения денежных значений с точность до одной десятитысячной денежной единицы. Число от -214 748,3648 до 214 748,3647 |
| money | 8 байт | Тип данных для хранения денежных значений с точность до одной десятитысячной денежной единицы. Число от -922 337 203 685 477,5808 до 922 337 203 685 477,5807 |

Таблица П1.2. Приблизительные числа

| Наименование типа | Хранилище | Описание |
|---|---|---|
| float (n) | Зависит от значения n: От 1 до 24 (7 знаков) = 4 байта; От 25 до 53 (15 знаков) = 8 байт. | Используется для числовых данных с плавающей запятой. n – это количество битов, используемых для хранения мантиссы числа в формате float при экспоненциальном представлении. n определяет точность данных и размер для хранения. Может принимать значение от 1 до 53, по умолчанию 53. Диапазон значений от –1,79E +308 до 1,79E+308. |

| Наименование типа | Хранилище | Описание |
|---|---|---|
| real | 4 байта | Используется для числовых данных с плавающей запятой. real соответствует в ISO типу float(24). Диапазон значений от –3.40E+38 до 3.40E+38. |

Не рекомендуется использовать столбцы с типами float и real в предложении WHERE, так как данные типы не хранят точных значений. Также не рекомендуется использовать float и real в финансовых приложениях, в операциях, связанных с округлением. Для этого лучше использовать decimal, money или smallmoney.

Таблица П1.3. Символьные строки

| Наименование типа | Хранилище | Описание |
|---|---|---|
| char (n) | n байт | Строка с фиксированной длиной не в Unicode, где n длина строки (от 1 до 8000). По умолчанию n = 1, если значение n не указано при использовании функций CAST и CONVERT, длина по умолчанию равна 30. |
| varchar ( n \| max ) | Размер занимаемой памяти в байтах = количество введенных символов + 2 байта. Если указать MAX, то максимально возможный размер = $2^{31}$-1 байт (2 ГБ). | Строковые данные переменной длины не в Unicode, где n длина строки (от 1 до 8000). По умолчанию n = 1, если значение n не указано при использовании функций CAST и CONVERT, длина по умолчанию равна 30. |
| text | Размер занимаемой памяти в байтах = количество введенных символов. Максимальный размер $2^{31}$-1 (2 147 483 647 байт, 2 ГБ). | Строка переменной длины не в Unicode. Является устаревшим типом данных, рекомендуется использовать varchar(max). |

Таблица П1.4. Символьные строки в Unicode

| **Наименование типа** | **Хранилище** | **Описание** |
| --- | --- | --- |
| nchar (n) | n * 2 байт | Строка с фиксированной длиной в Unicode, где n длина строки (от 1 до 4000). По умолчанию n = 1, если значение n не указано при использовании в функции CAST, длина по умолчанию равна 30. |
| nvarchar ( n \| max ) | Размер занимаемой памяти в байтах = количество введенных символов, умноженное на 2 + 2 байта. Если указать MAX, то максимально возможный размер = $2^{31}$-1 байт (2 ГБ). | Строка переменной длины в Unicode, где n длина строки (от 1 до 4000). По умолчанию n = 1, если значение n не указано при использовании в функции CAST, длина по умолчанию равна 30. |
| ntext | Размер занимаемой памяти в байтах = количество введенных символов, умноженное на 2. Максимальный размер $2^{30}$ — 1 (1 073 741 823 байт, 1 ГБ). | Строка переменной длины в Unicode. Является устаревшим типом данных, рекомендуется использовать nvarchar(max). |

Таблица П1.5. Дата и время

| Наименование типа | Хранилище | Диапазон | Точность | Описание |
|---|---|---|---|---|
| date | 3 байта | От 01.01.0001 до 31.12.9999 | 1 день | Используется для хранения даты. |
| datetime | 8 байт | От 01.01.1753 00:00:00 до 31.12.9999 23:59:59,997 | 0,00333 секунды | Используется для хранения даты, включая время с точностью до одной трехсотой секунды. |
| datetime2 | От 6 до 8 байт (в зависимости от точности: менее 3 цифр = 6 байт, 3-4 цифры = 7 байт, более 4 цифр = 8 байт) | От 01.01.0001 00:00:00.0000000 до 31.12.9999 23:59:59.9999999 | 100 наносекунд | Расширенный вариант типа данных datetime, имеет более широкий диапазон дат и большую точность в долях секунды (до 7 цифр). |
| smalldatetime | 4 байта | От 01.01.1900 00:00:00 до 06.06.2079 23:59:00 | 1 минута | Сокращенный вариант типа данных datetime, имеет меньший диапазон дат и не имеет долей секунд. |
| time [Точность] | От 3 до 5 байт | От 00:00:00.0000000 до 23:59:59.9999999 | 100 наносекунд | Используется для хранения времени дня. Точность может быть целым числом от 0 до 7, по умолчанию 7 (100 наносекунд, 5 байт). Если указать 0, то точность будет до секунды (3 байта). |

| Наименование типа | Хранилище | Диапазон | Точность | Описание |
|---|---|---|---|---|
| datetimeoffset [Точность] | От 8 до 10 байт | От 01.01.0001 00:00:00.0000000 до 9999-12-31 23:59:59.9999999 | 100 наносекунд | Используется для хранения даты и времени, включая смещение часовой зоны относительно универсального глобального времени. Точность определяет количество знаков в дробной части секунды, данное значение может быть от 0 до 7, по умолчанию 7 (100 наносекунд, 10 байт). |

Таблица П1.6. Двоичные данные

| Наименование типа | Хранилище | Описание |
|---|---|---|
| binary (n) | n байт | Двоичные данные фиксированной длины. n — значение от 1 до 8000. Если не указывать n, то значение по умолчанию 1, если не указать в функции CAST, то 30. Данный тип лучше использовать в случаях, когда размер данных, которые будут храниться в столбце, можно заранее определить. |
| varbinary (n \| max) | Размер занимаемой памяти в байтах = фактический размер данных + 2 байта. Если указать MAX, то максимально возможный размер = $2^{31}$-1 байт (2 ГБ). | Двоичные данные с переменной длиной. n — значение от 1 до 8000. Если не указывать n, то значение по умолчанию 1, если не указать в функции CAST, то 30. Данным типом лучше пользоваться, если размер данных в столбце заранее определить трудно. Если размер данных превышает 8000 байт, необходимо использовать тип varbinary(max). |

| **Наименование типа** | **Хранилище** | **Описание** |
|---|---|---|
| image | Максимальный размер до $2^{31}$-1 (2 147 483 647 байт, 2 ГБ). | Двоичные данные с переменной длиной. Является устаревшим типом данных, рекомендуется использовать varbinary(max). |

Таблица П1.7. Прочие типы данных

| **Наименование типа** | **Хранилище** | **Описание** |
|---|---|---|
| cursor | | Данный тип данных можно использовать в переменных или выходных параметрах хранимых процедур, которые содержат ссылку на курсор. Тип cursor не может быть использован в инструкции CREATE TABLE, т.е. для столбца в таблице. Может принимать значение NULL. |
| table | | Особый тип данных для переменных, который предназначен для хранения результирующего набора данных. Переменные с данным типом называют – табличные переменные. |
| sql_variant | | Универсальный тип данных, который может хранить значения различных типов данных. Однако sql_variant может хранить значения не всех типов, которые есть в SQL сервере, например следующие типы нельзя сохранить при помощи типа данных sql_variant: varchar(max), varbinary(max), nvarchar(max), xml, text, ntext, image, rowversion, hierarchyid, datetimeoffset, а также пространственные типы данных и определяемые пользователем типы. Тип sql_variant не может также иметь sql_variant в качестве базового типа. |

| Наименование типа | Хранилище | Описание |
|---|---|---|
| rowversion (timestamp) | 8 байт | Тип данных rowversion представляет собой автоматически создаваемые уникальные двоичные числа. В таблице может быть определен только один столбец типа rowversion. После любого обновления строки или вставки новой строки в таблицу, которая содержит столбец типа rowversion, значение увеличенной rowversion вставляется в столбец с данным типом. Поэтому столбец с типом данных rowversion не рекомендуется использовать в ключе, особенно в первичном ключе. timestamp является синонимом типа данных rowversion, но данный синтаксис устарел и его использовать нежелательно. |
| xml | Не более 2 ГБ. | Используется для хранения XML-данных. |
| uniqueidentifier | 16 байт | Глобальный уникальный идентификатор (GUID). Инициализировать столбец или переменную с типом uniqueidentifier можно с помощью функции NEWID или путем преобразования строки xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx, где каждый x – это шестнадцатеричная цифра (0–9 или A–F). |
| hierarchyid | Максимум 892 байта | Тип данных используется для представления положения в древовидной иерархии. |
| Пространственные типы | | К пространственным типам относятся: geography – это географический пространственный тип данных, который используется для представления данных в системе координат круглой земли, geometry – это пространственный тип данных для представления данных в евклидовом пространстве (плоской системе координат). |

В Microsoft SQL Server в случаях, когда оператор объединяет два выражения с разными типами данных, происходит неявное преобразование типов, если такое преобразование не поддерживается, SQL сервер будет выдавать ошибку. Чтобы определять какой тип данных из выражений преобразовывать, SQL Server применяет правила приоритета типов данных.

Для выполнения явных преобразований SQL Server предлагает универсальные функции CONVERT и CAST, с помощью которых значения одного типа преобразовываются в значения другого типа (если такие изменения возможны). CONVERT и CAST примерно одинаковы и могут быть взаимозаменяемыми:

CONVERT (тип_данных[(длина)], выражение )

CAST (выражение AS тип_данных).

**Приложение 2**

## Функции Transact-SQL

*Агрегатные функции*

Агрегатные функции позволяют производить вычисления на основе данных, хранящихся в некотором столбце таблицы (таблица П2.1). Параметром агрегатной функции может быть имя столбца таблицы или звездочка (*).

Таблица П2.1. Агрегатные функции

| Функция | Действие |
|---|---|
| AVG(столбец) | Возвращает среднее арифметическое значение указанного столбца таблицы |
| COUNT(столбец/*) | Возвращает число строк таблицы, в которых значение заданного столбца не равно значению NULL. Если параметром функции является звездочка, то подсчет ведется по всем строкам таблицы. |
| MAX(столбец) | Возвращает максимальное значение в столбце |
| MIN(столбец) | Возвращает минимальное значение в столбце |
| SUM(столбец) | Возвращает сумму значений заданного столбца |

Следует отметить, что все эти функции обрабатывают только те строки, которые удовлетворяют заданному критерию отбора.

*Строковые функции*

Строковые функции используются с строковыми типами данных (таблица П2.2).

Таблица П2.2. Строковые функции

| **Функция** | **Результат** |
|---|---|
| ASCII(строка) | Возвращает ASCII-код первого символа строки. |
| CHAR(целое число) | Преобразует целое число в символ. |
| CHARINDEX(символьное выражение, строка) | Возвращает номер позиции, с которой начинается символьное выражение в строке. Если заданное символьное выражение в строке отсутствует, то возвращается 0. |
| DIFFERENCE(строка, строка) | Выясняет степень схожести строк, возвращая значение от 0 до 4. Число 4 означает полное совпадение строк. |
| LOWER(строка) | Переводит символы строки в нижний регистр. |
| LTRIM(строка) | Удаляет пробелы в начале строки. |
| RATINDEX(шаблон, строка) | Возвращает номер позиции первого вхождения шаблона в строку. Шаблон должен быть заключен в знаки % и в нем допустимы другие символы маски. |
| REPLICATE(строка, целое число) | Повторяет строку указанное число раз. |
| REVERSE(строка) | Возвращает строку «задом наперед». |
| RIGHT(строка, целое число) | Возвращает часть строки, начиная с указанной позиции. |
| RTRIM(строка) | Удаляет пробелы в конце строки. |
| SOUNDEX(строка, строка) | Возвращает код из четырех цифр, который в дальнейшем используется при сравнении двух строк с помощью функции DIFFERENCE. |
| SPACE(целое число) | Возвращает строку из указанного числа пробелов. Если указано отрицательное число, то возвращается пустая строка. |
| STR(число, число символов, количество десятичных знаков) | Преобразует число в строку символов. |
| STUFF(строка_1, начальная позиция, длина, строка_2) | Удаляет из первой строки подстроку, которая определяется начальной позицией и длиной, и вставляет на это место вторую строку. |
| SUBSTRING(строка, начальная позиция, длина) | Возвращает часть строки, указанной длины, от начальной позиции. |
| UPPER(строка) | Переводит символы из нижнего регистра в верхний. |

*Математические функции*

Математические функции работают с числовыми типами данных и приведены в таблице П2.3.

Таблица П2.3. Математические функции

| **Функция** | **Типы аргументов** | **Результат** |
|---|---|---|
| ABS | Число | Абсолютное значение. |
| ACOS, ASIN, ATAN | Число с плавающей запятой | Обратные косинус, синус и тангенс. Возвращает угол в радианах. |
| ATAN2 | Число_1, число_2 | Возвращает угол в радианах, обратный тангенс которого равен частному от деления числа_1 на число_2. |
| COS, SIN, COT, TAN | Число с плавающей запятой, задающее угол в радианах | Косинус, синус, котангенс и тангенс угла. |
| CEILING | Число | Наименьшее целое, которое больше или равно указанному аргументу. |
| DEGREES | Число | Преобразует угол из радиан в градусы. |
| EXP | Число с плавающей запятой | Экспонента от аргумента. |
| FLOOR | Число | Наибольшее целое, которое меньше или равно указанному аргументу. |
| LOG | Число с плавающей запятой | Натуральный логарифм аргумента. |
| LOG10 | Число с плавающей запятой | Десятичный логарифм аргумента. |
| PI | - | Возвращает константу 3.141592653 (число $\pi$) |
| POWER | Число, y (число) | Возвращает аргумент в степени y. |
| RADIANS | Число | Преобразует угол из градусов в радианы. |
| RAND | Целое число. Аргумент необязателен. | Возвращает случайное число с плавающей запятой в диапазоне от 0 до 1. Аргумент может использоваться в качестве начального значения. |
| ROUND | Число, количество цифр | Число округляется до указанного количества цифр после запятой. |
| SIGN | Число | Возвращает знак числа. |
| SQRT | Число с плавающей запятой | Квадратный корень от числа. |

*Функции для работы с датами*

Функции для работы с датами вычисляют соответствующие части даты или времени выражения или возвращают значение временного интервала. Функции даты и времени принимают в качестве входных значений дату и время и возвращают либо строковые, числовые значения, либо значения в формате даты и времени. Поддерживаемые в Transact-SQL функции даты и их краткое описание приводятся в таблице П2.4.

Таблица П2.4. Функции для работы с датами

| **Функция** | **Параметры** | **Операция** |
|---|---|---|
| DATEADD | единицы, число, дата | Рассчитывает новую дату, добавляя к существующей указанное число единиц (дней, месяцев, часов и т.д.). |
| DATEDIFF | единицы, нач_дата, кон_дата | Возвращает количество единиц времени, между двумя указанными датами. |
| DATENAME | единицы, дата | Возвращает имя указанной единицы времени даты в виде строки. |
| DATEPART | единицы, дата | Возвращает имя указанной единицы времени даты в виде числа. |
| DAY | дата | Возвращает день для указанной даты в виде числа. |
| GETDATE | | Возвращает текущее системное время и дату. |
| MONTH | дата | Возвращает месяц для указанной даты в виде числа. |
| YEAR | дата | Возвращает год для указанной даты в виде числа. |

Для определения понятия «часть даты» существует аббревиатура и диапазоны возможных значений, которые приведены в таблице П2.5.

Таблица П2.5. Определения понятия «часть даты»

| Часть даты | Сокращение | Диапазон значений |
|---|---|---|
| Год | Yy | 1753 – 9999 |
| Квартал | Qq | 1 – 4 |
| Месяц | Mm | 1 – 12 |
| День года | Dy | 1 - 366 |
| День | Dd | 1 – 31 |
| Неделя | Wk | 0 – 51 |
| День недели | Dw | 1 - 7 |
| Час | Hh | 0 – 23 |
| Минута | Mi | 0 – 59 |
| Секунда | Ss | 0 - 59 |
| Миллисекунда | Ms | 0 - 999 |

## Приложение 3
## Демонстрационная база данных

Демонстрационная база данных предназначена для контроля поступления материалов и товаров на склад промышленного предприятия и состоит из таблиц Приход, Склад, Документ, Поставщик.

Таблица Приход содержит данные о поступлении на предприятие всех материалов и товаров. Причем один и тот же товар может поступать несколько раз. В таблице Склад хранятся данные о материалах и товарах на складе. В таблице Документ представлены данные о документах, по которым получен материал или товар. Таблица Поставщик содержит сведения о поставщиках материалов и товаров.

Реляционная схема этой базы данных выглядит следующим образом:

**Приход** (Ном_док, Ном_ном, Тип, Цена, Количество)
**Склад** (Ном_ном, Наименование, Цена, Количество)
**Документ** (Ном_док, Код_постав, Дата)
**Поставщик**(Код_постав, Название, Регион, Рейтинг)

Структура и свойства таблиц Приход, Склад, Документ, Поставщик приведены в таблицах П3.1 – П3.4.

Таблица П3.1. Структура и свойства таблицы Приход

| **Наименование поля** | **Тип и размер поля** | **Ключ** | **Допустимость неопределенных значений** | **Ограничения** | **Примечание** |
|---|---|---|---|---|---|
| Ном_док | int | первичный | NOT NULL | Каждый номер документа имеет уникальный номер. | Номер документа |
| Ном_ном | int | первичный | NOT NULL | Каждый номер товара уникальный номер. | Номенклатурный номер товара |
| Тип | varchar(10) | | NOT NULL | | Тип товара |
| Цена | money | | NOT NULL | | Цена единица товара |
| Количество | int | | NOT NULL | | Количество поступившего товара |

Таблица П3.2. Структура и свойства таблицы Склад

| **Наименование поля** | **Тип и размер поля** | **Ключ** | **Допустимость неопределенных значений** | **Ограничения** | **Примечание** |
|---|---|---|---|---|---|
| Ном_ном | int | первичный | NOT NULL | Каждый номер товара имеет уникальный номер. | Номенклатурный номер товара |
| Наименование | varchar(20) | | NOT NULL | | Наименование товара |
| Цена | money | | NOT NULL | | Цена товара |
| Количество | int | | NOT NULL | | Количество товара на складе |

Таблица П3.3. Структура и свойства таблицы Документ

| **Наименование поля** | **Тип и размер поля** | **Ключ** | **Допустимость неопределенных значений** | **Ограничения** | **Примечание** |
|---|---|---|---|---|---|
| Ном_док | int | первичный | NOT NULL | Каждый номер документа имеет уникальный номер. | Номер документа |
| Код_постав | int | Внешний ключ | NOT NULL | | Код поставщика |
| Дата | Data | | NOT NULL | | Дата оформления документа |

Таблица П3.4. Структура и свойства таблицы Поставщик

| **Наименование поля** | **Тип и размер поля** | **Ключ** | **Допустимость неопределенных значений** | **Ограничения** | **Примечание** |
|---|---|---|---|---|---|
| Код_постав | int | Первичный ключ | NOT NULL | Каждый поставщик имеет уникальный код. | Код поставщика |
| Название | varchar(20) | | NOT NULL | | Название поставщика |
| Регион | varchar(15) | | NULL | | Регион поставщика |
| Рейтинг | int | | NOT NULL | Значение по умолчанию: 100 | Рейтинг поставщика |

## ОГЛАВЛЕНИЕ

**ПРЕДИСЛОВИЕ** .......... 3

**Глава 1. ВВЕДЕНИЕ В БАЗЫ ДАННЫХ** .......... 5
1.1. Принципы обработки данных .......... 5
1.2. Основные понятия баз данных .......... 8
1.3. Информационные системы, использующие концепцию баз данных .......... 10
1.4. Функции СУБД .......... 13
1.5. Виды архитектуры информационной системы .......... 16
1.5.1. Централизованная архитектура .......... 16
1.5.2. Файл-серверная архитектура .......... 17
1.5.3. Клиент-серверная архитектура .......... 18
1.5.4. Трехуровневая архитектура .......... 20
1.6. Организация БД. Трехуровневое представление данных .......... 21
1.7. Понятие модели данных и виды моделей .......... 25
Вопросы для самопроверки .......... 27

**Глава 2. ДАТАЛОГИЧЕСКИЕ МОДЕЛИ ДАННЫХ** .......... 28
2.1. Компоненты даталогических моделей данных .......... 28
2.2. Иерархические модели данных .......... 29
2.3. Сетевые модели данных .......... 33
2.4. Реляционная модель данных .......... 36
2.4.1. Формальное описание реляционной модели данных .......... 36
2.4.2. Ключи отношения .......... 39
2.4.3. Типы связи таблиц .......... 41
2.4.4. Манипулирование данными в реляционной модели .......... 44
2.4.4.1. Реляционная алгебра .......... 45
2.4.4.2. Свойства операций реляционной алгебры .......... 51
2.4.4.3. Реляционное исчисление .......... 52
2.4.5. Достоинства и недостатки реляционных моделей данных .......... 55
2.4.6. Ограничения целостности базы данных .......... 56
Вопросы для самопроверки .......... 57

**Глава 3. ЯЗЫК SQL** .......... 58
3.1. Общие сведения о языке SQL .......... .58
3.1.1. История возникновения и стандарты языка SQL .......... 58
3.1.2. Элементы языкаTransact-SQL .......... 60
3.1.2.1. Алфавит языкаTransact-SQL .......... 61
3.1.2.2. Идентификаторы .......... 61
3.1.2.3. Комментарии .......... 62
3.1.2.4. Операторы .......... 62
3.1.2.5. Типы данных .......... 65
3.1.2.6. Функции .......... 65
3.1.3. Структура языка SQL .......... 66
Вопросы для самопроверки .......... 68
3.2. Извлечение данных из таблиц .......... 69
3.2.1. Команда SELECT .......... 69
3.2.1.1. Базовый синтаксис команды SELECT .......... 69
3.2.1.2. Список выбираемых столбцов .......... 72
3.2.1.3. Использование таблиц, входящих в базу данных .......... 80
3.2.1.4. Выборка строк .......... 80
3.2.1.5. Группировка данных .......... 88
3.2.1.6. Порядок вывода данных .......... 93
Вопросы для самопроверки .......... 96
Практические задания .......... 97
3.2.2. Выборка данных из нескольких таблиц .......... 97
3.2.2.1. Внутреннее соединение .......... 98
3.2.2.2. Внешнее соединение .......... 102
3.2.2.3. Объединение выборок .......... 104
Вопросы для самопроверки .......... 106
Практические задания .......... 107
3.2.3. Подзапросы .......... 108
3.2.3.1. Скалярные подзапросы .......... 110
3.2.3.2. Векторные подзапросы .......... 112
3.2.3.2.1. Подзапросы, начинающиеся с оператора IN .......... 112
3.2.3.2.2. Подзапросы, включающие ключевые слова ANY и ALL .......... 116

3.2.3.2.3. Проверка на существование......................124
3.2.3.3. Запросы в предложении FROM...................... 128
3.2.3.4. Правила формирования подзапросов ............ 130
Вопросы для самопроверки ............................................ 130
Практические задания ................................................... 131
3.2.4. Оконные функции T-SQL .......................................... 132
3.2.4.1. Описание оконных функций ......................... 132
3.2.4.2. Агрегатные оконные функции ...................... 134
3.2.4.3. Оконные функции ранжирования ................ 146
3.2.4.4. Оконные функции смещения......................... 152
3.2.4.5. Аналитические оконные функции ................ 156
Вопросы для самопроверки ............................................ 160
Практические задания ................................................... 160
3.3. Команды модификации данных .......................................... 161
3.3.1. Добавление записей в таблицу ................................. 161
3.3.2. Удаление записей из таблицы ................................. 165
3.3.3. Изменение значений полей таблицы ....................... 166
3.3.4. Использование подзапросов с командами модификации ........................................................... 168
Вопросы для самопроверки ..................................................... 172
Практические задания ............................................................. 174
3.4. Представления ................................................................. 175
3.4.1. Введение в представления ...................................... 175
3.4.2. Создание представлений ........................................... 176
3.4.3. Изменение значений базовой таблицы с помощью представлений ..................................... 181
3.4.4. Удаление представлений .......................................... 186
Вопросы для самопроверки ..................................................... 186
Практические задания ............................................................. 187
**ЗАКЛЮЧЕНИЕ** ............................................................................. 188
**БИБЛИОГРАФИЧЕСКИЙ СПИСОК** ........................................... 189
**ПРИЛОЖЕНИЯ**............................................................................. 190

*Учебное издание*

ГРАДУСОВ Александр Борисович

БАЗЫ ДАННЫХ
Введение в технологию баз данных

Учебно-практическое пособие

*Издается в авторской редакции*

Подписано в печать 03.03.21.
Формат 60х84/16. Усл. печ. л. 12,09. Тираж 50 экз.
Заказ
Издательство
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых.
600000, Владимир, ул. Горького, 87.