ИНФОРМАЦИОННЫЕ и КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ

СУБД: ЯЗЫК SQL В ПРИМЕРАХ И ЗАДАЧАХ

И.Ф. Астахова
В.М. Мельников
А.П. Толстобров
В.В. Фертиков

ИНФОРМАЦИОННЫЕ и КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ

СУБД: ЯЗЫК SQL в примерах и задачах

Допущено Министерством образования и науки Российской Федерации в качестве учебного пособия для студентов высших учебных заведений, обучающихся по направлению подготовки и специальности «Прикладная математика и информатика»

МОСКВА
ФИЗМАТЛИТ®
2007

УДК 681.066
ББК 22.18
С89

Астахова И. Ф., Мельников В. М., Толстобров А. П., Фертиков В. В. **СУБД: язык SQL в примерах и задачах.** — М.: ФИЗМАТЛИТ, 2009. — 168 с. — ISBN 978-5-9221-0816-4.

Учебное пособие содержит подборку примеров и упражнений различной степени сложности для практических занятий по изучению основ языка SQL в рамках учебного курса, посвященного информационным системам с базами данных.

Допущено Министерством образования и науки Российской Федерации в качестве учебного пособия для студентов высших учебных заведений, обучающихся по направлению подготовки и по специальности «Прикладная математика и информатика».

ISBN 978-5-9221-0816-4

ОГЛАВЛЕНИЕ

Введение

Информационные системы, использующие базы данных, в настоящее время представляют собой одну из важнейших областей современных компьютерных технологий. С этой сферой связана большая часть современного рынка программных продуктов. Одной из общих тенденций в развитии таких систем являются процессы интеграции и стандартизации, затрагивающие структуры данных и способы их обработки и интерпретации, системное и прикладное программное обеспечение, средства разработки взаимодействия компонентов баз данных и т. п. Современные системы управления базами данных (СУБД) основаны на реляционной модели представления данных — в большой степени благодаря простоте и четкости ее концептуальных понятий и строгому математическому обоснованию.

Неотъемлемая и важная часть любой системы, включающей базы данных, — языковые средства, предоставляющие возможность доступа к данным для получения необходимой информации и осуществления необходимых действий над содержимым данных, определения их структур, способов использования и интерпретации. Язык SQL появился в 70-е годы XX века как одно из таких средств. Его прототип был разработан фирмой IBM и известен под названием SEQUEL (Structured English QUEry Language). SQL вобрал в себя достоинства реляционной модели, в частности, достоинства лежащего в ее основе математического аппарата реляционной алгебры и реляционного исчисления, используя при этом сравнительно небольшое число операторов и относительно простой синтаксис. Благодаря своим качествам язык SQL стал — вначале де-факто, а затем и официально — утвержденным в качестве стандарта языком работы с реляционными базами данных.

Учитывая место, занимаемое языком SQL в современных информационных технологиях, его знание необходимо любому специалисту, работающему в этой области. Поэтому его практическое освоение является неотъемлемой частью учебных курсов, направленных на изучение информационных систем с базами данных. В настоящее время такие курсы входят в учебные планы ряда университетских специальностей. Несомненно, что для получения студентами устойчивых навыков владения языком SQL, соответствующий учебный курс, помимо теоретиче-

ского ознакомления с основами языка, должен обязательно содержать достаточно большой объем лабораторных занятий по его практическому использованию. Предлагаемое учебное пособие направлено в первую очередь на методическое обеспечение именно такого рода занятий. В связи с этим в нем основное внимание уделяется подбору практических примеров, задач и упражнений различной степени сложности по составлению SQL-запросов, позволяющих обеспечить проведение практических занятий по изучению языка в течение учебного семестра. При этом описание конструкций языка и тонкостей его применения в пособии приводится в минимальном объеме, необходимом для понимания студентами предлагаемых для решения упражнений и задач, и, возможно, с некоторым ущербом в строгости изложения материала.

Все приведенные в пособии задачи и упражнения составлены на примере использования одной общей базы данных, структура которой специально подобрана для обеспечения практической реализации и иллюстрации изучаемых конструкций языка. Для облегчения практической организации занятий по изучению языка с использованием предлагаемых в пособии упражнений в компьютерном классе на реальной базе данных дополнением пособия служит файл, содержащий SQL-сценарий создания и наполнения данными учебной базы данных, использованной в книге. Файл размещен на сайте издательства <http://www.fml.ru>. Первая часть сценария состоит из последовательности операторов DDL для создания таблиц, первичных и внешних ключей. Остальная часть образована командами DML, наполняющими таблицы учебной информацией. Авторы надеются, что возможные случайные совпадения с реальными данными не вызовут раздражения читателей.

Сценарий ориентирован на сервер Oracle, и для его использования с другой СУБД, по-видимому, потребуется учет специфики. Следует обратить внимание на типы полей таблиц, заданные DDL-операторами сценария, а также на использованный формат представления строк и дат в командах DML. В самом трудном случае адаптация сценария, возможно, потребует использования какого-либо текстового процессора (например, для изменения длины строк в запросах INSERT). Кроме данной преодолимой трудности, авторы не предвидят препятствий к использованию сценария: достаточно квалификации пользователя СУБД, прочитавшего нашу книгу.

Возможность составления студентами запросов к реальной базе данных с достаточно большим объемом специально подобранных данных позволяет существенно повысить продуктивность занятий, более наглядно увидеть особенности выполнения конкретных видов SQL-запросов, в частности, оценить реальное время их выполнения.

В пособии приведены ответы на большинство приведенных в нем задач, облегчающие преподавателю проверку результатов выполнения заданий и позволяющие использовать пособие для самостоятельной работы студентов. Примеры и задачи протестированы с использованием

СУБД Oracle и практически опробованы при проведении занятий в Воронежском госуниверситете на факультете компьютерных наук и факультете прикладной механики, математики и информатики.

В приложении приведены тексты дополнительных задач по проектированию баз данных. Эти задачи также могут использоваться при выполнении курсовых работ и самостоятельной работы студентов.

Авторы надеются, что пособие окажется полезным не только преподавателям и студентам, но и другим читателям, заинтересованным в получении начальных практических навыков использования языка SQL.

Авторы выражают искреннюю благодарность всем, кто помогал в создании этой книги, преподавателям ВГУ, использующим ее материалы при проведении занятий по языку SQL, за обсуждение книги и пожелания по ее содержанию. Особая благодарность Сергею Дмитриевичу Кузнецову, профессору кафедры системного программирования факультета вычислительной математики и кибернетики МГУ, который не пожалел времени на внимательное прочтение рукописи и сделал большое число ценных замечаний, позволивших значительно улучшить эту книгу.

Авторы с благодарностью примут любые замечания, пожелания, исправления, которые будут способствовать улучшению качества пособия, по адресу: 394693, Университетская пл., 1, Воронеж, Россия; электронный адрес: tap@main.vsu.ru.

Глава 1

ОСНОВНЫЕ ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ

1.1. Основные понятия реляционных баз данных

Основой современных систем, использующих базы данных, является *реляционная* модель данных. В этой модели данные, представляющие информацию о предметной области, организованы в виде двумерных таблиц, называемых *отношениями*. На рис. 1 приведен пример такой таблицы-отношения и поясняются основные термины реляционной модели.

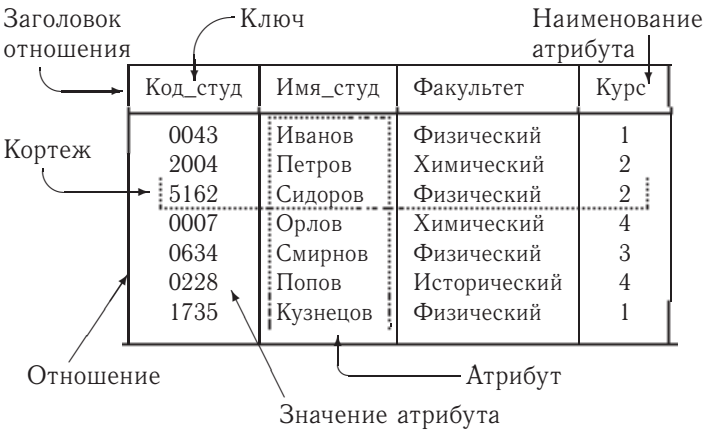

Рис. 1

- *Отношение* — это таблица, подобная приведенной на рис. 1, и состоящая из строк и столбцов. Верхняя строка таблицы-отношения называется *заголовком отношения*. Термины *отношение* и *таблица* обычно употребляются как синонимы, однако в языке SQL используется термин *таблица*.

- Строки таблицы-отношения называются *кортежами*, или *записями*. Столбцы называются *атрибутами*. Термины: атрибут, столбец, колонка, поле — обычно используются как синонимы. Каждый атрибут имеет наименование (имя), которое должно быть уникальным в конкретной таблице-отношении, однако в разных таблицах имена атрибутов могут совпадать.
- Количество кортежей в таблице-отношении называется *кардинальным числом* отношения, а количество атрибутов называется *степенью* отношения.
- *Ключ*, или *первичный ключ* отношения — это уникальный идентификатор строк (кортежей), т. е. такой атрибут (набор атрибутов), для которого в любой момент времени в отношении не существует строк с одинаковыми значениями этого атрибута (набора атрибутов). На рис. 1 таблицы ячейка с именем ключевого атрибута имеет нижнюю границу в виде двойной черты.
- *Домен* отношения — это совокупность значений, из которых могут выбираться значения конкретного атрибута, т. е. конкретный набор имеющихся в таблице значений атрибута в любой момент времени должен быть подмножеством множества значений домена, на котором определен этот атрибут. В общем случае на одном и том же домене могут быть определены значения разных атрибутов. Важным является то, что домены вводят ограничения на операции сравнения значений различных атрибутов. Эти ограничения состоят в том, что корректным образом можно сравнивать между собой только значения атрибутов, определенных на одном и том же домене.

Отношения реляционной базы данных обладают следующими свойствами:

- в отношениях не должно быть кортежей-дубликатов;
- кортежи отношений неупорядочены;
- атрибуты отношений также неупорядочены.

Из этих свойств отношения вытекают следующие важные следствия.

- Из уникальности кортежей следует, что в отношении *всегда* имеется атрибут или набор атрибутов, позволяющий *идентифицировать* кортеж; другими словами, в отношении ***всегда*** есть первичный ключ.
- Из неупорядоченности кортежей следует, во-первых, что в отношении не существует другого способа адресации кортежей, кроме адресации *по ключу*; во-вторых, что в отношении не существует таких понятий как первый кортеж, последний, предыдущий, следующий и т. п.
- Из неупорядоченности атрибутов следует, что единственным способом их адресации в запросах является использование наименования атрибута.

Относительно свойства реляционного отношения, касающегося отсутствия кортежей-дубликатов, следует сделать важное замечание. В этом пункте SQL не полностью соответствует реляционной модели. А именно, в отношениях, являющихся результатами запросов, SQL *допускает* наличие одинаковых строк. Для их устранения в запросе используется ключевое слово **DISTINCT** (см. ниже).

Информация в реляционных базах данных, как правило, хранится не в одной таблице-отношении, а в нескольких. При создании нескольких таблиц взаимосвязанной информации появляется возможность выполнения более сложных операций с данными, т. е. более сложной обработки данных. Для работы со связанными данными из нескольких таблиц важным является понятие так называемых *внешних ключей*.

Внешним ключом таблицы называется атрибут или набор атрибутов этой таблицы, каждое значение которых в текущем состоянии таблицы всегда совпадает со значением атрибутов, являющихся ключом, в другой таблице. Внешние ключи используются для связывания значений атрибутов из разных таблиц. С помощью внешних ключей обеспечивается так называемая *ссылочная целостность* базы данных, т. е. согласованность данных, описывающих одни и те же объекты, но хранящихся в разных таблицах.

1.2. Отличие SQL от процедурных языков программирования

Язык SQL относится к классу непроцедурных языков программирования. В отличие от универсальных процедурных языков, которые также могут быть использованы для работы с базами данных, язык SQL ориентирован не на *записи*, а на *множества*.

Это означает следующее. В качестве входной информации для формулируемого на языке SQL запроса к базе данных используется *множество кортежей-записей* одной или нескольких таблиц-отношений. В результате выполнения запроса также образуется *множество кортежей* результирующей таблицы-отношения. Другими словами, в SQL результатом любой операции над отношениями также является отношение. Запрос SQL задает не процедуру, т. е. последовательность действий, необходимых для получения результата, а условия, которым должны удовлетворять кортежи результирующего отношения, сформулированные в терминах входного отношения (входных отношений).

1.3. Интерактивный и встроенный SQL

Существуют и используются две формы языка SQL: *интерактивный SQL* и *встроенный SQL*.

Интерактивный SQL используется для непосредственного ввода SQL-запросов пользователем и получения результата в интерактивном режиме.

Встроенный SQL состоит из команд SQL, встроенных внутрь программ, которые обычно написаны на некотором другом языке (Паскаль, C, C++ и др.). Это делает программы, написанные на таких языках, более мощными, гибкими и эффективными, обеспечивая их применение для работы с данными, хранящимися в реляционных базах. При этом, однако, требуются дополнительные средства обеспечения интерфейса SQL с языком, в который он встраивается.

Данная книга посвящена интерактивному SQL, поэтому в ней не обсуждаются вопросы построения интерфейсов, позволяющих связать SQL с другими языками программирования.

1.4. Составные части SQL

И интерактивный, и встроенный SQL подразделяются на следующие составные части.

Язык Определения Данных — DDL (Data Definition Language): дает возможность создания, изменения и удаления различных объектов базы данных (таблиц, индексов, пользователей, привилегий и т. п.).

К числу дополнительных функций DDL могут быть отнесены средства определения ограничений целостности данных, определения порядка структур хранения данных, описания элементов физического уровня хранения данных.

Язык Обработки Данных — DML (Data Manipulation Language): предоставляет возможность выборки информации из базы данных и ее преобразования.

Тем не менее это не два различных языка, а компоненты единого SQL.

1.5. Типы данных

В языке SQL имеются средства, позволяющие для каждого атрибута указывать тип данных, которому должны соответствовать все значения этого атрибута.

Следует отметить, что определение типов данных является той частью, в которой коммерческие реализации языка не полностью согласуются с требованиями официального стандарта SQL. Это объясняется, в частности, желанием сделать SQL совместимым с другими языками программирования.

1.5.1. Тип данных “строка символов”. Тип данных **CHARACTER** или **CHAR** представляет символьные строки фиксированной длины. Его синтаксис имеет вид:

CHARACTER[(<*длина*>)] или
CHAR[(<*длина*>)].

Текстовые значения поля таблицы, для которого определен тип **CHAR**, имеют *фиксированную* длину, которая определяется параметром

<*длина*>. Этот параметр может принимать значения от 1 до 255, т. е. строка может содержать до 255 символов. Если во вводимой в поле текстовой константе фактическое число символов меньше числа, определенного параметром <*длина*>, то эта константа автоматически дополняется справа пробелами до заданного числа символов. Квадратные скобки указывают на то, что значение параметра <*длина*> может не указываться явно. В этом случае длина строки полагается равной одному символу.

Тип данных для строк переменной длины может обозначаться ключевыми словами **VARCHAR**, **CHARACTER VARYING** или **CHAR VARYING**. Он описывает текстовую строку, которая может иметь *произвольную* длину до определенного конкретной реализацией SQL максимума (в Oracle до 2000 символов). В отличие от типа **CHAR**, в этом случае при вводе текстовой константы, фактическая длина которой меньше заданной, ее дополнение пробелами до заданного максимального значения не производится.

Константы, имеющие тип **CHARACTER** или **VARCHAR**, в выражениях SQL заключаются в одиночные кавычки, например, '<*текст*>'.

Следующие предложения эквивалентны:

VARCHAR[(<*длина*>)], **CHAR VARYING**[(<*длина*>)], **CHARACTER VARYING**[(<*длина*>)].

Если длина строки не указана явно, она полагается равной одному символу во всех случаях.

По сравнению с типом **CHAR** тип данных **VARCHAR** позволяет более экономно использовать память, выделяемую для хранения текстовых значений, и оказывается более удобным при выполнении операций, связанных со сравнением текстовых констант.

1.5.2. Числовые типы данных. Стандартными числовыми типами данных SQL являются:

- **INTEGER** — используется для представления целых чисел в диапазоне от -2^{31} до $+2^{31}$;
- **SMALLINT** — используется для представления целых чисел в диапазоне меньшем, чем для **INTEGER**, а именно от -2^{15} до $+2^{15}$;
- **DECIMAL**(<*точность*>[, <*масштаб*>]) — десятичное число с фиксированной точкой; точность указывает, сколько значащих цифр имеет число. Масштаб указывает максимальное число цифр справа от точки;
- **NUMERIC**(<*точность*>[, <*масштаб*>]) — десятичное число с фиксированной точкой, такое же, как и **DECIMAL**;
- **FLOAT**[(<*точность*>)] — число с плавающей точкой и указанной минимальной точностью;
- **REAL** — такое же число, как и **FLOAT**, за исключением того, что точность устанавливается по умолчанию в зависимости от конкретной реализации SQL.

- **DOUBLE PRECISION** — такое же число, как и **REAL**, но точность в два раза превышает точность для **REAL**.

СУБД Oracle использует дополнительно тип данных **NUMBER** для представления всех числовых данных: целых, с фиксированной или плавающей точкой. Его синтаксис:

NUMBER[(<*точность*>[, <*масштаб*>])] .

Если значение параметра <*точность*> не указано явно, оно полагается равным 38. Значение параметра <*масштаб*> по умолчанию предполагается равным 0. Значение параметра <*точность*> может изменяться от 1 до 38; значение параметра <*масштаб*> может изменяться от −84 до 128. Использование отрицательных значений масштаба означает сдвиг десятичной точки в сторону старших разрядов. Например, определение **NUMBER**(7, −3) означает округление до тысяч.

Типы **DECIMAL** и **NUMERIC** полностью эквивалентны типу **NUMBER**. Синтаксис:

DECIMAL[(<*точность*>[, <*масштаб*>])],
DEC[(<*точность*>[, <*масштаб*>])],
NUMERIC[(<*точность*>[, <*масштаб*>])].

Напоминаем, что квадратные скобки указывают на необязательность заключенных в них параметров.

1.5.3. Дата и время. Представление дат и времени в SQL зависит от конкретной СУБД. В Oracle тип данных **DATE** используется для представления даты и времени. Наличие типа данных для хранения даты позволяет поддерживать специальную арифметику дат. Добавление к переменной типа **DATE** целого числа означает увеличение даты на соответствующее число дней, а вычитание соответствует определению более ранней даты.

Константы типа **DATE** записываются в зависимости от формата, принятого в конкретной системе. Например, `'03.05.1999'` или `'12/06/1989'`, или `'03-nov-1999'`, или `'03-apr-99'`.

1.5.4. Неопределенные или отсутствующие данные (NULL). Для обозначения отсутствующих, пропущенных или неизвестных значений атрибута в SQL используется ключевое слово **NULL**. Довольно часто можно встретить словосочетание "*атрибут имеет значение* **NULL**". Строго говоря, **NULL** не является значением в обычном понимании, а используется именно для обозначения того факта, что действительное значение атрибута на самом деле по каким-либо причинам отсутствует. Это приводит к ряду особенностей, что следует учитывать при использовании значений атрибутов, которые могут находиться в состоянии **NULL**.

- В агрегирующих функциях, позволяющих получать сводную информацию по множеству значений атрибута, например, суммарное или среднее значение, для обеспечения точности и однозначности

толкования результатов отсутствующие или **NULL**-значения атрибутов игнорируются.

- Условные операторы расширяются от булевой двузначной логики **истина/ложь** до трехзначной логики **истина/ложь/неизвестно**.
- Все операторы возвращают пустое значение (**NULL**), если значение любого из операндов отсутствует (имеет "значение **NULL**").
- Для проверки на пустое значение следует использовать операторы **IS NULL** и **IS NOT NULL** (использование для этого оператора сравнения "=" является ошибкой).
- Функции преобразования типов, имеющие **NULL** в качестве аргумента, возвращают пустое значение (**NULL**).

1.6. Используемые термины и обозначения

Ключевые слова — это используемые в выражениях SQL слова, имеющие специальное назначение (например, они могут обозначать конкретные команды SQL). Ключевые слова нельзя использовать для других целей, к примеру, в качестве имен объектов базы данных. В книге они выделяются шрифтом: **КЛЮЧЕВОЕСЛОВО**.

Команды, или *предложения*, являются инструкциями, с помощью которых SQL обращается к базе данных. Команды состоят из нескольких (одной или более) логических частей, называемых предложениями. Предложения начинаются ключевым словом и состоят из ключевых слов и аргументов.

Объекты базы данных, имеющие имена (таблицы, атрибуты и др.), в книге также выделяются особым образом: `ТАБЛИЦА1`, `АТРИБУТ_2`.

В описании синтаксиса команд SQL оператор определения "**::=**" разделяет определяемый элемент (слева от оператора) и собственно его определение (справа от оператора); квадратные скобки "[ ]" указывают *необязательный* элемент синтаксической конструкции; многоточие "..." указывает, что выражение, предшествующее ему, может повторяться любое число раз; фигурные скобки "{ }" объединяют последовательность элементов в *логическую группу*, один из элементов которой должно быть обязательно использован; вертикальная черта "|" указывает, что часть определения, следующая за этим символом, является одним из возможных вариантов; в угловые скобки "< >" заключаются элементы, которые объясняются по мере того, как вводятся.

1.7. Учебная база данных

В приводимых в пособии примерах построения SQL-запросов и контрольных упражнениях используется база данных, состоящая из следующих таблиц.

Таблица 1.1. STUDENT (Студент)

STUDENT_ID	SURNAME	NAME	STIPEND	KURS	CITY	BIRTHDAY	UNIV_ID
1	Иванов	Иван	150	1	Орел	3/12/1988	10
3	Петров	Петр	200	3	Курск	11/12/1986	10
6	Сидоров	Вадим	150	4	Москва	7/06/1985	22
10	Кузнецов	Борис	0	2	Брянск	8/12/1987	10
12	Зайцева	Ольга	250	2	Липецк	21/05/1987	10
265	Павлов	Андрей	0	3	Воронеж	5/11/1985	10
32	Котов	Павел	150	5	Белгород	**NULL**	14
654	Лукин	Артем	200	3	Воронеж	11/12/1987	10
276	Петров	Антон	200	4	**NULL**	5/08/1987	22
55	Белкин	Вадим	250	5	Воронеж	20/01/1986	10
…	…	…	…	…	…	…	…

STUDENT_ID — числовой код, идентифицирующий студента (идентификатор студента),
SURNAME — фамилия студента,
NAME — имя студента,
STIPEND — стипендия, которую получает студент,
KURS — курс, на котором учится студент,
CITY — город, в котором живет студент,
BIRTHDAY — дата рождения студента,
UNIV_ID — идентификатор университета, в котором учится студент.

Таблица 1.2. LECTURER (Преподаватель)

LECTURER_ID	SURNAME	NAME	CITY	UNIV_ID
24	Колесников	Борис	Воронеж	10
46	Никонов	Иван	Воронеж	10
74	Лагутин	Павел	Москва	22
108	Струков	Николай	Москва	22
276	Николаев	Виктор	Воронеж	10
328	Сорокин	Андрей	Орел	10
…	…	…	…	…

LECTURER_ID — идентификатор преподавателя,
SURNAME — фамилия преподавателя,
NAME — имя преподавателя,
CITY — город, в котором живет преподаватель,
UNIV_ID — идентификатор университета, в котором работает преподаватель.

Таблица 1.3. SUBJECT (Предмет обучения)

SUBJ_ID	SUBJ_NAME	HOUR	SEMESTER
10	Информатика	56	1
22	Физика	34	1
43	Математика	56	2
56	История	34	4
94	Английский	56	3
73	Физкультура	34	5
...	...	...	...

SUBJ_ID — идентификатор предмета обучения,

SUBJ_NAME — наименование предмета обучения,

HOUR — количество часов, отводимых на изучение предмета,

SEMESTER — семестр, в котором изучается данный предмет.

Таблица 1.4. UNIVERSITY (Университет)

UNIV_ID	UNIV_NAME	RATING	CITY
22	МГУ	610	Москва
10	ВГУ	296	Воронеж
11	НГУ	345	Новосибирск
32	РГУ	421	Ростов
14	БГУ	326	Белгород
15	ТГУ	373	Томск
18	ВГМА	327	Воронеж
...	...	...	...

UNIV_ID — идентификатор университета,

UNIV_NAME — название университета,

RATING — рейтинг университета,

CITY — город, в котором расположен университет.

Таблица 1.5. EXAM_MARKS (Экзаменационные оценки)

EXAM_ID	STUDENT_ID	SUBJ_ID	MARK	EXAM_DATE
145	12	10	5	12/01/2006
34	32	10	4	23/01/2006
75	55	10	5	05/01/2006
238	12	22	3	17/06/2005
639	55	22	**NULL**	22/06/2005
43	6	22	4	18/01/2006
…	…	…	…	…

EXAM_ID — идентификатор экзамена,
STUDENT_ID — идентификатор студента,
SUBJ_ID — идентификатор предмета обучения,
MARK — экзаменационная оценка,
EXAM_DATE — дата экзамена.

Таблица 1.6. SUBJ_LECT
(Учебные дисциплины преподавателей)

LECTURER_ID	SUBJ_ID
24	24
46	46
74	74
108	108
276	276
328	328
…	…

LECTURER_ID — идентификатор преподавателя,
SUBJ_ID — идентификатор предмета обучения.

ВОПРОСЫ

1. Какие поля приведенных таблиц являются первичными ключами?
2. Какие данные хранятся в столбце 2 в таблице “Предмет обучения”?
3. Как по-другому называется строка? столбец?
4. Почему мы не можем запросить для просмотра первые пять строк?

Глава 2

ВЫБОРКА ДАННЫХ (ОПЕРАТОР SELECT)

2.1. Простейшие SELECT-запросы

Оператор **SELECT** (ВЫБРАТЬ) языка SQL является самым важным и наиболее часто используемым оператором. Он предназначен для *выборки* информации из таблиц базы данных. Упрощенный синтаксис оператора **SELECT** выглядит следующим образом.

SELECT [DISTINCT]<*список выражений над атрибутами и константами*>
FROM <*список таблиц*>
[WHERE <*условие выборки*>**]**
[GROUP BY <*список атрибутов*>**]**
[HAVING <*условие*>**]**
[UNION <*выражение с оператором* **SELECT**>**]**
[ORDER BY <*список атрибутов*>**];**

В квадратных скобках указаны элементы, которые могут отсутствовать в запросе.

Ключевое слово **SELECT** сообщает базе данных, что данное предложение является запросом на *выборку* информации. После слова **SELECT** через запятую перечисляются *наименования полей* (список атрибутов), содержимое которых запрашивается.

Обязательным ключевым словом в предложении-запросе **SELECT** является слово **FROM** (ИЗ). За ключевым словом **FROM** указывается список разделенных запятыми имен таблиц, из которых извлекается информация.

Например,

```
SELECT NAME, SURNAME
    FROM STUDENT;
```

Любой SQL-запрос должен заканчиваться символом “**;**” (*точка с запятой*).

Приведенный запрос осуществляет выборку всех значений полей NAME и SURNAME из таблицы STUDENT.

Его результатом является таблица следующего вида:

NAME	SURNAME
Иван	Иванов
Петр	Петров
Вадим	Сидоров
Борис	Кузнецов
Ольга	Зайцева
Андрей	Павлов
Павел	Котов
Артем	Лукин
Антон	Петров
Вадим	Белкин
...	...

Порядок следования столбцов в этой таблице соответствует порядку полей NAME и SURNAME, указанному в запросе, а не их порядку во входной таблице STUDENT.

Если необходимо вывести значения *всех* столбцов таблицы, то можно вместо перечисления их имен использовать символ "*" (звездочка).

```
SELECT *
   FROM STUDENT;
```

В данном случае в результате выполнения запроса будет получена вся таблица STUDENT.

Еще раз обратим внимание на то, что получаемые в результате SQL-запроса таблицы не в полной мере отвечают определению реляционного отношения. В частности, в них могут оказаться кортежи с одинаковыми значениями атрибутов.

Например, запрос "Получить список названий городов, где проживают студенты, сведения о которых находятся в таблице STUDENT", можно записать в следующем виде:

```
SELECT CITY FROM STUDENT;
```

Его результатом будет таблица

CITY
Орел
Курск
Москва
Брянск
Липецк
Воронеж
Белгород
Воронеж
NULL
Воронеж
...

Видно, что в таблице встречаются одинаковые строки (выделены жирным шрифтом).

Для исключения из результата **SELECT**-запроса повторяющихся записей используется ключевое слово **DISTINCT** (ОТЛИЧНЫЙ). Если запрос **SELECT** извлекает множество полей, то **DISTINCT** *исключает* дубликаты строк, в которых значения *всех* выбранных полей идентичны.

Запрос "Определить список названий *различных* городов, где проживают студенты, сведения о которых находятся в таблице STUDENT", можно записать в следующем виде.

```
SELECT DISTINCT CITY
   FROM STUDENT;
```

В результате получим таблицу, в которой дубликаты строк исключены:

CITY
Орел
Курск
Москва
Брянск
Липецк
Воронеж
Белгород
NULL
...

Ключевое слово **ALL** (ВСЕ), в отличие от **DISTINCT**, оказывает противоположное действие, т. е. при его использовании повторяющиеся строки *включаются* в состав выходных данных. Режим, задаваемый ключевым словом **ALL**, действует по умолчанию, поэтому в реальных запросах для этих целей оно практически не используется.

Использование в операторе **SELECT** предложения, определяемого ключевым словом **WHERE** (ГДЕ), позволяет задавать выражение условия (предикат), принимающее значение *истина* или *ложь* (а также *неизвестно* при использовании **NULL**) для значений полей строк таблиц, к которым обращается оператор **SELECT**. Предложение **WHERE** определяет, ***какие строки*** *указанных таблиц должны быть выбраны*. В таблицу, являющуюся результатом запроса, включаются только те строки, для которых условие (предикат), указанное в предложении **WHERE**, принимает значение *истина*.

Пример.

Написать запрос, выполняющий выборку имен (NAME) всех студентов с фамилией (SURNAME) Петров, сведения о которых находятся в таблице STUDENT.

```
SELECT SURNAME, NAME
   FROM STUDENT
   WHERE SURNAME = 'Петров';
```

Результатом этого запроса будет таблица:

SURNAME	NAME
Петров	Петр
Петров	Антон

В задаваемых в предложении **WHERE** условиях могут использоваться операции сравнения, определяемые следующими операторами: = (равно), > (больше), < (меньше), >= (больше или равно), <= (меньше или равно), <> (не равно), а также логические операторы **AND**, **OR** и **NOT**.

Например, запрос для получения *имен* и *фамилий* студентов, обучающихся на *третьем* курсе и получающих стипендию (размер стипендии *больше нуля*) будет выглядеть таким образом:

```
SELECT NAME, SURNAME
   FROM STUDENT
   WHERE (KURS = 3 AND STIPEND > 0);
```

Результат выполнения этого запроса имеет вид:

SURNAME	NAME
Петров	Петр
Лукин	Артем

УПРАЖНЕНИЯ

1. Напишите запрос к таблице SUBJECT, выводящий для каждой ее строки идентификатор (номер) предмета обучения, его наименование, семестр, в котором он читается, и количество отводимых на него часов.
2. Напишите запрос, позволяющий вывести все строки таблицы EXAM_MARKS, в которых предмет обучения имеет номер (SUBJ_ID), равный 12.
3. Напишите запрос, выбирающий все данные из таблицы STUDENT, расположив столбцы таблицы в следующем порядке: KURS, SURNAME, NAME, STIPEND.
4. Напишите запрос **SELECT**, который для каждого предмета обучения (SUBJECT) выполняет вывод его наименования (SUBJ_NAME) и следом за ним количества часов (HOUR) в 4-м семестре (SEMESTR).
5. Напишите запрос, позволяющий получить из таблицы EXAM_MARKS значения столбца MARK (экзаменационная оценка) для всех студентов, исключив из списка повторение одинаковых строк.
6. Напишите запрос, который выполняет вывод списка фамилий студентов, обучающихся на третьем и более старших курсах.
7. Напишите запрос, выбирающий данные фамилию, имя и номер курса для студентов, получающих стипендию больше 140.
8. Напишите запрос, выполняющий выборку из таблицы SUBJECT названий всех предметов обучения, на которые отводится более 30 часов.
9. Напишите запрос, который выполняет вывод списка университетов, рейтинг которых превышает 300 баллов.
10. Напишите запрос к таблице STUDENT для вывода списка всех студентов со стипендией не меньше 100, живущих в Воронеже — с указанием фамилии (SURNAME), имени (NAME) и номера курса (KURS).
11. Какие данные будут получены в результате выполнения запроса?

```
SELECT *
   FROM STUDENT
   WHERE (STIPEND < 100 OR
      NOT (BIRTHDAY >= '10/03/1980'
      AND STUDENT_ID > 1003));
```

12. Какие данные будут получены в результате выполнения запроса?

```
SELECT *
   FROM STUDENT
   WHERE NOT ((BIRTHDAY = '10/03/1980' OR
                STIPEND > 100)
              AND STUDENT_ID >= 1003);
```

13. Напишите запрос для получения списка студентов старше 25 лет, обучающихся на 1-м курсе.
14. Напишите запрос для получения списка предметов, для которых в 1-м семестре отведено более 100 часов.
15. Напишите запрос для получения списка преподавателей, живущих в Воронеже.
16. Напишите запрос для получения списка университетов, расположенных в Москве и имеющих рейтинг меньший, чем у ВГУ. Константу в ограничении на рейтинг можно определить по этой же таблице.
17. Напишите запрос для получения списка студентов, проживающих в Воронеже и не получающих стипендию.
18. Напишите запрос для получения списка студентов моложе 20 лет.
19. Напишите запрос для получения списка студентов без определенного места жительства.

2.2. Операторы IN, BETWEEN, LIKE, IS NULL

При задании логического условия в предложении **WHERE** могут быть использованы операторы **IN**, **BETWEEN**, **LIKE**, **IS NULL**.

Операторы **IN** (РАВЕН ЛЮБОМУ ИЗ СПИСКА) и **NOT IN** (НЕ РАВЕН НИ ОДНОМУ ИЗ СПИСКА) используются для сравнения проверяемого значения поля с заданным списком. Этот список значений указывается в скобках справа от оператора **IN**. Список значений не обязательно задается в явном виде, он может представлять собой результат подзапроса.

Построенный с использованием **IN** предикат (условие) считается истинным, если значение поля, имя которого указано слева от **IN**, в точности *совпадает* с одним из значений, перечисленных в списке, указанном в скобках справа от **IN**.

Предикат, построенный с использованием **NOT IN**, считается истинным, если значение поля, имя которого указано слева от **NOT IN**, *не совпадает* ни с одним из значений, принадлежащих списку, указанному в скобках справа от **NOT IN**.

Пример 1.

Получить из таблицы EXAM_MARKS сведения о студентах, *имеющих* экзаменационные оценки только 4 и 5.

```
SELECT *
   FROM EXAM_MARKS
   WHERE MARK IN (4, 5);
```

Пример 2.

Получить сведения о студентах, *не имеющих* ни одной экзаменационной оценки, равной 4 или 5.

```
SELECT *
   FROM EXAM_MARKS
   WHERE MARK NOT IN (4, 5);
```

Оператор **BETWEEN** используется для проверки условия вхождения значения поля в заданный интервал, т. е. вместо списка значений атрибута этот оператор задает границы его изменения.

Например, запрос, выполняющий вывод записей о предметах обучения, для которых количество отводимых часов лежит в пределах между 30 и 40, имеет вид:

```
SELECT *
   FROM SUBJECT
   WHERE HOUR BETWEEN 30 AND 40;
```

Граничные значения, в данном случае значения 30 и 40, *входят* во множество значений, с которыми производится сравнение. Оператор **BETWEEN** может использоваться как для числовых, так и для символьных типов полей.

Оператор **LIKE** применим только к символьным полям типа **CHAR** или **VARCHAR**. Этот оператор осуществляет просмотр строковых значений полей с целью определения, входит ли заданная в операторе **LIKE** подстрока (образец поиска) в символьную строку, являющуюся значением проверяемого поля.

Для того чтобы осуществлять выборку строковых значений по заданному образцу подстроки, можно применять шаблон искомого образца строки, использующий следующие символы:

- символ подчеркивания "_", указанный в шаблоне образца, определяет возможность наличия в указанном месте *одного любого* символа,
- символ "%" допускает присутствие в указанном месте проверяемой строки последовательности любых символов произвольной длины.

Пример.

Написать запрос, выбирающий из таблицы **STUDENT** сведения о студентах, у которых фамилии начинаются на букву "P".

```
SELECT *
   FROM STUDENT
   WHERE SURNAME LIKE 'P%';
```

Так как символы "_" и "%" выполняют в языке SQL указанные выше специальные функции, возникает проблема, когда необходимо их указывать в текстовом образце в качестве обычных, а не служебных символов. В этих случаях применяют специальный механизм, позволяющий при интерпретации системой строки-образца отключить управляющие функции этих символов. Отключить служебные функции сиволов "_" и "%" можно путем вставки непосредственно перед ними так

называемого escape-символа (escape character). Этот символ, который можно еще назвать знаком перехода или знаком отключения, является служебным знаком, используемым для указания того, что должен быть изменен характер интерпретации следующего непосредственно за ним символа. В нашем случае если такой символ предшествует знаку "_" или "%", то этот знак будет интепретироваться уже буквально, как любой другой символ, а не как служебный символ. В SQL в качестве такого переключающего (escape) символа может быть назначен любой символ. для этих целей предназначено специальное ключевое слово **ESCAPE**.

Например, можно задать образец поиска с помощью следующего выражения:

```
LIKE '_\_P' ESCAPE '\'.
```

В этом выражении символ "\" с помощью ключевого слова **ESCAPE** объявляется escape-символом. Первый символ "_" в заданном шаблоне поиска "__P" будет соответствовать, как и ранее, любому набору символов в проверяемой строке. Однако второй символ "_", следующий после символа "\", объявленного escape-символом, уже будет интерпретироваться буквально как обычный символ, так же как и символ P в заданном шаблоне.

Обращаем ваше внимание на то, что в операторах сравнения =, <, >, <=, >=, <> и операторах **IN**, **BETWEEN** и **LIKE** при использовании **NULL** в качестве операнда будет возвращаться также **NULL**. В связи с этим, для проверки содержимого поля на наличие (отсутствие) в нем пустого значения **NULL** следует использовать специально предназначенные для этого операторы **IS NULL** (ЯВЛЯЕТСЯ ПУСТЫМ) и **IS NOT NULL** (НЕ ЯВЛЯЕТСЯ ПУСТЫМ), а не выражения = **NULL** или <> **NULL**.

УПРАЖНЕНИЯ

1. Напишите запрос, выполняющий вывод находящихся в таблице EXAM_MARKS номеров предметов обучения, экзамены по которым сдавались между 10 и 20 января 2005 г.
2. Напишите запрос, выбирающий данные обо всех предметах обучения, экзамены по которым сданы студентами, имеющими идентификаторы 12 и 32.
3. Напишите запрос, который выполняет вывод названий предметов обучения, начинающихся на букву 'И'.
4. Напишите запрос, выбирающий сведения о студентах, у которых имена начинаются на букву 'И' или 'С'.
5. Напишите запрос для выбора из таблицы EXAM_MARKS записей, для которых отсутствуют значения оценок (поле MARK).
6. Напишите запрос, выполняющий вывод из таблицы EXAM_MARKS записей, для которых в поле MARK проставлены значения оценок.

7. Напишите запрос для получения списка преподавателей, проживающих в городах, в названиях которых присутствует дефис.
8. Напишите запрос для получения списка учебных заведений, в названиях которых использованы кавычки.
9. Напишите запрос для получения списка предметов, названия которых оканчиваются на `'ия'`.
10. Напишите запрос для получения списка учебных заведений, в названиях которых содержится слово `'университет'`.
11. Напишите запрос для получения списка студентов, фамилии которых начинаются на `'Ков'` или на `'Куз'`.
12. Напишите запрос для получения списка предметов обучения, названия которых состоят из более одного слова.
13. Напишите запрос для получения списка учебных заведений, названия которых состоят как минимум из 7 слов.
14. Напишите запрос для получения списка студентов, фамилии которых состоят из трех букв.

2.3. Преобразование вывода и встроенные функции

В SQL реализованы операторы преобразования данных и встроенные функции, предназначенные для работы со значениями столбцов и/или константами в выражениях. Использование этих операторов допустимо в запросах везде, где можно использовать выражения.

2.3.1. Числовые, символьные и строковые константы. Несмотря на то, что SQL работает с данными в понятиях строк и столбцов таблиц, имеется возможность применения значений выражений, построенных с использованием встроенных функций, констант, имен столбцов, которые определяются как своего рода виртуальные столбцы. Они помещаются в списке столбцов и могут сопровождаться псевдонимами.

Если в запросе вместо спецификации столбца SQL обнаруживает *число*, то оно интерпретируется как *числовая константа*.

Символьные константы должны указываться в одинарных кавычках. Если одинарная кавычка должна выводиться как часть строковой константы, то ее нужно предварить другой одинарной кавычкой.

Например, результатом выполнения запроса

```
SELECT 'Фамилия', SURNAME, 'Имя', NAME, 100
   FROM STUDENT;
```

является таблица следующего вида:

	SURNAME		NAME	
Фамилия	Иванов	Имя	Иван	100
Фамилия	Петров	Имя	Петр	100
Фамилия	Сидоров	Имя	Вадим	100
Фамилия	Кузнецов	Имя	Борис	100
Фамилия	Зайцева	Имя	Ольга	100
Фамилия	Павлов	Имя	Андрей	100
Фамилия	Котов	Имя	Павел	100
Фамилия	Лукин	Имя	Артем	100
Фамилия	Петров	Имя	Антон	100
Фамилия	Белкин	Имя	Вадим	100
...	...	...	...	...

2.3.2. Арифметические операции для преобразования числовых данных.

- Унарный (одиночный) оператор – (знак минус) изменяет знак числового значения, перед которым он стоит, на противоположный.
- Бинарные операторы +, –, * и / предоставляют возможность выполнения арифметических операций сложения, вычитания, умножения и деления.

Например, результат запроса

```
SELECT SURNAME, NAME, STIPEND, -(STIPEND*KURS)/2
   FROM STUDENT
   WHERE KURS = 4 AND STIPEND > 0;
```

будет выглядеть следующим образом

SURNAME	NAME	STIPEND	KURS	
Сидоров	Вадим	150	4	–300
Петров	Антон	200	4	–400
...	...	...	...	...

2.3.3. Символьная операция конкатенации строк. Операция конкатенации, обозначаемая символом "||", позволяет соединять ("склеивать") значения двух или более столбцов символьного типа или символьных констант в одну строку.

Эта операция имеет синтаксис

<значимое символьное выражение> || *<значимое символьное выражение>*.

Например:

```
SELECT SURNAME || '_' || NAME, STIPEND
   FROM STUDENT
   WHERE KURS = 4 AND STIPEND > 0;
```

Результат запроса будет выглядеть следующим образом:

	STIPEND
Сидоров_Вадим	150
Петров_Антон	200
…	…

2.3.4. Символьные функции преобразования букв различных слов в строке.

- **LOWER** — перевод в строчные символы (нижний регистр)

  LOWER(<*строка*>)

- **UPPER** — перевод в прописные символы (верхний регистр)

  UPPER(<*строка*>)

- **INITCAP** — перевод первой буквы каждого слова строки в заглавную (прописную)

  INITCAP(<*строка*>)

Например:

```
SELECT LOWER(SURNAME), UPPER(NAME)
   FROM STUDENT
   WHERE KURS = 4 AND STIPEND > 0;
```

Результат запроса будет выглядеть следующим образом

SURNAME	NAME
Сидоров	ВАДИМ
Петров	АНТОН
…	…

2.3.5. Символьные строковые функции.

- **LPAD** — дополнение строки слева

  LPAD(<*строка*>, <*длина*>[, <*подстрока*>])

  – <*строка*> дополняется ***слева*** указанной в <*подстроке*> последовательностью символов до указанной <*длины*> (возможно, с повторением последовательности);

  – если <*подстрока*> не указана, то по умолчанию <*строка*> дополняется пробелами;

  – если <*длина*> меньше длины <*строки*>, то исходная <*строка*> усекается слева до заданной <*длины*>.

- **RPAD** — дополнение строки справа

  RPAD(<*строка*>, <*длина*>[, <*подстрока*>])

  – <*строка*> дополняется ***справа*** указанной в <*подстроке*> последовательностью символов до указанной <*длины*> (возможно, с повторением последовательности);
  – если <*подстрока*> не указана, то по умолчанию <*строка*> дополняется пробелами;
  – если <*длина*> меньше длины <*строки*>, то исходная <*строка*> усекается справа до заданной <*длины*>.

- **LTRIM** — удаление левых граничных символов

  LTRIM(<*строка*>[, <*подстрока*>])

  – из <*строки*> удаляются слева символы, указанные в <*подстроке*>;
  – если <*подстрока*> не указана, то по умолчанию удаляются пробелы;
  – в <*строку*> справа добавляется столько пробелов, сколько символов слева было удалено, т. е. длина <*строки*> остается неизменной.

- **RTRIM** — удаление правых граничных символов

  RTRIM(<*строка*>[, <*подстрока*>])

  – из <*строки*> удаляются справа символы, указанные в <*подстроке*>;
  – если <*подстрока*> не указана, то по умолчанию удаляются пробелы;
  – в <*строку*> слева добавляется столько пробелов, сколько символов справа было удалено, т. е. длина <*строки*> остается неизменной.

Функции **LTRIM** и **RTRIM** рекомендуется использовать при написании условных выражений, в которых сравниваются текстовые строки. Дело в том, что наличие начальных или конечных пробелов в сравниваемых операндах может исказить результат сравнения.

Например, константы `' AAA'` и `' AAA '` не равны друг другу.

- **SUBSTR** — выделение подстроки

  SUBSTR(<*строка*>, <*начало*>[, <*количество*>])

  – из <*строки*> выбирается заданное <*количество*> символов, начиная с указанной позиции в строке <*начало*>;
  – если <*количество*> не задано, символы выбираются с <*начала*> и до конца <*строки*>;

- возвращается подстрока, содержащая число символов, заданное параметром <*количество*>, либо число символов от позиции, заданной параметром <*начало*>, до конца <*строки*>;
- если указанное <*начало*> превосходит длину <*строки*>, то возвращается строка, состоящая из пробелов. Длина этой строки будет равна заданному <*количеству*> или исходной длине <*строки*> (при не заданном <*количестве*>).

- **INSTR** — поиск подстроки

  INSTR(<*строка*>, <*подстрока*>[, <*начало поиска*>[, <*номер вхождения*>]])

  - <*начало поиска*> задает начальную позицию в строке для поиска <*подстроки*>; если не задано, то по умолчанию принимается значение 1;
  - <*номер вхождения*> задает порядковый номер искомой подстроки; если не задан, то по умолчанию принимается значение 1;
  - значимые выражения в <*начале поиска*> или в <*номере вхождения*> должны иметь беззнаковый целый тип или приводиться к этому типу;
  - тип возвращаемого значения — **INT**;
  - функция возвращает позицию найденной подстроки.

- **LENGTH** — определение длины строки

  LENGTH(<*строка*>)

  - длина <*строки*>, тип возвращаемого значения — **INT**;
  - функция возвращает **NULL**, если <*строка*> имеет **NULL**-значение.

Пример 1.
Результат запроса

```
SELECT LPAD(SURNAME, 10, '@'), RPAD(NAME, 10, '$')
  FROM STUDENT
  WHERE KURS = 3 AND STIPEND > 0;
```

будет выглядеть следующим образом

@@@@Петров	Петр$$$$$$
@@@@Павлов	Андрей$$$$
@@@@@Лукин	Артем$$$$$
…	…

Пример 2.
Запрос

```
SELECT SUBSTR(NAME, 1, 1) || '.' || SURNAME, CITY,
     LENGTH(CITY)
  FROM STUDENT
  WHERE KURS IN(2, 3, 4) AND STIPEND > 0;
```

выдаст результат

	CITY	
П.Петров	Курск	5
С.Сидоров	Москва	6
О.Зайцева	Липецк	6
А.Лукин	Воронеж	7
А.Петров	**NULL**	**NULL**
...	...	...

2.3.6. Функции работы с числами.

- **ABS** — абсолютное значение

  ABS(<*значимое числовое выражение*>)

- **FLOOR** — наибольшее целое, не превосходящее заданное число с плавающей точкой

  FLOOR(<*значимое числовое выражение*>)

- **CEIL** — наименьшее целое, которое равно или больше заданного числа

  CEIL(<*значимое числовое выражение*>)

- Функция округления — **ROUND**

  ROUND(<*значимое числовое выражение*>, <*точность*>)

аргумент <*точность*> задает точность округления (см. пример ниже).

- Функция усечения — **TRUNC**

  TRUNC(<*значимое числовое выражение*>, <*точность*>)

- Тригонометрические функции — **COS**, **SIN**, **TAN**

  COS(<*значимое числовое выражение*>)
  SIN(<*значимое числовое выражение*>)
  TAN(<*значимое числовое выражение*>)

- Гиперболические функции — **COSH**, **SINH**, **TANH**

  COSH(<*значимое числовое выражение*>)
  SINH(<*значимое числовое выражение*>)
  TANH(<*значимое числовое выражение*>)

- Экспоненциальная функция — **EXP**

  EXP(<*значимое числовое выражение*>)

- Логарифмические функции — **LN**, **LOG**

  LN(<*значимое числовое выражение*>)
  LOG(<*значимое числовое выражение*>)

- Функция возведения в степень — **POWER**

  POWER(<*значимое числовое выражение*>,
  <*показатель степени*>)

- Определение знака числа — **SIGN**

  SIGN(<*значимое числовое выражение*>)

- Вычисление квадратного корня — **SQRT**

  SQRT(<*значимое числовое выражение*>)

Пример.
Запрос

```
SELECT UNIV_NAME, RATING,
   ROUND(RATING, -1), TRUNC(RATING, -1)
   FROM UNIVERSITY;
```

вернет результат

UNIV_NAME	RATING		
МГУ	610	610	600
ВГУ	296	300	290
НГУ	345	350	340
РГУ	421	420	410
БГУ	326	330	320
ТГУ	373	370	360
ВГМА	327	330	320
…	…	…	…

2.3.7. Функции преобразования значений.

- Преобразование в символьную строку — **TO_CHAR**

  TO_CHAR(<*значимое выражение*>[, <*символьный формат*>])

  – <*значимое выражение*> должно представлять числовое значение или значение типа дата-время;

  – для числовых значений <*символьный формат*> должен иметь синтаксис [S]9[9…][.9[9…]], где S — представление знака числа (при отсутствии предполагается без отображе-

ния знака), 9 — представление цифр-знаков числового значения (для каждого знакоместа). Символьный формат определяет вид отображения чисел. По умолчанию для числовых значений используется формат '999999.99';

– для значений типа **ДАТА-ВРЕМЯ** <*символьный формат*> имеет вид (т. е. вид отображения значений даты и времени)

— в части даты:

'DD-Mon-YY'
'DD-Mon-YYYY'
'MM/DD/YY'
'MM/DD/YYYY'
'DD.MM.YY'
'DD.MM.YYYY'

— в части времени:

'HH24'
'HH24:MI'
'HH24:MI:SS'
'HH24:MI:SS.FF'

где:

HH24 — часы в диапазоне от 0 до 24;
MI — минуты;
SS — секунды;
FF — тики (сотые доли секунды).

При выводе времени в качестве разделителя по умолчанию используется двоеточие (":"), но при желании можно использовать любой другой символ.

Возвращаемое значение — символьное представление <*значимого выражения*> в соответствии с заданным <*символьным форматом*> преобразования.

- Преобразование из символьного значения в числовое — **TO_NUMBER**

  TO_NUMBER(<*значимое символьное выражение*>)

  При этом <*значимое символьное выражение*>должно задавать символьное значение числового типа.
- Преобразование символьной строки в дату — **TO_DATE**

  TO_DATE(<*значимое символьное выражение*>[,
  <*символьный формат*>])

  – <*значимое символьное выражение*> должно задавать символьное значение типа **ДАТА-ВРЕМЯ**;

– <*символьный формат*> должен описывать представление значения типа **ДАТА–ВРЕМЯ** в <*значимом символьном выражении*>. Допустимые форматы (в том числе и формат по умолчанию) приведены выше.

Возвращаемое значение — <*значимое символьное выражение*> во внутреннем представлении. Тип возвращаемого значения — **DATE**. Над значениями типа **DATE** разрешены следующие операции:

- к значению типа **DATE** можно прибавлять значения типа **INTERVAL**, в результате чего получается значение типа **DATE**;
- при вычитании двух значений типа **DATE** получается значение типа **INTERVAL**;
- при вычитании из значения типа **DATE** значения типа **INTERVAL** получается значение типа **DATE**.

В бинарных операциях один из операндов должен иметь значение отдельного элемента даты: только год, или только месяц, или только день.

Пример.
Запрос

```
SELECT SURNAME, NAME, BIRTHDAY,
     TO_CHAR(BIRTHDAY, 'DD-Mon-YYYY'),
     TO_CHAR(BIRTHDAY, 'DD.MM.YY')
  FROM STUDENT;
```

вернет результат

SURNAME	NAME	BIRTHDAY		
Иванов	Иван	3/12/1988	3-дек-1988	3.12.88
Петров	Петр	11/12/1986	11-дек-1986	11.12.86
Сидоров	Вадим	7/06/1985	7-июн-1985	7.06.85
Кузнецов	Борис	8/12/1987	8-дек-1987	8.12.87
Зайцева	Ольга	21/05/1987	21-май-1987	21.05.87
Павлов	Андрей	5/11/1985	5-ноя-1985	5.11.85
Котов	Павел	**NULL**	**NULL**	**NULL**
Лукин	Артем	11/12/1987	11-дек1987	11.12.87
Петров	Антон	5/08/1987	5-авг-1987	5.08.87
Белкин	Вадим	20/01/1986	20-янв-1986	20.01.86
…	…	…	…	…

Функция **CAST** является средством явного преобразования данных из одного типа в другой. Синтаксис этой команды имеет вид

CAST <*значимое выражение*> **AS** <*тип данных*>

- <*значимое выражение*> должно иметь числовой или символьный тип языка SQL (возможно, с указанием длины, точности и масштаба) или быть **NULL**-значением.

- Любое числовое выражение может быть явно преобразовано в любой другой числовой тип.
- Символьное выражение может быть преобразовано в любой числовой тип. При этом в результате такого преобразования отсекаются начальные и конечные пробелы, а остальные символы преобразуются в числовое значение по правилам языка SQL.
- Если заданная явным образом длина символьного типа недостаточна и преобразованное значение не размещается в нем, то результативное значение усекается справа.
- Возможно явное преобразование символьного типа в символьный с другой длиной. Если длина результата больше длины аргумента, то значение дополняется пробелами; если меньше, то усекается.
- **NULL** преобразуется в **NULL**.
- Числовое выражение может быть преобразовано в символьный тип.

Пример.

```
SELECT CAST STUDENT_ID AS CHAR(10)
   FROM STUDENT;
```

УПРАЖНЕНИЯ

1. Составьте запрос для таблицы STUDENT таким образом, чтобы выходная таблица содержала один столбец, содержащий последовательность разделенных символом ";" (точка с запятой) значений всех столбцов этой таблицы; при этом текстовые значения должны отображаться прописными символами (верхний регистр), т. е. быть представленными в следующем виде:
   10;КУЗНЕЦОВ;БОРИС;0;БРЯНСК; 8.12.1987;10.
2. Составьте запрос для таблицы STUDENT таким образом, чтобы выходная таблица содержала всего один столбец в следующем виде: Б.КУЗНЕЦОВ; место жительства - БРЯНСК; родился - 8.12.87.
3. Составьте запрос для таблицы STUDENT таким образом, чтобы выходная таблица содержала всего один столбец в следующем виде: б.кузнецов; место жительства - брянск; родился: 8-дек-1987.
4. Составьте запрос для таблицы STUDENT таким образом, чтобы выходная таблица содержала всего один столбец в следующем виде: Борис Кузнецов родился в 1987 году.
5. Составьте запрос, выводящий фамилии, имена студентов и величину получаемых ими стипендий, при этом значения стипендий должны быть увеличены в 100 раз.
6. То же, что и в упр. 4, но только для студентов 1, 2 и 4 курсов и таким образом, чтобы фамилии и имена были выведены прописными буквами.

7. Составьте запрос для таблицы UNIVERSITY таким образом, чтобы выходная таблица содержала всего один столбец в следующем виде: `Код-10; ВГУ-г.ВОРОНЕЖ; Рейтинг=296`.
8. То же, что и в упр. 7, но значения рейтинга требуется округлить до первого знака (например, значение 382 округляется до 400).

2.4. Агрегирование и групповые функции

Агрегирующие функции позволяют получать из таблицы сводную (агрегированную) информацию, выполняя операции над группой строк таблицы. Для задания в **SELECT**-запросе агрегирующих операций используются следующие ключевые слова:

- **COUNT** определяет количество строк или значений поля, выбранных посредством запроса и не являющихся **NULL**-значениями;
- **SUM** вычисляет арифметическую сумму всех выбранных значений данного поля;
- **AVG** вычисляет среднее значение для всех выбранных значений данного поля;
- **MAX** вычисляет наибольшее из всех выбранных значений данного поля;
- **MIN** вычисляет наименьшее из всех выбранных значений данного поля.

В **SELECT**-запросе агрегирующие функции используются аналогично именам полей, при этом последние (имена полей) используются в качестве аргументов этих функций.

Функция **AVG** предназначена для подсчета среднего значения поля на множестве записей таблицы.

Например, для определения среднего значения поля MARK (оценки) по всем записям таблицы EXAM_MARKS можно использовать запрос с функцией **AVG** следующего вида:

```
SELECT AVG(MARK)
   FROM EXAM_MARKS;
```

Для подсчета общего количества строк в таблице следует использовать функцию **COUNT** со звездочкой:

```
SELECT COUNT(*)
   FROM EXAM_MARKS;
```

При подсчете значений конкретных атрибутов аргументы **DISTINCT** и **ALL** позволяют соответственно исключать и включать дубликаты обрабатываемых функцией **COUNT** значений. При этом необходимо учитывать, что при использовании опции **ALL** неопределенные значения атрибута (**NULL**) все равно не войдут в число подсчитываемых значений.

```
SELECT COUNT(DISTINCT SUBJ_ID)
   FROM SUBJECT;
```

Предложение **GROUP BY** (ГРУППИРОВАТЬ ПО) позволяет группировать записи в подмножества, определяемые значениями какого-либо поля, и применять агрегирующие функции уже не ко всем записям таблицы, а раздельно к каждой сформированной группе.

Предположим, требуется найти максимальное значение оценки, полученной каждым студентом. Запрос будет выглядеть следующим образом:

```
SELECT STUDENT_ID, MAX(MARK)
   FROM EXAM_MARKS
   GROUP BY STUDENT_ID;
```

Выбираемые из таблицы EXAM_MARKS записи группируются по значениям поля STUDENT_ID, указанного в предложении **GROUP BY**, и для каждой группы находится максимальное значение поля MARK. Предложение **GROUP BY** позволяет применять агрегирующие функции к каждой группе, определяемой общим значением поля или полей, указанных в этом предложении. В приведенном запросе рассматриваются группы записей, сгруппированные по идентификаторам студентов.

В конструкции **GROUP BY** для группирования может быть использовано более одного столбца. Например:

```
SELECT STUDENT_ID, SUBJ_ID, MAX(MARK)
   FROM EXAM_MARKS
   GROUP BY STUDENT_ID, SUBJ_ID;
```

В этом случае строки вначале группируются по значениям первого столбца, а внутри этих групп — в подгруппы по значениям второго столбца. Таким образом, **GROUP BY** не только устанавливает столбцы, по которым осуществляется группировка, но и указывает порядок разбиения столбцов на группы.

Следует иметь в виду, что после ключевого слова **SELECT** должны быть использованы только те имена столбцов, которые указаны в предложении **GROUP BY**.

При необходимости часть сформированных с помощью **GROUP BY** групп может быть исключена с помощью предложения **HAVING**.

Предложение **HAVING** определяет критерий, по которому группы следует включать в выходные данные (по аналогии с предложением **WHERE**, которое осуществляет это для отдельных строк):

```
SELECT SUBJ_NAME, MAX(HOUR)
   FROM SUBJECT
   GROUP BY SUBJ_NAME
   HAVING MAX(HOUR) >= 72;
```

В условии, задаваемом предложением **HAVING**, должны быть указаны только поля или выражения, которые на выходе имеют единственное значение для каждой выводимой группы.

УПРАЖНЕНИЯ

1. Напишите запрос для подсчета количества студентов, сдававших экзамен по предмету обучения с идентификатором 20.
2. Напишите запрос, который позволяет подсчитать в таблице EXAM_MARKS количество различных предметов обучения.
3. Напишите запрос, который для каждого студента выполняет выборку его идентификатора и минимальной из полученных им оценок.
4. Напишите запрос, который для каждого студента выполняет выборку его идентификатора и максимальной из полученных им оценок.
5. Напишите запрос, выполняющий вывод первой по алфавиту фамилии студента, начинающейся на букву 'И'.
6. Напишите запрос, который для каждого предмета обучения выводит наименование предмета и максимальное значение номера семестра, в котором этот предмет преподается.
7. Напишите запрос, который для каждого конкретного дня сдачи экзамена выводит данные о количестве студентов, сдававших экзамен в этот день.
8. Напишите запрос, выдающий средний балл для каждого студента.
9. Напишите запрос, выдающий средний балл для каждого экзамена.
10. Напишите запрос, определяющий количество сдававших студентов для каждого экзамена.
11. Напишите запрос для определения количества предметов, изучаемых на каждом курсе.
12. Для каждого университета напишите запрос, выводящий суммарную стипендию обучающихся в нем студентов, с последующей сортировкой списка по этому значению.
13. Для каждого семестра напишите запрос, выводящий общее количество часов, отводимое на изучение соответствующих предметов.
14. Для каждого студента напишите запрос, выводящий среднее значение оценок, полученных им на всех экзаменах.
15. Для каждого студента напишите запрос, выводящий среднее значение оценок, полученных им по каждому предмету.
16. Напишите запрос, выводящий количество студентов, проживающих в каждом городе. Список отсортировать в порядке убывания количества студентов.

17. Для каждого университета напишите запрос, выводящий количество обучающихся в нем студентов, с последующей сортировкой списка по этому количеству.
18. Для каждого университета напишите запрос, выводящий количество работающих в нем преподавателей, с последующей сортировкой списка по этому количеству.
19. Для каждого университета напишите запрос, выводящий сумму стипендии, выплачиваемой студентам каждого курса.
20. Для каждого города напишите запрос, выводящий максимальный рейтинг университетов, в нем расположенных, с последующей сортировкой списка по значениям рейтингов.
21. Для каждого дня сдачи экзаменов напишите запрос, выводящий среднее значение всех экзаменационных оценок.
22. Для каждого дня сдачи экзаменов напишите запрос, выводящий максимальные оценки, полученные по каждому предмету.
23. Для каждого дня сдачи экзаменов напишите запрос, выводящий общее количество студентов, сдававших экзамены.
24. Для каждого дня сдачи экзаменов напишите запрос, выводящий общее количество экзаменов, сдававшихся каждым студентом.
25. Для каждого преподавателя напишите запрос, выводящий количество преподаваемых им предметов.
26. Для каждого предмета напишите запрос, выводящий количество преподавателей, ведущих по нему занятия.
27. Напишите запрос, выполняющий вывод количества студентов, имеющих только отличные оценки.
28. Напишите запрос, выполняющий вывод количества экзаменов, сданных (с положительной оценкой) студентом с идентификатором 32.

2.5. Неопределенные значения (NULL) в агрегирующих функциях

Отсутствие значений в полях таблицы, обозначенное ключевым словом **NULL**, приводит к особенностям при выполнении агрегирующих операций над данными, которые следует учитывать при их использовании в SQL-запросах.

2.5.1. Влияние NULL-значений в функции COUNT. Если аргумент функции **COUNT** является константой или столбцом без пустых значений, то функция возвращает количество строк, к которым применимо определенное условие или группирование.

Если аргументом функции является *столбец*, содержащий пустое значение, то **COUNT** вернет число строк, которые не содержат пустые значения и к которым применимо определенное условие или группирование.

Если бы механизм **NULL** не был доступен, то неприменимые и отсутствующие значения пришлось бы исключать с помощью конструкции **WHERE**.

Поведение функции **COUNT**(*) не зависит от пустых значений. Она возвратит общее количество строк в таблице.

2.5.2. Влияние NULL-значений в функции AVG. *Среднее* значение множества чисел равно сумме чисел, деленной на число элементов множества. Однако если некоторые элементы пусты, т. е. их значения неизвестны или не существуют, то деление на количество всех элементов множества приведет к неправильному результату.

Функция **AVG** вычисляет среднее значение всех *известных* значений множества элементов, т. е. эта функция подсчитывает сумму *известных* значений и делит ее на количество *этих* значений, а не на общее количество значений, среди которых могут быть **NULL**-значения. Если столбец состоит только из пустых значений, то функция **AVG** также возвратит **NULL**.

2.6. Результат действия трехзначных условных операторов

Условные операторы при отсутствии пустых значений возвращают либо **TRUE** (ИСТИНА), либо **FALSE** (ЛОЖЬ). Если же в столбце присутствуют пустые значения, то может быть возвращено и третье значение: **UNKNOWN** (НЕИЗВЕСТНО). В этой схеме, например, условию **WHERE** A=2, где A — имя столбца, значения которого могут быть неизвестны, при A=2 будет соответствовать **TRUE**, при A=4 в результате будет получено значение **FALSE**, а при отсутствующем значении A (**NULL**-значение) результат будет **UNKNOWN**. Пустые значения оказывают влияние на использование логических операторов **NOT**, **AND** и **OR**.

Оператор NOT

Обычный унарный оператор **NOT** обращает оценку **TRUE** в **FALSE** и наоборот. Однако **NOT**, примененный к неизвестному значению **UNKNOWN**, будет возвращать **UNKNOWN**. При этом следует отличать случай **NOT UNKNOWN** от условия **IS NOT NULL**, которое является противоположностью выражения **IS NULL**, отделяя известные значения от неизвестных.

Оператор AND

- Если результат двух условий, объединенных оператором **AND**, известен, то применяются правила булевой логики, т. е. при обоих утверждениях **TRUE** составное утверждение также будет **TRUE**. Если же хотя бы одно из двух утверждений будет **FALSE**, то составное утверждение будет **FALSE**.
- Если результат одного из утверждений неизвестен, а другой оценивается как **TRUE**, то состояние неизвестного утверждения

является определяющим, и, следовательно, итоговый результат также неизвестен.
- Если результат одного из утверждений неизвестен, а другой оценивается как **FALSE**, то итоговый результат будет **FALSE**.
- Если результат обоих утверждений неизвестен, то результат также остается неизвестным.

Оператор OR
- Если результат двух условий, объединенных оператором **OR**, известен, то применяются правила булевой логики, а именно: если хотя бы одно из двух утверждений соответствует **TRUE**, то и составное утверждение будет **TRUE**; если оба утверждения оцениваются как **FALSE**, то и составное утверждение будет **FALSE**.
- Если результат одного из утверждений неизвестен, а другой оценивается как **TRUE**, итоговый результат будет **TRUE**.
- Если результат одного из утверждений неизвестен, а другой оценивается как **FALSE**, то состояние неизвестного утверждения играет роль. Следовательно, итоговый результат также неизвестен.
- Если результат обоих утверждений неизвестен, то результат также остается неизвестным.

Примечание. Отсутствующие (**NULL**) значения целесообразно использовать в столбцах, предназначенных для агрегирования, чтобы извлечь преимущества из способа обработки пустых значений в функциях **COUNT** и **AVG**. Так как при наличии **NULL** существенно усложняется корректное построение условий отбора, приводя иногда к неожиданным для составителя запроса результатам выборки, то во всех остальных случаях следует по возможности избегать использования пустых значений. В частности, для индикации отсутствующих, неприменимых или по какой-то причине неизвестных данных можно использовать значения по умолчанию, устанавливаемые заранее (например, с помощью команды **CREATE TABLE** (раздел 4.1).

2.7. Упорядочение выходных полей (ORDER BY)

Как уже отмечалось, записи в таблицах реляционной базы данных неупорядочены. Однако, данные, выводимые в результате выполнения запроса, могут быть упорядочены. Для этого в операторе **SELECT** используется предложение с ключевым словом **ORDER BY**, которое позволяет упорядочивать выводимые записи в соответствии со значениями одного или нескольких выбранных столбцов. При этом можно задать возрастающую (**ASC**) или убывающую (**DESC**) последовательность сортировки для каждого из столбцов. По умолчанию принята возрастающая последовательность сортировки.

Запрос, позволяющий выбрать все данные из таблицы предметов обучения SUBJECT, с упорядочением по наименованиям предметов, выглядит следующим образом:

```
SELECT *
   FROM SUBJECT
   ORDER BY SUBJ_NAME;
```

Тот же список, но упорядоченный в обратном порядке, можно получить запросом:

```
SELECT *
   FROM SUBJECT
   ORDER BY SUBJ_NAME DESC;
```

Можно упорядочить выводимый список предметов обучения по значениям семестров, а внутри семестров — по наименованиям предметов:

```
SELECT *
   FROM SUBJECT
   ORDER BY SEMESTR, SUBJ_NAME;
```

Предложение **ORDER BY** может использоваться с **GROUP BY**. При этом оператор **ORDER BY** в запросе *всегда должен быть* ***последним***:

```
SELECT SUBJ_NAME, SEMESTR, MAX(HOUR)
   FROM SUBJECT
   GROUP BY SEMESTR, SUBJ_NAME
   ORDER BY SEMESTR;
```

При упорядочивании вместо наименований столбцов можно указывать их номера, имея, однако, в виду, что в данном случае это номера столбцов, указанные при определении выходных данных в запросе, а не номера столбцов в таблице. Полем с номером 1 является первое поле, указанное в предложении **ORDER BY** — независимо от его расположения в таблице. В запросе

```
SELECT SUBJ_ID, SEMESTR
   FROM SUBJECT
   ORDER BY 2 DESC;
```

выводимые записи будут упорядочены по полю SEMESTR.

Если в поле, которое используется для упорядочивания, существуют **NULL**-значения, то все они размещаются в конце или предшествуют всем остальным значениям этого поля.

УПРАЖНЕНИЯ

1. Предположим, что стипендия всем студентам увеличена на 20%. Напишите запрос к таблице STUDENT, выполняющий вывод номера студента, его фамилии и величины увеличенной стипендии. Выходные данные упорядочите: а) по значению последнего столбца (величине стипендии); б) в алфавитном порядке фамилий студентов.

2. Напишите запрос, который по таблице EXAM_MARKS позволяет найти а) максимальные и б) минимальные оценки каждого студента и выводит их вместе с идентификатором студента.
3. Напишите запрос, выполняющий вывод списка предметов обучения в порядке а) убывания семестров и б) возрастания количества отводимых на предмет часов. Поле семестра в выходных данных должно быть первым, за ним должны следовать имя предмета обучения и идентификатор предмета.
4. Напишите запрос, который для каждой даты сдачи экзаменов выполняет вывод суммы баллов всех студентов и представляет результаты в порядке убывания этих сумм.
5. Напишите запрос, который для каждой даты сдачи экзаменов выполняет вывод а) среднего, б) минимального, в) максимального баллов всех студентов и представляет результаты в порядке убывания этих значений.

2.8. Вложенные подзапросы

SQL позволяет использовать одни запросы внутри других запросов, т. е. вкладывать запросы друг в друга. Предположим, известна фамилия студента (Петров), но неизвестно значение поля STUDENT_ID для него. Чтобы извлечь данные обо всех оценках этого студента, можно записать следующий запрос:

```
SELECT *
   FROM EXAM_MARKS
   WHERE STUDENT_ID =
      (SELECT STUDENT_ID
       FROM STUDENT SURNAME = 'Петров');
```

Следует обратить внимание, что этот корректен только в том случае, если в результате выполнения указанного в скобках *подзапроса* возвращается *единственное значение*. Если в результате выполнения подзапроса будет возвращено несколько значений, то при выполнении запроса будет зафиксирована ошибка. В данном примере это произойдет, если в таблице STUDENT будет несколько записей со значениями поля SURNAME = 'Петров'.

В некоторых случаях для гарантии получения единственного значения в результате выполнения подзапроса используется **DISTINCT**. Одним из видов функций, которые автоматически *всегда* выдают в результате единственное значение для любого количества строк, являются агрегирующие функции.

Оператор **IN** также широко применяется в подзапросах. Он задает список значений, с которыми сравниваются другие значения для определения истинности задаваемого этим оператором предиката.

Данные обо всех оценках (таблица EXAM_MARKS) студентов из Воронежа можно выбрать с помощью следующего запроса:

```
SELECT *
   FROM EXAM_MARKS
   WHERE STUDENT_ID IN
      (SELECT STUDENT_ID
       FROM STUDENT
       WHERE CITY = 'Воронеж');
```

Подзапросы можно применять внутри предложения **HAVING**. Пусть требуется определить количество предметов обучения с оценкой, превышающей среднее значение оценки студента с идентификатором 301:

```
SELECT COUNT(DISTINCT SUBJ_ID), MARK
   FROM EXAM_MARKS
   GROUP BY MARK
   HAVING MARK >
      (SELECT AVG(MARK)
       FROM EXAM_MARKS
       WHERE STUDENT_ID = 301);
```

УПРАЖНЕНИЯ

1. Напишите запрос, выводящий список студентов, получающих максимальную стипендию, отсортировав его в алфавитном порядке по фамилиям.
2. Напишите запрос, выводящий список студентов, получающих стипендию, превышающую среднее значение стипендии.
3. Напишите запрос, выводящий список студентов, обучающихся в Воронеже, с последующей сортировкой по идентификаторам университетов и курсам.
4. Напишите запрос, выводящий список предметов, на изучение которых отведено максимальное количество часов.
5. Напишите запрос, выполняющий вывод имен и фамилий студентов, место проживания которых не совпадает с городом, в котором находится их университет.
6. Напишите запрос, выводящий список университетов, расположенных в Москве и имеющих рейтинг меньший, чем у ВГУ.

2.9. Формирование связанных подзапросов

При использовании подзапросов во внутреннем запросе можно ссылаться на таблицу, имя которой указано в предложении **FROM** внешнего запроса. Такие подзапросы называются связанными.

Связанный подзапрос выполняется по одному разу для каждой строки таблицы основного запроса, а именно:

- выбирается строка из таблицы, имя которой указано во внешнем запросе;

- выполняется подзапрос, и полученное в результате его выполнения значение применяется для анализа этой строки в условии предложения **WHERE** внешнего запроса;
- по результату оценки этого условия принимается решение о включении или невключении строки в состав выходных данных;
- процедура повторяется для следующей строки таблицы внешнего запроса.

Пример.

Выбрать сведения обо всех предметах обучения, по которым проводился экзамен 20 января 2005 г.

```
SELECT *
   FROM SUBJECT SU
   WHERE '20/01/2005' IN
      (SELECT EXAM_DATE
       FROM EXAM_MARKS EX
       WHERE SU.SUBJ_ID = EX.SUBJ_ID);
```

В некоторых СУБД для выполнения этого запроса, возможно, потребуется преобразование значения даты в символьный тип. В приведенном запросе SU и EX являются псевдонимами (алиасами), т. е. специально вводимыми именами, которые могут быть использованы в данном запросе вместо настоящих имен. В приведенном примере они используются вместо имен таблиц SUBJECT и EXAM_MARKS.

Эту же задачу можно решить с помощью операции соединения таблиц:

```
SELECT DISTINCT FIRST.SUBJ_ID, SUBJ_NAME,
                HOUR, SEMESTER
   FROM SUBJECT FIRST, EXAM_MARKS SECOND
   WHERE FIRST.SUBJ_ID = SECOND.SUBJ_ID
   AND SECOND.EXAM_DATE = '20/01/2005';
```

В этом выражении алиасами таблиц являются имена FIRST и SECOND.

Можно использовать подзапросы, связывающие таблицу со своей собственной копией. Например, надо найти идентификаторы, фамилии и стипендии студентов, получающих стипендию выше средней на курсе, на котором они учатся:

```
SELECT DISTINCT STUDENT_ID, SURNAME, STIPEND
   FROM STUDENT E1
   WHERE STIPEND >
      (SELECT AVG(STIPEND)
       FROM STUDENT E2
       WHERE E1.KURS = E2.KURS);
```

Тот же результат можно получить с помощью следующего запроса:

```
SELECT DISTINCT STUDENT_ID, SURNAME, STIPEND
   FROM STUDENT E1,
      (SELECT KURS, AVG(STIPEND) AS AVG_STIPEND
       FROM STUDENT E2
       GROUP BY E2.KURS) E3
   WHERE E1.STIPEND > AVG_STIPEND AND
                      E1.KURS=E3.KURS;
```

Обратите внимание — второй запрос должен выполниться гораздо быстрее. Дело в том, что в первом варианте запроса агрегирующая функция **AVG** выполняется над таблицей, указанной в подзапросе, для *каждой* строки внешнего запроса. В другом варианте вторая таблица (алиас E2) обрабатывается агрегирующей функцией один раз, в результате чего формируется вспомогательная таблица (в запросе она имеет алиас E3), со строками которой затем соединяются строки первой таблицы (алиас E1). Следует иметь в виду, что на самом деле реальное время выполнения запроса в большой степени зависит от оптимизатора запросов конкретной СУБД, и, вполне возможно, что такое преобразование запроса будет выполнено оптимизатором.

УПРАЖНЕНИЯ

1. Напишите запрос для получения списка студентов, которые учатся в своем городе.
2. Напишите запрос для получения списка иногородних студентов (обучающихся не в своем городе), с последующей сортировкой по идентификаторам университетов и курсам.
3. Напишите запрос для получения списка преподавателей, работающих не в своем городе, с последующей сортировкой по идентификаторам университетов и городам проживания преподавателей.
4. Напишите запрос для получения списка предметов, на изучение которых отведено максимальное количество часов среди всех предметов, изучаемых в том же семестре. Список упорядочить по семестрам.
5. Напишите запрос для получения списка студентов, получающих стипендию, превосходящую среднее значение стипендии на их курсе.
6. Напишите запрос для получения списка студентов, получающих минимальную стипендию в своем университете, с последующей сортировкой по значениям идентификатора университета и стипендии.
7. Напишите запрос для получения списка университетов, в которых учится более 50 студентов, с последующей сортировкой по рейтингам.
8. Напишите запрос для получения списка университетов, в которых работает более 5 преподавателей, с последующей сортировкой по рейтингам университетов.

9. Напишите запрос для получения списка отличников (студентов, получивших только отличные оценки), с последующей сортировкой по идентификаторам университетов и курсам.
10. Напишите запрос для получения списка неуспевающих студентов (получивших хотя бы одну неудовлетворительную оценку), с последующей сортировкой по идентификаторам университетов и курсам.
11. Напишите запрос, выполняющий вывод списка студентов, средняя оценка которых превышает 4 балла.
12. Напишите запрос, выполняющий вывод списка фамилий студентов, имеющих только отличные оценки и проживающих в городе, не совпадающем с городом их университета.

2.10. Связанные подзапросы в HAVING

В разделе 2.4 указывалось, что предложение **GROUP BY** позволяет группировать выводимые **SELECT**-запросом записи по значению некоторого поля. Использование предложения **HAVING** позволяет при выводе осуществлять фильтрацию таких групп. Предикат предложения **HAVING** оценивается не для каждой строки результата, а для каждой группы выходных записей, сформированной предложением **GROUP BY** внешнего запроса.

Пусть, например, необходимо по данным из таблицы EXAM_MARKS определить сумму полученных студентами оценок (значений поля MARK), сгруппировав значения оценок по датам экзаменов и исключив те дни, когда число студентов, сдававших в течение дня экзамены, было меньше 10:

```
SELECT EXAM_DATE, SUM(MARK)
   FROM EXAM_MARKS A
   GROUP BY EXAM_DATE
   HAVING 10 <
      (SELECT COUNT(MARK)
       FROM EXAM_MARKS B
       WHERE A.EXAM_DATE = B.EXAM_DATE);
```

Подзапрос вычисляет количество строк, у которых значения поля EXAM_DATE (дата экзамена) совпадает с датой, для которой сформирована очередная группа основного запроса.

УПРАЖНЕНИЯ

1. Напишите запрос с подзапросом для получения всех оценок студента с фамилией Иванов. Предположим, что его персональный номер неизвестен. Всегда ли такой запрос будет корректным?
2. Напишите запрос, выбирающий имена всех студентов, имеющих по предмету с идентификатором 101 балл выше общего среднего балла.

3. Напишите запрос, который выполняет выборку имен всех студентов, имеющих по предмету с идентификатором 102 балл ниже общего среднего балла.
4. Напишите запрос, выполняющий вывод количества предметов, по которым экзаменовался каждый студент, сдававший более 20 предметов.
5. Напишите команду **SELECT**, использующую связанные подзапросы и выполняющую вывод имен и идентификаторов студентов, у которых стипендия совпадает с максимальным значением стипендии для города, в котором живет студент.
6. Напишите запрос, который позволяет вывести имена и идентификаторы всех студентов, о которых точно известно, что они проживают в городе, где нет ни одного университета.
7. Напишите два запроса, которые позволяют вывести имена и идентификаторы всех студентов, о которых точно известно, что они проживают не в том городе, где расположен их университет: один запрос с использованием связанного подзапроса, а другой — с использованием соединения.

2.11. Использование оператора EXISTS

Используемое в SQL ключевое слово **EXISTS** (СУЩЕСТВУЕТ) представляет собой предикат, принимающий значение *истина* или *ложь*. Используя подзапросы в качестве аргумента, этот оператор оценивает результат выполнения подзапроса как истинный, если этот подзапрос генерирует выходные данные, то есть в случае *существования* (возврата) хотя бы одного найденного значения. В противном случае результат подзапроса — ложный. Оператор **EXISTS** не может принимать значение *неизвестно*).

Пусть, например, нужно извлечь из таблицы EXAM_MARKS данные о студентах, получивших хотя бы одну неудовлетворительную оценку:

```
SELECT DISTINCT STUDENT_ID
   FROM EXAM_MARKS A
   WHERE EXISTS
      (SELECT *
       FROM EXAM_MARKS B
       WHERE MARK < 3
       AND B.STUDENT_ID=A.STUDENT_ID);
```

При использовании связанных подзапросов предложение **EXISTS** анализирует каждую строку таблицы, на которую имеется ссылка во внешнем запросе. Главный запрос получает строки-кандидаты на проверку условия. Для каждой строки-кандидата выполняется подзапрос. Как только при выполнении подзапроса находится строка, в которой значение в столбце MARK удовлетворяет условию, его выполнение пре-

кращается и возвращается значение ***истина*** внешнему запросу, который затем анализирует свою строку-кандидата.

Например, требуется получить идентификаторы предметов обучения, экзамены по которым сдавались не одним, а несколькими студентами:

```
SELECT DISTINCT SUBJ_ID
   FROM EXAM_MARKS A
   WHERE EXISTS
      (SELECT *
       FROM EXAM_MARKS B
       WHERE A.SUBJ_ID = B.SUBJ_ID
       AND A.STUDENT_ID <> B.STUDENT_ID);
```

Часто **EXISTS** применяется с оператором **NOT** (по-русски **NOT EXISTS** интерпретируется, как "*не существует...*"). Если предыдущий запрос сформулировать следующим образом: "Найти идентификаторы предметов обучения, которые сдавались одним и только одним студентом (другими словами, для которых не существует другого сдававшего студента)," то достаточно просто поставить **NOT** перед **EXISTS**.

Возможности применения вложенных запросов весьма разнообразны. Например, пусть из таблицы STUDENT требуется извлечь строки для каждого студента, сдавшего более одного предмета:

```
SELECT *
   FROM STUDENT FIRST
   WHERE EXISTS
   (SELECT SUBJ_ID
       FROM EXAM_MARKS SECOND
       GROUP BY SUBJ_ID
       HAVING COUNT(SUBJ_ID) > 1
       WHERE FIRST.STUDENT_ID = SECOND.STUDENT_ID);
```

УПРАЖНЕНИЯ

1. Напишите запрос с **EXISTS**, позволяющий вывести данные обо всех студентах обучающихся в вузах, имеющих рейтинг выше 300.
2. Напишите предыдущий запрос, используя соединения.
3. Напишите запрос с **EXISTS**, выбирающий сведения о каждом студенте, для которого в городе его проживания имеется хотя бы один университет, в котором он не учится.
4. Напишите запрос, выбирающий из таблицы SUBJECT данные о названиях предметов обучения, экзамены по которым сданы более чем одним студентом.
5. Напишите запрос для получения списка городов проживания студентов, в которых есть хотя бы один университет.

6. Напишите запрос для получения списка городов проживания студентов, в которых нет ни одного университета.
7. Напишите запрос для получения списка предметов, для которых не назначено ни одного преподавателя.
8. Напишите запрос для получения списка предметов, изучаемых в течение одного семестра.
9. Напишите запрос для получения списка предметов, изучаемых более чем в одном семестре.
10. Напишите запрос для получения списка университетов, в которых не работает ни один преподаватель.
11. Напишите запрос для получения списка университетов, в которых не учится ни один студент.
12. Напишите запрос для получения списка предметов, по которым на экзаменах не получено ни одной неудовлетворительной оценки.
13. Напишите запрос для получения списка предметов, по которым на экзаменах получена хотя бы одна неудовлетворительная оценка.
14. Напишите запрос для получения списка студентов, не получивших на экзаменах ни одной неудовлетворительной оценки.
15. Напишите запрос для получения списка студентов, получивших на экзаменах хотя бы одну неудовлетворительную оценку.
16. Напишите запрос, выполняющий вывод имен и фамилий студентов, получивших хотя бы одну отличную оценку.
17. Напишите запрос, выполняющий вывод имен и фамилий студентов, не получивших ни одной отличной оценки.
18. Напишите запрос, выполняющий вывод количества студентов, не имеющих ни одной оценки.
19. Напишите запрос, выполняющий вывод количества студентов, имеющих только отличные оценки.
20. Напишите запрос, выполняющий вывод количества студентов, имеющих хотя бы одну неудовлетворительную оценку и проживающих в городе, не совпадающем с городом их университета.

2.12. Операторы сравнения с множеством значений IN, ANY, ALL

Операторы сравнения с множеством значений имеют следующий смысл.

IN	*Равно* хотя бы одному из значений, полученных во внутреннем запросе.
NOT IN	*Не равно* ни одному из значений, полученных во внутреннем запросе.
=ANY	То же, что и **IN**. Соответствует логическому оператору **OR**.

>ANY, **>=ANY**	*Больше* (либо *больше или равно*) хотя бы одного из полученных значений. Эквивалентно > (или >=) для наименьшего из полученных значений.
<ANY, **<=ANY**	*Меньше* (либо *меньше или равно*) хотя бы одного из полученных значений. Эквивалент < (или <=) для наибольшего из полученных значений.
=ALL	*Равно* каждому из полученных значений. Эквивалентно логическому оператору **AND**.
>ALL, **>=ALL**	*Больше* (либо *больше или равно*) каждого из полученных значений. Эквивалент > (или >=) для наибольшего из полученных значений.
<ALL, **<=ALL**	*Меньше* (либо *меньше или равно*) каждого из полученных значений. Эквивалентно < (или <=) наименьшего из полученных значений.

Пример 1.

Выбрать сведения о студентах, обучающихся в университете, расположенном в городе их проживания.

```
SELECT *
   FROM STUDENT S
   WHERE CITY = ANY
      (SELECT CITY
       FROM UNIVERSITY U
       WHERE U.UNIV_ID = S.UNIV_ID);
```

Пример 2.

Другой вариант запроса из примера 1:

```
SELECT *
   FROM STUDENT S
   WHERE CITY IN
      (SELECT CITY
       FROM UNIVERSITY U
       WHERE U.UNIV_ID = S.UNIV_ID);
```

Пример 3.

Выборка данных об идентификаторах студентов, у которых оценки превосходят величину по крайней мере одной из оценок, полученных 6 октября 2005 г.:

```
SELECT DISTINCT STUDENT_ID
   FROM EXAM_MARKS
   WHERE MARK > ANY
      (SELECT MARK
```

```
FROM EXAM_MARKS
WHERE EXAM_DATE = '06/10/2005');
```

Оператор **ALL**, как правило, эффективно используется с неравенствами, а не с равенствами, поскольку значение *равно всем*, которое должно получиться в этом случае в результате выполнения подзапроса, может иметь место, только если все результаты идентичны. Такая ситуация практически не может быть реализована, так как если подзапрос генерирует множество различных значений, то никакое отдельное значение не может быть равно сразу всем значениям в обычном смысле. В SQL выражение **<>ALL** реально означает *не равно ни одному* из результатов подзапроса.

Пример 4.
Подзапрос, выбирающий данные о названиях всех университетов с рейтингом более высоким, чем рейтинг любого университета в Воронеже:

```
SELECT *
   FROM UNIVERSITY
   WHERE RATING > ALL
      (SELECT RATING
       FROM UNIVERSITY
       WHERE CITY = 'Воронеж');
```

Пример 5.
В предыдущем примере вместо **ALL** можно также использовать **ANY** (проанализируйте, как в этом случае изменится смысл приведенного запроса?):

```
SELECT *
   FROM UNIVERSITY
   WHERE NOT RATING > ANY
      (SELECT RATING
       FROM UNIVERSITY
       WHERE CITY = 'Воронеж');
```

УПРАЖНЕНИЯ

1. Напишите запрос для получения списка названий предметов, изучаемых в нескольких семестрах.
2. Напишите запрос, выполняющий вывод имен и фамилий студентов, имеющих весь набор положительных (тройки, четверки и пятерки) оценок.
3. Напишите запрос, выполняющий выборку значений идентификаторов студентов, имеющих такие же оценки, что и студент с идентификатором 12.

4. Напишите запрос, выполняющий вывод количества студентов, не имеющих ни одной оценки.
5. Напишите запрос, выполняющий вывод данных о предметах обучения, которые преподает Колесников.
6. Напишите запрос, выполняющий вывод имен и фамилий преподавателей, проводящих занятия на первом курсе.
7. Напишите запрос, выполняющий вывод данных о преподавателях, ведущих обучение хотя бы по одному из предметов обучения, которые преподает Сорокин.
8. Напишите запрос, выполняющий вывод списка фамилий студентов университета, расположенного в городе, название которого стоит первым в алфавитном списке городов.
9. Напишите запрос, выполняющий вывод списка фамилий студентов, имеющих только отличные оценки и проживающих в городе, не совпадающем с городом их университета.

2.13. Особенности применения операторов ANY, ALL, EXISTS при обработке отсутствующих данных

Необходимо иметь в виду, что операторы **ANY** и **ALL** по-разному реагируют на ситуацию, когда правильный подзапрос не генерирует никаких выходных данных. В этом случае оператор **ALL** автоматически принимает значение *истина*, а оператор **ANY** — значение *ложь*.

В случае, когда в базе отсутствуют данные об университетах из города Саратова, запрос

```
SELECT *
   FROM UNIVERSITY
   WHERE RATING > ANY
      (SELECT RATING
       FROM UNIVERSITY
       WHERE CITY = 'Саратов');
```

не генерирует никаких выходных данных. В такой же ситуации запрос

```
SELECT *
   FROM UNIVERSITY
   WHERE RATING > ALL
      (SELECT RATING
       FROM UNIVERSITY
       WHERE CITY = 'New York');
```

полностью воспроизведет таблицу UNIVERSITY.

Использование **NULL**-значений также создает определенные проблемы для рассматриваемых операторов.

Рассмотрим в качестве примера две реализации запроса “Найти все данные об университетах, рейтинг которых меньше рейтинга любого университета в Москве”.

1)
```
SELECT *
    FROM UNIVERSITY
    WHERE RATING < ANY
       (SELECT RATING
        FROM UNIVERSITY
        WHERE CITY = 'Москва');
```

2)
```
SELECT *
    FROM UNIVERSITY A
    WHERE NOT EXISTS
       (SELECT *
        FROM UNIVERSITY B
        WHERE A.RATING >= B.RATING
          AND B.CITY = 'Москва');
```

При отсутствии в таблице **NULL**-значений оба эти запроса ведут себя совершенно одинаково.

Пусть теперь в таблице UNIVERSITY есть строка с **NULL**-значениями в столбце RATING. В версии запроса с **ANY** в основном запросе, когда выбирается поле RATING с **NULL**, предикат принимает значение *неизвестно* и строка, так же как и в случае, когда результатом сравнения будет *ложь*, не включается в состав выходных данных. Во втором же варианте запроса, когда в основном запросе выбирается строка с **NULL** в поле RATING, предикат, используемый в подзапросе, примет значение *неизвестно*. Поэтому при выполнении подзапроса не будет получено ни одной строки, в результате чего оператор **NOT EXISTS** примет значение *истина*, и, следовательно, данная строка с **NULL**-значением попадет в выходные данные основного запроса. По смыслу этого запроса такой результат не является правильным, так как на самом деле рейтинг университета, описываемого данной строкой, может быть и больше рейтинга какого-либо московского университета (он просто неизвестен). Указанная проблема связана с тем, что значение **EXISTS** всегда принимает значения *истина* или *ложь*, и никогда — *неизвестно*. Это является доводом для использования в данном случае оператора **ANY** вместо **NOT EXISTS**.

УПРАЖНЕНИЕ

1. Ниже приведены два варианта запроса, выполняющего вывод количества студентов, имеющих только отличные оценки. Всегда ли эти запросы будут выдавать одинаковые результаты?

```
SELECT COUNT(DISTINCT STUDENT_ID)
   FROM EXAM_MARKS S
   WHERE NOT EXISTS
      (SELECT *
       FROM EXAM_MARKS
       WHERE STUDENT_ID=S.STUDENT_ID AND MARK<5);
```

```
SELECT COUNT (*)
   FROM
      (SELECT STUDENT_ID, MIN(MARK)
       FROM EXAM_MARKS
       GROUP BY STUDENT_ID
       HAVING MIN(MARK)=5);
```

2.14. Использование COUNT вместо EXISTS

При отсутствии **NULL**-значений оператор **EXISTS** может быть использован вместо **ANY** и **ALL**. Также вместо **EXISTS** и **NOT EXISTS** могут быть использованы те же самые подзапросы, но с использованием **COUNT**(*) в предложении **SELECT**. Например, запрос

```
SELECT *
   FROM UNIVERSITY A
   WHERE NOT EXISTS
      (SELECT *
       FROM UNIVERSITY B
       WHERE A.RATING >= B.RATING
       AND B.CITY = 'Москва');
```

может быть представлен и в следующем виде:

```
SELECT *
   FROM UNIVERSITY A
   WHERE 1 >
      (SELECT COUNT(*)
       FROM UNIVERSITY B
       WHERE A.RATING >= B.RATING
       AND B.CITY = 'Москва');
```

УПРАЖНЕНИЯ

1. Напишите запрос, выбирающий данные о названиях университетов, рейтинг которых равен или превосходит рейтинг ВГУ.
2. Напишите запрос с использованием **ANY** или **ALL**, выполняющий выборку данных о студентах, у которых в городе их постоянного местожительства нет университета.
3. Напишите запрос, выбирающий из таблицы EXAM_MARKS названия предметов обучения, для которых все оценки (поле MARK) превышают любые оценки по предмету, имеющему идентификатор 105.
4. Напишите этот же запрос с использованием **MAX**.

2.15. Соединение таблиц. Оператор JOIN

Если в операторе **SELECT** после ключевого слова **FROM** указывается не одна, а две таблицы, то в результате выполнения запроса, в котором отсутствует предложение **WHERE**, каждая строка одной таблицы будет соединена с каждой строкой второй таблицы. Такая операция называется *декартовым произведением*, или *полным соединением* таблиц базы данных. Сама по себе эта операция не имеет практического значения, более того, при ошибочном использовании она может привести к неожиданным нештатным ситуациям, так как в этом случае в ответе на запрос количество записей будет равно произведению числа записей в соединяемых таблицах, т. е. может оказаться чрезвычайно большим. Соединение таблиц имеет смысл тогда, когда соединяются *не все* строки исходных таблиц, а только те, которые интересуют пользователя. Такое ограничение может быть осуществлено с помощью использования в запросе соответствующего условия в предложении **WHERE**. Таким образом, SQL позволяет выводить информацию из нескольких таблиц, связывая их по значениям определенных полей.

Например, если необходимо получить фамилии студентов (таблица STUDENT) и для каждого студента — названия университетов (таблица UNIVERSITY), расположенных в городе, где живет студент, то необходимо получить все комбинации записей о студентах и университетах в обеих таблицах, в которых значение поля CITY совпадает. Это можно сделать с помощью следующего запроса:

```
SELECT STUDENT.SURNAME, UNIVERSITY.UNIV_NAME,
       STUDENT.CITY
  FROM STUDENT, UNIVERSITY
  WHERE STUDENT.CITY = UNIVERSITY.CITY;
```

Соединение, использующее предикаты, основанные на равенствах, называется *эквисоединением*. Рассмотренный пример соединения таблиц относятся к виду так называемого *внутреннего* (**INNER**) *соединения*. При таком типе соединения соединяются только те строки таблиц, для которых является истинным предикат, задаваемый в предложении **ON** выполняемого запроса.

Приведенный выше запрос может быть записан иначе, с использованием ключевого слова **JOIN**:

```
SELECT STUDENT.SURNAME, UNIVERSITY.UNIV_NAME,
       STUDENT.CITY
  FROM STUDENT INNER JOIN UNIVERSITY
  ON STUDENT.CITY = UNIVERSITY.CITY;
```

Ключевое слово **INNER** в запросе может быть опущено, так как эта опция в операторе **JOIN** действует по умолчанию.

УПРАЖНЕНИЯ

1. Напишите запрос для получения списка предметов вместе с фамилиями студентов, изучающих их на соответствующем курсе.
2. Напишите запрос, выполняющий вывод имен и фамилий студентов, имеющих весь набор положительных (тройки, четверки и пятерки) оценок.

2.15.1. Операции соединения таблиц посредством ссылочной целостности. Информация в таблицах STUDENT и EXAM_MARKS уже связана посредством поля STUDENT_ID. В таблице STUDENT поле STUDENT_ID является первичным ключом, а в таблице EXAM_MARKS — ссылающимся на него внешним ключом. Состояние связанных таким образом таблиц называется состоянием ссылочной целостности. В данном случае ссылочная целостность этих таблиц подразумевает, что *каждому* значению поля STUDENT_ID в таблице EXAM_MARKS *обязательно* соответствует *такое же значение* поля STUDENT_ID в таблице STUDENT. Другими словами, в таблице EXAM_MARKS не может быть записей, имеющих идентификаторы студентов, которых нет в таблице STUDENT. Стандартное применение операции соединения состоит в извлечении данных в терминах этой связи.

Чтобы получить список фамилий студентов с полученными ими оценками и идентификаторами предметов, можно использовать следующий запрос:

```
SELECT SURNAME, MARK, SUBJ_ID
   FROM STUDENT, EXAM_MARKS
   WHERE STUDENT.STUDENT_ID =
         EXAM_MARKS.STUDENT_ID;
```

Тот же самый результат может быть получен при использовании в запросе для задания операции соединения таблиц ключевого слова **JOIN**. Запрос с оператором **JOIN** выглядит следующим образом:

```
SELECT SURNAME, MARK
   FROM STUDENT JOIN EXAM_MARKS
   ON STUDENT.STUDENT_ID = EXAM_MARKS.STUDENT_ID;
```

Для такого рода запросов, когда соединение таблиц осуществляется по одноименным столбцам, можно использовать так называемое естественное соединение, задаваемое ключевым словом **NATURAL**. В этом случае в запросе не указывается предложение **ON** условия отбора записей. Приведенный выше запрос будет выглядеть следующим образом:

```
SELECT SURNAME, MARK
   FROM STUDENT NATURAL JOIN EXAM_MARKS;
```

Хотя выше речь шла о соединении двух таблиц, можно сформировать запросы путем соединения более чем двух таблиц.

Пусть требуется найти фамилии всех студентов, получивших неудовлетворительную оценку, вместе с названиями предметов обучения, по которым получена эта оценка.

```
SELECT SUBJ_NAME, SURNAME, MARK
   FROM STUDENT, SUBJECT, EXAM_MARKS
   WHERE STUDENT.STUDENT_ID = EXAM_MARKS.STUDENT_ID
      AND SUBJECT.SUBJ_ID = EXAM_MARKS.SUBJ_ID
      AND EXAM_MARKS.MARK = 2;
```

То же самое с использованием оператора **JOIN**:

```
SELECT SUBJ_NAME, SURNAME, MARK
   FROM STUDENT JOIN EXAM_MARKS
   ON STUDENT.STUDENT_ID = EXAM_MARKS.STUDENT_ID
   JOIN SUBJECT
   ON SUBJECT.SUBJ_ID = EXAM_MARKS.SUBJ_ID
   WHERE MARK = 2;
```

УПРАЖНЕНИЯ

1. Напишите запрос для получения списка университетов с указанием количества студентов, обучающихся на каждом курсе.
2. Напишите запрос для получения списка преподавателей с указанием их учебных предметов.
3. Напишите запрос для получения списка преподавателей с указанием нагрузки (суммарного количества часов) в каждом семестре.
4. Напишите запрос для получения списка университетов вместе с названиями преподаваемых в них предметов.
5. Напишите запрос для получения списка университетов с указанием суммарного количества аудиторных часов в каждом семестре.
6. Напишите запрос для получения списка университетов с указанием суммарного количества часов, отводимых на изучение каждого предмета.
7. Напишите запрос для получения списка преподавателей с указанием суммарного количества часов, отведенных для обучения каждому из предметов.
8. Напишите запрос для сортировки списка университетов по значениям максимальной стипендии, выплачиваемой студентам.
9. Напишите запрос для получения списка университетов вместе с фамилиями самых молодых студентов, обучаемых в них.
10. Напишите запрос для получения списка университетов вместе с фамилиями студентов, получающих максимальную для каждого университета стипендию.
11. Напишите запрос для получения списка студентов вместе с названиями предметов и оценками, полученными по каждому предмету на экзаменах.

12. Напишите запрос для получения списка предметов вместе с фамилиями студентов, получивших по данному предмету максимальную оценку.
13. Напишите запрос для получения списка предметов вместе с фамилиями студентов, сдававших экзамен по данному предмету последними.
14. Напишите запрос для получения списка предметов вместе с фамилиями студентов, первыми сдавших экзамен по данному предмету.
15. Напишите запрос для получения списка преподавателей, преподающих более одного предмета.
16. Напишите запрос для получения списка преподавателей, преподающих только один предмет.
17. Напишите запрос для получения списка студентов, сдававших экзамены по какому-либо из предметов более одного раза.
18. Напишите запрос для получения списка университетов вместе с фамилиями студентов, получивших хотя бы одну неудовлетворительную оценку.
19. Напишите запрос, выполняющий вывод имен и фамилий студентов, получивших хотя бы одну отличную оценку.
20. Напишите запрос, выполняющий вывод данных о предметах обучения, которые преподает Колесников.
21. Напишите запрос, выполняющий вывод имен и фамилий преподавателей, проводящих занятия на первом курсе.
22. Напишите запрос, выполняющий вывод имен и фамилий преподавателей, проводящих занятия в двух и более семестрах.
23. Напишите запрос, выполняющий вывод наименований предметов обучения, читаемых двумя или более преподавателями.
24. Напишите запрос, выполняющий вывод количества часов занятий, проводимых преподавателем Лагутиным.
25. Напишите запрос, выполняющий вывод фамилий преподавателей, учебная нагрузка которых (количество учебных часов) превышает нагрузку преподавателя Николаева.
26. Напишите запрос, выполняющий вывод фамилий преподавателей университетов с рейтингом, меньшим 200.
27. Напишите запрос, выполняющий вывод общего количества учебных часов занятий, проводимых для студентов первого курса ВГУ.
28. Напишите запрос, выполняющий вывод среднего количества учебных часов предметов обучения, преподаваемых студентам второго курса ВГУ.
29. Напишите запрос, выполняющий вывод списка фамилий студентов, имеющих две или более отличных оценок в каждом семестре.
30. Приведите как можно больше формулировок запроса “Получить фамилии студентов, сдававших экзамен по информатике”.

31. Приведите как можно больше формулировок запроса “Получить фамилии преподавателей, преподающих информатику”.

2.15.2. Внешнее соединение таблиц. Как отмечалось ранее, при использовании *внутреннего* (**INNER**) соединения таблиц соединяются только те их строки, в которых совпадают значения полей, задаваемые в предложении **WHERE** запроса. Однако во многих случаях это может привести к нежелательной потере информации. Рассмотрим еще раз приведенный выше пример запроса на выборку списка фамилий студентов с полученными ими оценками и идентификаторами предметов. При использовании, как это было сделано в рассматриваемом примере, внутреннего соединения в результат запроса не попадут студенты, которые еще не сдавали экзамены и которые, следовательно, отсутствуют в таблице EXAM_MARKS. Если же необходимо иметь записи об этих студентах в выдаваемом запросом списке, то можно присоединить сведения о студентах, не сдававших экзамен, путем использования оператора **UNION** с соответствующим запросом. Например, следующим образом:

```
SELECT SURNAME, CAST MARK AS CHAR(1),
                CAST SUBJ_ID AS CHAR(10)
   FROM STUDENT, EXAM_MARKS
   WHERE STUDENT.STUDENT_ID = EXAM_MARKS.STUDENT_ID
UNION
SELECT SURNAME, CAST NULL AS CHAR(1),
                CAST NULL AS CHAR(10)
   FROM STUDENT
   WHERE NOT EXIST
      (SELECT *
       FROM EXAM_MARKS
       WHERE STUDENT.STUDENT_ID =
             EXAM_MARKS.STUDENT_ID);
```

(здесь функция преобразования типов **CAST** используется для обеспечения совместимости типов полей объединяемых запросов).

Нужный результат, однако, может быть получен и путем использования оператора *внешнего соединения*, точнее, одной из его разновидностей — *левого внешнего соединения* **LEFT OUTER JOIN**. В этом случае запрос будет выглядеть следующим образом:

```
SELECT SURNAME, MARK
   FROM STUDENT LEFT OUTER JOIN EXAM_MARKS
   ON STUDENT.STUDENT_ID = EXAM_MARKS.STUDENT_ID;
```

При использовании *левого* соединения расширение выводимой таблицы осуществляется за счет записей входной таблицы, имя которой указано *слева* от оператора **JOIN**.

Следует заметить, что в СУБД Oracle для обозначения внешних соединений наряду со стандартной может использоваться и другая но-

тация. Например, приведенный выше запрос может иметь следующий вид:

```
SELECT SURNAME, MARK, SUBJ_ID
   FROM STUDENT, EXAM_MARKS
   WHERE STUDENT.STUDENT_ID =
         EXAM_MARKS.STUDENT_ID(+);
```

Знак (**+**) ставится у той таблицы, которая дополняется записями с **NULL**-значениями, чтобы при соединении таблиц в выходное отношение попали и те записи другой таблицы, для которых в таблице со знаком (**+**) не находится строк с соответствующими значениями атрибутов, используемых для соединения, т. е. для *левого* внешнего соединения в запросе Oracle-SQL указатель (**+**) ставится у *правой* таблицы.

Приведенный выше запрос может быть реализован и с применением *правого внешнего соединения*. Он будет иметь следующий вид:

```
SELECT SURNAME, MARK
   FROM EXAM_MARKS RIGHT OUTER JOIN STUDENT
   ON EXAM_MARKS.STUDENT_ID = STUDENT.STUDENT_ID;
```

Здесь таблица **STUDENT**, за счет записей которой осуществляется расширение выводимой таблицы, стоит справа от оператора **JOIN**.

В нотации, допустимой в СУБД Oracle, этот запрос может выглядеть следующим образом:

```
SELECT SURNAME, MARK, SUBJ_ID
   FROM STUDENT, EXAM_MARKS
   WHERE EXAM_MARKS.STUDENT_ID(+) =
         STUDENT.STUDENT_ID;
```

Видно, что использование правого или левого внешнего соединения позволяет существенно упростить запрос, сделать его запись более компактной.

Иногда возникает необходимость включения в результат запроса записей из обеих (правой и левой) соединяемых таблиц, для которых не удовлетворяется условие соединения. Такое соединение называется *полным внешним соединением* и осуществляется указанием в запросе ключевых слов **FULL OUTER JOIN**.

УПРАЖНЕНИЯ

1. Напишите запрос, который выполняет вывод фамилий студентов, *сдававших* экзамены, вместе с идентификаторами каждого сданного ими предмета обучения.
2. Напишите запрос, который выполняет выборку фамилий *всех* студентов, с указанием для студентов, сдававших экзамены, идентификаторов сданных ими предметов обучения.

3. Напишите запрос, который выполняет вывод фамилий студентов, *сдававших* экзамены, вместе с наименованиями каждого сданного ими предмета обучения.
4. Напишите запрос на выдачу списка *всех* студентов. Для студентов, сдававших экзамены, укажите названия соответствующих предметов обучения.
5. Напишите запрос на выдачу названий всех предметов, по которым студенты получили только хорошие (4 и 5) оценки. В выходных данных должны быть приведены фамилии студентов, названия предметов и оценка.
6. Напишите запрос, который выполняет вывод списка университетов с рейтингом, превышающим 300, вместе со значением максимального размера стипендии, получаемой студентами в этих университетах.
7. Напишите запрос на выдачу списка студентов (в алфавитном порядке фамилий) вместе со значением рейтинга университета, где каждый из них учится, включив в список и тех студентов, место учебы которых в базе данных не указано.
8. Напишите запрос для получения списка всех студентов вместе с названиями университетов, в которых они учатся. Отдельным запросом получите записи, расширяющие данный список по сравнению с тем, который был бы получен внутренним соединением.
9. Напишите запрос для получения списка всех университетов вместе с фамилиями студентов, в них обучающихся. Отдельным запросом получите записи, расширяющие данный список по сравнению с тем, который был бы получен внутренним соединением.
10. Напишите запрос для получения списка всех университетов вместе с фамилиями преподавателей, в них работающих. Отдельным запросом получите записи, расширяющие данный список по сравнению с тем, который был бы получен внутренним соединением.
11. Напишите запрос для получения списка всех преподавателей вместе с университетами, в которых они работают. Есть ли отличие списка от того, который был бы получен внутренним соединением? Подтвердите отдельным запросом ваш вывод.
12. Напишите запрос для получения списка всех студентов и оценок, полученных ими на экзаменах. Отдельным запросом получите записи, расширяющие данный список по сравнению с тем, который был бы получен внутренним соединением.
13. Напишите запрос для получения списка всех экзаменационных оценок вместе с фамилиями получивших их студентов. Есть ли отличие списка от того, который был бы получен внутренним соединением? Подтвердите отдельным запросом ваш вывод.
14. Напишите запрос для получения полного списка предметов с соответствующими экзаменационными оценками. Есть ли отличие списка от того, который был бы получен внутренним соединением? Подтвердите отдельным запросом ваш вывод.

15. Напишите запрос для получения полного списка оценок вместе с названиями предметов, по которым они получены. Есть ли отличие списка от того, который был бы получен внутренним соединением? Подтвердите отдельным запросом ваш вывод.

2.15.3. Использование псевдонимов при соединении копий одной таблицы. Часто при получении информации из таблиц базы данных необходимо осуществлять соединение таблицы с ее же копией. Например, это требуется в случае, когда требуется найти фамилии студентов, имеющих одинаковые имена. При соединении таблицы с ее же копией вводят псевдонимы (алиасы) таблицы. Запрос для поиска фамилий студентов, имеющих одинаковые имена, выглядит следующим образом:

```
SELECT FIRST.SURNAME, SECOND.SURNAME
   FROM STUDENT FIRST, STUDENT SECOND
   WHERE FIRST.NAME = SECOND.NAME;
```

В этом запросе введены два псевдонима для одной таблицы STUDENT, что позволяет корректно задать выражение, связывающее две копии таблицы. Чтобы исключить повторения строк в выводимом результате запроса из-за повторного сравнения одной и той же пары студентов, необходимо задать порядок следования для двух значений так, чтобы одно значение было меньше, чем другое, что делает предикат асимметричным.

```
SELECT FIRST.SURNAME, SECOND.SURNAME
   FROM STUDENT FIRST, STUDENT SECOND
   WHERE FIRST.NAME = SECOND.NAME
      AND FIRST.SURNAME < SECOND.SURNAME;
```

УПРАЖНЕНИЯ

1. Напишите запрос, выполняющий вывод списка всех пар фамилий студентов, проживающих в одном городе. При этом не включать в список комбинации фамилий студентов самих с собой (т. е. комбинации типа “Иванов–Иванов”) и комбинации фамилий студентов, отличающиеся порядком следования (т. е. из двух комбинаций типа “Иванов–Петров” и “Петров–Иванов” включать только одну).

2. Напишите запрос, выполняющий вывод списка всех пар названий университетов, расположенных в одном городе, не включая в список комбинации названий университетов самих с собой и пары названий университетов, отличающиеся порядком следования.

3. Напишите запрос, который позволяет получить названия университетов с рейтингом, не меньшим рейтинга ВГУ, и городов, в которых они расположены.

4. Напишите запрос, выполняющий выборку идентификаторов студентов, имеющих такие же оценки, что и студент с идентификатором 12.
5. Напишите запрос, выполняющий выборку всех пар идентификаторов преподавателей, ведущих один и тот же предмет обучения.

2.16. Оператор объединения UNION

Оператор **UNION** используется для объединения выходных данных двух или более SQL-запросов в единое множество строк и столбцов. Например, для того, чтобы получить в одной таблице фамилии и идентификаторы студентов и преподавателей из Москвы, можно использовать следующий запрос:

```
SELECT 'Студент', SURNAME, STUDENT_ID
   FROM STUDENT
   WHERE CITY = 'Москва'
UNION
SELECT 'Преподаватель', SURNAME, LECTURER_ID
   FROM LECTURER
   WHERE CITY = 'Москва';
```

Обратите внимание на то, что символом ";" (точка с запятой) оканчивается только последний запрос. Отсутствие этого символа в конце **SELECT**-запроса означает, что этот запрос, как и следующий за ним, является частью общего запроса с **UNION**.

Использование оператора **UNION** возможно только при объединении запросов, соответствующие столбцы которых *совместимы по объединению*. Совместимость по объединению означает, что столбцы, как минимум, должны относиться к одному типу. При этом если говорить о таких конкретных характеристиках типов, как, например, количество символов для полей символьного типа, размер и точность числовых полей, то возможность оператора **UNION** зависит от конкретной реализации СУБД. В одних системах задание оператора **UNION** требует полного совпадения характеристик типов столбцов, а в других возможно неявное приведение отличающихся характеристик. В примерах, приводимых в данном пособии, подразумевается, что такое неявное приведение имеет место. Аналогично, в зависимости от реализации СУБД, оператор **UNION** может иметь ограничение, связанное с использованием **NULL**-значений: если **NULL**-значения запрещены для столбца хотя бы одного любого подзапроса объединения, то они должны быть запрещены и для всех соответствующих столбцов в других подзапросах объединения.

2.16.1. Устранение дублирования в UNION. В отличие от обычных запросов, **UNION** автоматически исключает из выходных данных дубликаты строк: например, в запросе

```
SELECT CITY
   FROM STUDENT
UNION
SELECT CITY
   FROM LECTURER;
```

совпадающие наименования городов будут исключены.

Если все же необходимо в каждом запросе вывести все строки независимо от того, имеются ли такие же строки в других объединяемых запросах, то следует использовать во множественном запросе конструкцию с оператором **UNION ALL**. Так, в запросе

```
SELECT CITY
   FROM STUDENT
UNION ALL
SELECT CITY
   FROM LECTURER;
```

дубликаты значений городов, выводимые второй частью запроса, не будут исключаться.

Приведем еще один пример использования оператора **UNION**. Пусть необходимо составить отчет, содержащий для каждой даты сдачи экзаменов сведения по каждому студенту, получившему максимальную или минимальную оценку.

```
SELECT 'макс оц', A.STUDENT_ID, SURNAME,
       MARK, EXAM_DATE
   FROM STUDENT A, EXAM_MARKS B
   WHERE (A.STUDENT_ID = B.STUDENT_ID
      AND B.MARK =
         (SELECT MAX(MARK)
          FROM EXAM_MARKS C
          WHERE C.EXAM_DATE = B.EXAM_DATE))
UNION ALL
SELECT 'мин оц ', A.STUDENT_ID, SURNAME,
       MARK, EXAM_DATE
   FROM STUDENT A, EXAM_MARKS B
      WHERE (A.STUDENT_ID = B.STUDENT_ID
         AND B.MARK =
            (SELECT MIN(MARK)
             FROM EXAM_MARKS C
             WHERE C.EXAM_DATE = B.EXAM_DATE));
```

Для различения строк, выводимых первой и второй частями запроса, в них вставлены текстовые константы `'макс оц'` и `'мин оц'`.

В приведенном запросе агрегирующие функции используются в подзапросах. Можно предложить альтернативный вариант формулиров-

ки запроса, возможно, более рациональный с точки зрения времени, затрачиваемого на выполнение запроса:

```
SELECT 'макс оц', A.STUDENT_ID, SURNAME,
                  E.MARK, E.EXAM_DATE
   FROM STUDENT A,
      (SELECT B.STUDENT_ID, B.MARK, B.EXAM_DATE
        FROM EXAM_MARKS B,
         (SELECT MAX(MARK) AS MAX_MARK, C.EXAM_DATE
           FROM EXAM_MARKS C
         GROUP BY C.EXAM_DATE) D
        WHERE B.EXAM_DATE=D.EXAM_DATE
          AND B.MARK=MAX_MARK) E
   WHERE A.STUDENT_ID=E.STUDENT_ID
UNION ALL
SELECT 'мин оц ', A.STUDENT_ID, SURNAME,
                  E.MARK, E.EXAM_DATE
   FROM STUDENT A,
      (SELECT B.STUDENT_ID, B.MARK, B.EXAM_DATE
        FROM EXAM_MARKS B,
         (SELECT MIN(MARK) AS MIN_MARK, C.EXAM_DATE
           FROM EXAM_MARKS C
         GROUP BY C.EXAM_DATE) D
        WHERE B.EXAM_DATE=D.EXAM_DATE
      AND B.MARK=MIN_MARK) E
   WHERE A.STUDENT_ID=E.STUDENT_ID;
```

2.16.2. Использование UNION с ORDER BY. Предложение **ORDER BY** применяется для упорядочения выходных данных объединения запросов так же, как и для отдельных запросов. Последний пример, при необходимости упорядочения выходных данных запроса по фамилиям студентов и датам экзаменов, может выглядеть так:

```
SELECT 'макс оц', A.STUDENT_ID, SURNAME,
                  E.MARK, E.EXAM_DATE
   FROM STUDENT A,
      (SELECT B.STUDENT_ID, B.MARK, B.EXAM_DATE
        FROM EXAM_MARKS B,
         (SELECT MAX(MARK) AS MAX_MARK, C.EXAM_DATE
           FROM EXAM_MARKS C
           GROUP BY C.EXAM_DATE) D
        WHERE B.EXAM_DATE=D.EXAM_DATE
            AND B.MARK=MAX_MARK) E
   WHERE A.STUDENT_ID=E.STUDENT_ID
UNION ALL
SELECT 'мин оц ', A.STUDENT_ID, SURNAME,
                  E.MARK, E.EXAM_DATE
```

```
   FROM STUDENT A,
      (SELECT B.STUDENT_ID, B.MARK, B.EXAM_DATE
       FROM EXAM_MARKS B,
         (SELECT MIN(MARK) AS MIN_MARK, C.EXAM_DATE
          FROM EXAM_MARKS C
          GROUP BY C.EXAM_DATE) D
       WHERE B.EXAM_DATE=D.EXAM_DATE
      AND B.MARK=MIN_MARK) E
   WHERE A.STUDENT_ID=E.STUDENT_ID
ORDER BY SURNAME, E.EXAM_DATE;
```

Часто полезна операция объединения двух запросов, в которой второй запрос выбирает строки, исключенные первым.

Рассмотрим пример. Пусть в таблице STUDENT имеются записи о студентах без указания идентификатора университета. Требуется составить список студентов с указанием наименования университета для тех студентов, у которых эти данные есть, но при этом не отбрасывая и студентов, у которых университет не указан. Можно получить желаемые сведения, сформировав объединение двух запросов, один из которых выполняет выборку студентов с названиями их университетов, а второй выбирает студентов с **NULL**-значениями в поле UNIV_ID. В данном случае оказывается полезной возможность вставки в запрос констант, в нашем случае текстовой константы 'неизвестен', чтобы отметить в списке тех студентов, у которых отсутствует информация об университете.

```
SELECT SURNAME, NAME, UNIV_NAME
   FROM STUDENT, UNIVERSITY
   WHERE STUDENT.UNIV_ID = UNIVERSITY.UNIV_ID
UNION
SELECT SURNAME, NAME, 'неизвестен'
   FROM STUDENT
   WHERE UNIV_ID IS NULL
ORDER BY 1;
```

Для совместимости столбцов объединяемых запросов константу 'неизвестен' во втором запросе следует дополнить пробелами так, чтобы ее длина соответствовала длине поля UNIV_NAME или использовать для согласования типов функцию **CAST**. В некоторых СУБД согласование типов поля и замещающей его текстовой константы осуществляется автоматически.

УПРАЖНЕНИЯ

1. Создайте объединение двух запросов, которые выдают значения полей UNIV_NAME, CITY, RATING для всех университетов. Те из них, у которых рейтинг равен или выше 300, должны иметь комментарий 'высокий', все остальные — 'низкий'.

2. Напишите команду, которая выдает список фамилий студентов с комментарием: `'успевает'` у студентов, имеющих все положительные оценки; `'не успевает'` для сдававших экзамены, но имеющих хотя бы одну неудовлетворительную оценку; `'не сдавал'` — для всех остальных. В выводимом результате фамилии студентов упорядочите по алфавиту.
3. Выведите объединенный список студентов и преподавателей, живущих в Москве, с соответствующими комментариями `'студент'` или `'преподаватель'`.
4. Выведите объединенный список студентов и преподавателей ВГУ с соответствующими комментариями `'студент'` или `'преподаватель'`.
5. Для каждого города выведити названия университетов с минимальным и максимальным для данного города рейтингом. Пометьте строки списка словами `'min'` и `'max'`, поместив их в дополнительном столбце.
6. Для каждого курса выведите фамилии студентов, получающих минимальные и максимальные на их курсе стипендии. Пометьте строки списка словами `'min'` и `'max'`, поместив их в дополнительном столбце.
7. Для каждого курса выведите фамилии самого старшего и самого младшего студентов. Пометьте строки списка словами `'младший'` и `'старший'`, поместив их в дополнительном столбце.
8. Напишите запрос для получения полного списка университетов вместе с фамилиями студентов, которые в них учатся. Для университетов, не имеющих студентов, поместите в список фразу `'Студентов нет'`.
9. Напишите запрос для получения полного списка университетов вместе с фамилиями преподавателей, в них работающих. Для университетов, не имеющих преподавателей, поместите в список фразу `'Преподавателей нет'`.
10. Выведити полный список студентов вместе с оценками, полученными ими на экзаменах. Для студентов, не сдававших экзамены, в поле оценки поместите 0.

Глава 3

МАНИПУЛИРОВАНИЕ ДАННЫМИ

3.1. Операторы манипулирования данными

В SQL для выполнения операций ввода данных в таблицу, их изменения и удаления предназначены три оператора языка манипулирования данными (DML). Это операторы **INSERT** (ВСТАВИТЬ), **UPDATE** (ОБНОВИТЬ), **DELETE** (УДАЛИТЬ).

Оператор **INSERT** осуществляет ***вставку*** в таблицу новой строки. В простейшем случае он имеет следующий вид:

```
INSERT INTO <имя таблицы>
   VALUES (<значение>, <значение>, ...);
```

При такой записи указанные в скобках после ключевого слова **VALUES** значения вводятся в поля добавленной в таблицу новой строки в том порядке, в котором соответствующие столбцы указаны при создании таблицы, т. е. в операторе **CREATE TABLE**.

Например, ввод новой строки в таблицу STUDENT может быть осуществлен следующим образом:

```
INSERT INTO STUDENT
   VALUES (101, 'Иванов', 'Александр', 200, 3,
           'Москва', '6/10/1979', 15);
```

Чтобы такой оператор мог быть выполнен, таблица с указанным в нем именем (STUDENT) должна быть предварительно определена (создана) оператором **CREATE TABLE**. Если в какое-либо поле необходимо вставить **NULL**-значение, то оно вводится как обычное значение:

```
INSERT INTO STUDENT
   VALUES (101, 'Иванов', NULL, 200, 3,
           'Москва', '6/10/1979', 15);
```

В случаях, когда необходимо ввести значения полей в порядке, отличном от порядка столбцов, заданного оператором **CREATE TABLE**,

или если требуется ввести значения не во все столбцы, то следует использовать следующую форму оператора **INSERT**:

```
INSERT INTO STUDENT (STUDENT_ID, CITY, SURNAME,
                     NAME)
   VALUES (101, 'Москва', 'Иванов', 'Саша');
```

Столбцам, наименования которых не указаны в списке, приведенном в скобках, автоматически присваивается значение по умолчанию, если оно назначено при определении таблицы (оператор **CREATE TABLE**), либо значение **NULL**.

С помощью оператора **INSERT** можно извлечь значение из одной таблицы и разместить его в другой, к примеру, запросом следующего вида:

```
INSERT INTO STUDENT1
   SELECT *
   FROM STUDENT
   WHERE CITY = 'Москва';
```

При этом таблица STUDENT1 должна быть предварительно создана оператором **CREATE TABLE** (раздел 4.1) и иметь структуру, идентичную таблице STUDENT.

Удаление строк из таблицы осуществляется с помощью оператора **DELETE**.

Следующее выражение удаляет все строки таблицы EXAM_MARKS1:

```
DELETE FROM EXAM_MARKS1;
```

В результате таблица становится пустой (после этого она может быть удалена командой **DROP TABLE**).

Для удаления из таблицы сразу нескольких строк, удовлетворяющих некоторому условию, можно воспользоваться предложением **WHERE**, например,

```
DELETE FROM EXAM_MARKS1
   WHERE STUDENT_ID = 103;
```

Можно удалить группу строк

```
DELETE FROM STUDENT1
   WHERE CITY = 'Москва';
```

Оператор **UPDATE** позволяет ***изменять***, т. е. обновлять, значения некоторых или всех полей в существующей строке или строках таблицы. Например, чтобы для всех университетов, сведения о которых находятся в таблице UNIVERSITY1, изменить рейтинг на значение 200, можно использовать конструкцию:

```
UPDATE UNIVERSITY1
   SET RATING = 200;
```

Если требуется изменить значения полей в конкретных строках таблицы, то в операторе **UPDATE** можно использовать предикат, задаваемый в предложении **WHERE**.

```
UPDATE UNIVERSITY1
   SET RATING = 200
   WHERE CITY = 'Москва';
```

В результате выполнения этого запроса будет изменен рейтинг только у университетов, расположенных в Москве.

Оператор **UPDATE** позволяет изменять не только один, но и множество столбцов. Для указания конкретных столбцов, значения которых должны быть модифицированы, используется предложение **SET**.

Например, пусть наименование предмета обучения 'Математика' (для него SUBJ_ID = 43) должно быть заменено на название 'Высшая математика', при этом идентификационный номер необходимо сохранить, но в соответствующие поля строки таблицы ввести новые данные об этом предмете обучения. Запрос будет выглядеть следующим образом:

```
UPDATE SUBJECT1
SET SUBJ_NAME = 'Высшая математика', HOUR = 36,
    SEMESTER = 1
WHERE SUBJ_ID = 43;
```

В предложении **SET** оператора **UPDATE** можно использовать скалярные выражения, указывающие способ изменения значений поля, в которые могут входить значения изменяемого и других полей.

```
UPDATE UNIVERSITY1
   SET RATING = RATING*2;
```

Например, чтобы удвоить значения поля STIPEND в таблице STUDENT1 для студентов из Москвы, можно использовать запрос

```
UPDATE STUDENT1
   SET STIPEND = STIPEND*2
   WHERE CITY = 'Москва';
```

УПРАЖНЕНИЯ

1. Напишите команду, которая вводит в таблицу SUBJECT строку для нового предмета обучения со следующими значениями полей: SEMESTER = 4; SUBJ_NAME = 'Алгебра'; HOUR = 72; SUBJ_ID = 201.
2. Введите запись для нового студента, которого зовут Орлов Николай, обучающегося на первом курсе ВГУ, живущего в Воронеже, сведения о дате рождения и размере стипендии неизвестны.
3. Напишите команду, удаляющую из таблицы EXAM_MARKS записи обо всех оценках студента, идентификатор которого равен 100.

4. Напишите команду, которая увеличивает на 5 значение рейтинга всех университетов, расположенных в Санкт-Петербурге.
5. Измените в таблице значение города, в котором проживает студент Иванов, на 'Воронеж'.

3.2. Использование подзапросов в INSERT

Применение оператора **INSERT** с подзапросом позволяет загружать сразу несколько строк в одну таблицу, используя информацию из другой таблицы. В то время как оператор **INSERT**, использующий **VALUES**, добавляет только одну строку, **INSERT** с подзапросом добавляет в таблицу столько строк, сколько подзапрос извлекает из другой таблицы. При этом количество и тип возвращаемых подзапросом столбцов должно соответствовать количеству и типу столбцов таблицы, в которую вставляются данные.

Например, пусть таблица STUDENT1 имеет структуру, полностью совпадающую со структурой таблицы STUDENT. В разделе 3.1 приведен запрос, позволяющий заполнить таблицу STUDENT1 записями из таблицы STUDENT обо всех студентах из Москвы:

```
INSERT INTO STUDENT1
   SELECT *
   FROM STUDENT
   WHERE CITY = 'Москва';
```

Для того же, чтобы добавить в таблицу STUDENT1 сведения обо всех студентах, которые *учатся* в Москве, можно использовать в предложении **WHERE** соответствующий подзапрос. Например,

```
INSERT INTO STUDENT1
   SELECT *
   FROM STUDENT
   WHERE UNIV_ID IN
      (SELECT UNIV_ID
       FROM UNIVERSITY
       WHERE CITY = 'Москва');
```

3.2.1. Использование подзапросов, основанных на таблицах внешних запросов. Предположим, существует таблица SSTUD, в которой хранятся сведения о студентах, обучающихся в том же городе, в котором они живут. Можно заполнить эту таблицу данными из таблицы STUDENT, используя связанные подзапросы, следующим образом:

```
INSERT INTO SSTUD
   SELECT *
   FROM STUDENT A
   WHERE CITY IN
      (SELECT CITY
```

```
      FROM UNIVERSITY B
      WHERE A.UNIV_ID = B.UNIV_ID);
```

Предположим, что требуется выбрать список студентов, имеющих максимальный балл на каждый день сдачи экзаменов, и разместить его в другой таблице с именем **EXAM**. Это можно осуществить с помощью запроса

```
INSERT INTO EXAM
   SELECT EXAM_ID, STUDENT_ID, SUBJ_ID, MARK,
          EXAM_DATE
     FROM EXAM_MARKS A
       WHERE MARK =
       (SELECT MAX(MARK)
        FROM EXAM_MARKS B
        WHERE A.EXAM_DATE = B.EXAM_DATE);
```

3.2.2. Использование подзапросов с DELETE. Пусть филиал университета в Нью-Васюках ликвидирован, и требуется удалить из таблицы STUDENT записи о студентах, которые там учились. Эту операцию можно выполнить с помощью запроса

```
DELETE
   FROM STUDENT
   WHERE UNIV_ID IN
      (SELECT UNIV_ID
       FROM UNIVERSITY
       WHERE CITY = 'Нью-Васюки');
```

В предикате предложения **FROM** (подзапроса) нельзя ссылаться на таблицу, из которой осуществляется удаление. Однако можно ссылаться на текущую строку из таблицы, являющуюся кандидатом на удаление, т. е. на строку, которая в настоящее время проверяется в основном предикате.

```
DELETE
   FROM STUDENT
   WHERE EXISTS
      (SELECT *
       FROM UNIVERSITY
       WHERE RATING = 401
         AND STUDENT.UNIV_ID = UNIVERSITY.UNIV_ID);
```

Часть **AND** предиката внутреннего запроса ссылается на таблицу STUDENT. Команда удаляет данные о студентах, которые учатся в университетах с рейтингом 401. Существуют и другие способы решения этой задачи:

```
DELETE
   FROM STUDENT
```

```
   WHERE 401 IN
      (SELECT RATING
       FROM UNIVERSITY
       WHERE STUDENT.UNIV_ID = UNIVERSITY.UNIV_ID);
```

Пусть для каждого дня сдачи экзаменов нужно найти наименьшее значение выставленной оценки и удалить из таблицы сведения о студенте, который получил эту оценку. Запрос будет иметь вид

```
DELETE
   FROM STUDENT
   WHERE STUDENT_ID IN
      (SELECT STUDENT_ID
       FROM EXAM_MARKS A
       WHERE MARK =
         (SELECT MIN(MARK)
          FROM EXAM_MARKS B
          WHERE A.EXAM_DATE = B.EXAM_DATE));
```

Так как столбец STUDENT_ID является первичным ключом, то удаляется единственная строка.

Если в какой-то день сдавался только один экзамен (т. е. получена только одна минимальная оценка) и по какой-либо причине запись, в которой находится эта оценка, требуется оставить, то решение будет иметь вид:

```
DELETE
   FROM STUDENT
   WHERE STUDENT_ID IN
      (SELECT STUDENT_ID
       FROM EXAM_MARKS A
       WHERE MARK =
         (SELECT MIN(MARK)
          FROM EXAM_MARKS B
          WHERE A.EXAM_DATE = B.EXAM_DATE
            AND 1 <
            (SELECT COUNT(SUBJ_ID)
             FROM EXAM_MARKS B
             WHERE A.EXAM_DATE = B.EXAM_DATE)));
```

3.2.3. Использование подзапросов с UPDATE. С помощью команды **UPDATE** можно применять подзапросы в любой форме, приемлемой для команды **DELETE**.

Например, используя связанные подзапросы, можно увеличить значение размера стипендии на 20 в записях студентов, сдавших экзамены на 4 и 5:

```
UPDATE STUDENT1
   SET STIPEND = STIPEND + 20
```

```
WHERE 4 <=
(SELECT MIN(MARK)
 FROM EXAM_MARKS
 WHERE EXAM_MARKS.STUDENT_ID =
       STUDENT1.STUDENT_ID);
```

Другой запрос — уменьшить величину стипендии на 20 всем студентам, получившим на экзамене минимальную оценку:

```
UPDATE STUDENT1
   SET STIPEND = STIPEND - 20
   WHERE STUDENT_ID IN
      (SELECT STUDENT_ID
       FROM EXAM_MARKS A
       WHERE MARK =
         (SELECT MIN(MARK)
          FROM EXAM_MARKS B
          WHERE A.EXAM_DATE = B.EXAM_DATE));
```

УПРАЖНЕНИЯ

1. Пусть существует таблица с именем STUDENT1, определения столбцов которой полностью совпадают с определениями столбцов таблицы STUDENT. Вставьте в эту таблицу сведения о студентах, успешно сдавших экзамены более чем по пяти предметам обучения.
2. Напишите команду, удаляющую из таблицы SUBJECT1 сведения о предметах обучения, по которым студентами не получено ни одной оценки.
3. Напишите запрос, увеличивающий размер стипендии на 20% всем студентам, у которых общая сумма баллов превышает значение 50.

Глава 4

СОЗДАНИЕ ОБЪЕКТОВ БАЗЫ ДАННЫХ

4.1. Создание таблиц базы данных

Создание объектов базы данных осуществляется с помощью операторов языка определения данных (DDL).

Таблицы базы данных создаются с помощью оператора **CREATE TABLE**. Эта команда создает пустую таблицу, т.е. таблицу, не имеющую строк. Значения в эту таблицу вводятся с помощью оператора **INSERT**. Оператор **CREATE TABLE** определяет имя таблицы и множество поименованных столбцов в указанном порядке. Для каждого столбца должны быть определены тип и размер. Каждая создаваемая таблица должна иметь по крайней мере один столбец. Упрощенный синтаксис оператора **CREATE TABLE** имеет следующий вид:

```
CREATE TABLE <имя таблицы>
        (<имя столбца><тип данных>[(<размер>)], ...);
```

Следующий пример показывает запрос, создающий таблицу STUDENT1:

```
CREATE TABLE STUDENT1
   (STUDENT_ID  INTEGER,
    SURNAME     VARCHAR(60),
    NAME        VARCHAR(60),
    STIPEND     DOUBLE,
    KURS        INTEGER,
    CITY        VARCHAR(60),
    BIRTHDAY    DATE,
    UNIV_ID     INTEGER);
```

4.2. Использование индексации для быстрого доступа к данным

Операции поиска-выборки (**SELECT**) данных из таблиц по значениям их полей могут быть существенно ускорены путем использования индексации данных. Индекс содержит упорядоченный (в алфавитном или числовом порядке) список содержимого столбца или группы столбцов в индексируемой таблице с идентификаторами соответствующих строк (**ROWID**). Для пользователей индексирование таблицы по тем или иным столбцам представляет собой способ *логического* упорядочения значений индексированных столбцов, позволяющего, в отличие от последовательного перебора строк, существенно повысить скорость доступа к конкретным строкам таблицы при выборках, использующих значения этих столбцов. Индексация позволяет находить содержащий индексированную строку блок данных, выполняя небольшое число обращений к внешнему устройству хранения данных.

При использовании индексации следует, однако, иметь в виду, что управление индексом существенно увеличивает время выполнения операций, связанных с обновлением данных (таких, как **INSERT** и **DELETE**), так как эти операции требуют перестройки индексов.

Индексы можно создавать как по одному, так и по множеству полей. Если указано более одного поля для создания единственного индекса, то данные упорядочиваются по значениям первого поля, по которому осуществляется индексирование. Внутри получившейся группы осуществляется упорядочивание по значениям второго поля, для получившихся в результате групп осуществляется упорядочивание по значениям третьего поля и т.д.

Синтаксис оператора создания индекса имеет следующий вид:

```
CREATE INDEX <имя индекса> ON <имя таблицы>
   (<имя столбца> [,<имя столбца>]...);
```

При этом таблица должна уже быть созданной и содержать столбцы, имена которых указаны в команде создания индекса. Имя индекса, определенное в команде, должно быть уникальным в базе данных. Будучи однажды созданным, индекс является невидимым для пользователя, все операции с ним осуществляет СУБД.

Пример.

Если таблица EXAM_MARKS часто используется для поиска оценки конкретного студента по значению поля STUDENT_ID, то следует создать индекс по этому полю:

```
CREATE INDEX STUDENT_ID_1
   ON EXAM_MARKS (STUDENT_ID);
```

Для удаления индекса (при этом обязательно требуется знать его имя) используется команда **DROP INDEX**, имеющая следующий синтаксис:

DROP INDEX <*имя индекса*>;

Удаление индекса не изменяет содержимого поля или полей, индекс которых удаляется.

4.3. Изменение существующей таблицы

Для модификации структуры и параметров существующей таблицы используется оператор **ALTER TABLE**. Упрощенный синтаксис оператора **ALTER TABLE** для добавления столбцов в таблицу имеет вид

ALTER TABLE <*имя таблицы*>
ADD (<*имя столбца*><*тип данных*><*размер*>);

При выполнении этого оператора для существующих в таблице строк добавляется новый столбец, в который заносится **NULL**-значение. Этот столбец становится последним в таблице. Можно добавлять несколько столбцов, в этом случае их определения в команде **ALTER TABLE** разделяются запятой.

Возможно изменение описания столбцов. Часто это связано с изменением размеров столбцов, добавлением или удалением ограничений, накладываемых на их значения. Синтаксис оператора в этом случае имеет вид

ALTER TABLE <*имя таблицы*>
MODIFY <*имя столбца*><*тип данных*><*размер/точность*>;

Следует иметь в виду, что модификация характеристик столбца может осуществляться не в любом случае, а с учетом следующих ограничений:

- изменение типа данных возможно только, если столбец пуст;
- для незаполненного столбца можно изменять размер/точность; для заполненного столбца размер/точность можно увеличить, но нельзя понизить;
- ограничение **NOT NULL** может быть установлено, если ни одно значение в столбце не содержит **NULL**; опцию **NOT NULL** всегда можно отменить;
- разрешается изменять значения, установленные по умолчанию.

4.4. Удаление таблицы

Синтаксис оператора, осуществляющего удаление таблицы, имеет следующий вид:

DROP TABLE <*имя таблицы*>;

УПРАЖНЕНИЯ

1. Напишите команду **CREATE TABLE** для создания таблицы LECTURER1.
2. Напишите команду **CREATE TABLE** для создания таблицы SUBJECT1.
3. Напишите команду **CREATE TABLE** для создания таблицы UNIVERSITY1.
4. Напишите команду **CREATE TABLE** для создания таблицы EXAM_MARKS1.
5. Напишите команду **CREATE TABLE** для создания таблицы SUBJ_LECT1.
6. Напишите команду, которая позволит быстро выбрать данные о студентах по курсам, на которых они учатся.
7. Создайте индекс, который позволит для каждого студента быстро осуществить поиск оценок, сгруппированных по датам.

4.5. Ограничения на множество допустимых значений данных

До сих пор рассматривалось только следующее ограничение: значения, вводимые в таблицу, должны иметь типы данных и размеры, совместимые с типами и размерами данных столбцов, в которые эти значения вводятся (как определено в команде **CREATE TABLE** или **ALTER TABLE**). Описание таблицы может быть дополнено более сложными ограничениями, накладываемыми на значения, которые могут быть вставлены в столбец или группу столбцов. Ограничения (**CONSTRAINTS**) являются частью определения таблицы.

При создании (изменении) таблицы могут быть определены ограничения на вводимые значения. В этом случае SQL будет отвергать любое из вводимых значений, не соответствующее заданному ограничению. Ограничения могут быть статическими, ограничивающими значения или диапазон значений, вставляемых в столбец (**CHECK**, **NOT NULL**). Они могут иметь связь со всеми значениями столбца, ограничивая новые строки значениями, которые не содержатся в столбцах или их наборах (уникальные значения, первичные ключи). Ограничения могут также определяться связью со значениями, находящимися в другой таблице, допуская, например, вставку в столбец только тех значений, которые *в данный момент* содержатся также в другом столбце другой или этой же таблицы (внешний ключ). Эти ограничения носят динамический характер.

Существуют два основных типа ограничений — ограничения на столбцы и ограничения на таблицу. Ограничения на столбцы (**COLUMN CONSTRAINTS**) применимы только к отдельным столбцам, а ограничения на таблицу (**TABLE CONSTRAINTS**) применимы к группам, состоящим из одного или более столбцов. Ограничения на столбец

добавляются в конце определения столбца после указания типа данных и перед окончанием описания столбца (запятой). Ограничения на таблицу размещаются в конце определения таблицы, после определения последнего столбца. Команда **CREATE TABLE** имеет следующий синтаксис, расширенный включением ограничений:

```
CREATE TABLE <имя таблицы>
   (<имя столбца><тип данных><ограничения на столбец>,
    <имя столбца><тип данных><ограничения на столбец>,
    ...
    <ограничения на таблицу>
    (<имя столбца>[, <имя столбца>...])...);
```

Поля, заданные в круглых скобках после описания ограничений таблицы, — это поля, на которые эти ограничения распространяются. Ограничения на столбцы применяются к тем столбцам, за именами которых они описаны.

4.5.1. Ограничение NOT NULL. Чтобы запретить возможность использования в поле **NULL**-значений, можно при создании таблицы командой **CREATE TABLE** указать для соответствующего столбца ключевое слово **NOT NULL**. Это ограничение применимо только к столбцам таблицы. Как уже говорилось выше, **NULL** — это специальный маркер, обозначающий тот факт, что поле пусто. Но он полезен не всегда. Первичные ключи, например, в принципе не должны содержать **NULL**-значений (быть пустыми), поскольку это нарушило бы требование уникальности первичного ключа (более строго – функциональную зависимость атрибутов таблицы от первичного ключа). Во многих других случаях также необходимо, чтобы поля *обязательно* содержали определенные значения. Если ключевое слово **NOT NULL** размещается непосредственно после типа данных (включая размер) столбца, то любые попытки оставить значение поля пустым (ввести в поле **NULL**-значение) будут отвергнуты системой.

Например, для того, чтобы в определении таблицы STUDENT1 запретить использование **NULL**-значений для столбцов STUDENT_ID, SURNAME и NAME, можно записать следующее:

```
CREATE TABLE STUDENT1
   (STUDENT_ID INTEGER NOT NULL,
    SURNAME CHAR(25) NOT NULL,
    NAME CHAR(10) NOT NULL,
    STIPEND INTEGER,
    KURS INTEGER,
    CITY CHAR(15),
    BIRTHDAY DATE,
    UNIV_ID INTEGER);
```

Важно помнить, что если для столбца указано **NOT NULL**, то при использовании оператора **INSERT** обязательно должно быть указано

конкретное значение, вводимое в это поле. При отсутствии ограничения **NOT NULL** в столбце значение может отсутствовать, если только не указано значение столбца по умолчанию (**DEFAULT**). Если при создании таблицы ограничение **NOT NULL** не было указано, то его можно указать позже, используя оператор **ALTER TABLE**. Для вновь вводимого с помощью оператора **ALTER TABLE** столбца можно задать ограничение **NOT NULL**, если таблица, в которую добавляется столбец, пустая, или если для столбца указывается значение по умолчанию.

4.5.2. Уникальность как ограничение на столбец. Иногда требуется, чтобы все значения, введенные в столбец, отличались друг от друга. Например, этого требуют первичные ключи. Если при создании таблицы для столбца указывается ограничение **UNIQUE**, то база данных отвергает любую попытку ввести в это поле какой-либо строки значение, уже содержащееся в том же поле другой строки. Можно предложить следующее определение таблицы STUDENT, использующее ограничение **UNIQUE**:

```
CREATE TABLE STUDENT1
   (STUDENT_ID INTEGER NOT NULL UNIQUE,
    SURNAME CHAR(25) NOT NULL,
    NAME CHAR(10) NOT NULL,
    STIPEND INTEGER,
    KURS INTEGER,
    CITY CHAR(15),
    BIRTHDAY DATE,
    UNIV_ID INTEGER);
```

Объявляя поле STUDENT_ID уникальным, можно быть уверенным, что в таблице не появится записей для двух студентов с одинаковыми идентификаторами. Столбцы, отличные от первичного ключа, для которых требуется поддержать уникальность значений, называются возможными ключами (**CANDIDATE KEYS**) или уникальными ключами (**UNIQUE KEYS**).

4.5.3. Уникальность как ограничение таблицы. Можно установить требование уникальности для сочетания значений группы полей. В этом случае ключевое слово **UNIQUE** указывается в качестве ограничений *таблицы*. При объединении полей в группу важен порядок, в котором они указываются. Ограничение на таблицу **UNIQUE** полезно, если требуется поддерживать уникальность группы полей. Например, если в нашей базе данных не допускается, чтобы студент сдавал в один день больше одного экзамена, то можно в таблице объявить уникальной комбинацию значений полей STUDENT_ID и EXAM_DATE. Для этого следует создать таблицу EXAM_MARKS таким способом:

```
CREATE TABLE EXAM_MARKS1
   (EXAM_ID INTEGER NOT NULL,
```

```
    STUDENT_ID INTEGER NOT NULL,
    SUBJ_ID INTEGER NOT NULL,
    MARK INTEGER,
    EXAM_DATE DATE NOT NULL,
    UNIQUE (STUDENT_ID, EXAM_DATE));
```

Обратите внимание на то, что оба поля в ограничении таблицы **UNIQUE** все еще используют ограничение столбца **NOT NULL**. Если бы использовалось ограничение столбца **UNIQUE** для поля STUDENT_ID, то такое ограничение таблицы было бы необязательным.

Если значения поля STUDENT_ID должны быть уникальными для каждой строки в таблице EXAM_MARKS, то это можно сделать, объявив **UNIQUE** как ограничение самого поля STUDENT_ID. В этом случае будет обеспечена уникальность в комбинации значений полей STUDENT_ID, EXAM_DATE. Следовательно, указание **UNIQUE** как ограничение таблицы наиболее полезно использовать в случаях, когда не требуется уникальность индивидуальных полей, как это имеет место на самом деле в рассматриваемом примере.

4.5.4. Присвоение имен ограничениям. Ограничениям таблиц можно присваивать уникальные имена. Преимущество явного задания имени ограничения состоит в том, что в этом случае при выдаче системой сообщения о нарушении установленного ограничения будет указано его имя, что упрощает обнаружение ошибок.

Для присвоения имени ограничению используется несколько измененный синтаксис команд **CREATE TABLE** и **ALTER TABLE**.

Приведенный выше пример запроса изменяется следующим образом:

```
CREATE TABLE EXAM_MARKS1
   (EXAM_ID INTEGER NOT NULL,
    STUDENT_ID INTEGER NOT NULL,
    SUBJ_ID INTEGER NOT NULL,
    MARK INTEGER,
    EXAM_DATE DATE NOT NULL,
CONSTRAINT STUD_SUBJ
        UNIQUE (STUDENT_ID, EXAM_DATE));
```

В этом запросе STUD_SUBJ_CONSTR — это имя, присвоенное указанному ограничению таблицы.

4.5.5. Ограничение первичных ключей. *Первичные ключи таблицы* — это специальные случаи комбинирования ограничений **UNIQUE** и **NOT NULL**. Первичные ключи имеют следующие особенности:

- таблица может содержать только один первичный ключ;
- внешние ключи по умолчанию ссылаются на первичный ключ таблицы;
- первичный ключ является идентификатором строк таблицы (строки, однако, могут идентифицироваться и другими способами).

Улучшенный вариант создания таблицы STUDENT1 с объявленным первичным ключом имеет теперь следующий вид:

```
CREATE TABLE STUDENT1
   (STUDENT_ID INTEGER PRIMARY KEY,
    SURNAME CHAR(25) NOT NULL,
    NAME CHAR(10) NOT NULL,
    STIPEND INTEGER,
    KURS INTEGER,
    CITY CHAR(15),
    BIRTHDAY DATE,
    UNIV_ID INTEGER);
```

4.5.6. Составные первичные ключи. Ограничение **PRIMARY KEY** может также быть применено для нескольких полей, составляющих уникальную комбинацию значений — *составной* первичный ключ. Рассмотрим таблицу EXAM_MARKS. Очевидно, что ни к полю идентификатора студента (STUDENT_ID), ни к полю идентификатора предмета обучения (EXAM_ID) по отдельности нельзя предъявить требование уникальности. Однако для того, чтобы в таблице не могли появиться разные записи для одинаковых комбинаций значений полей STUDENT_ID и EXAM_ID (конкретный студент на конкретном экзамене не может получить более одной оценки), имеет смысл объявить уникальной комбинацию этих полей. Для этого мы можем применить ограничение таблицы **PRIMARY KEY**, объявив пару EXAM_ID и STUDENT_ID первичным ключом таблицы:

```
CREATE TABLE NEW_EXAM_MARKS
   (STUDENT_ID INTEGER NOT NULL,
    SUBJ_ID INTEGER NOT NULL,
    MARK INTEGER,
    DATA DATE,
CONSTRAINT EX_PR_KEY
   PRIMARY KEY (EXAM_ID, STUDENT_ID));
```

4.5.7. Проверка значений полей. Ограничение **CHECK** позволяет определять условие, которому должно удовлетворять вводимое в поле таблицы значение, прежде чем оно будет принято. Любая попытка обновить или заменить значение поля таким, для которого предикат, задаваемый ограничением **CHECK**, имеет значение ***ложь***, будет отвергаться.

Рассмотрим таблицу STUDENT. Значение столбца STIPEND в этой таблице STUDENT выражается десятичным числом. Наложим на значения этого столбца следующее ограничение — величина размера стипендии должна быть меньше 200.

Соответствующий запрос имеет следующий вид:

```
CREATE TABLE STUDENT
   (STUDENT_ID INTEGER PRIMARY KEY,
    SURNAME CHAR(25) NOT NULL,
    NAME CHAR(10) NOT NULL,
    STIPEND INTEGER CHECK (STIPEND < 200),
    KURS INTEGER,
    CITY CHAR(15),
    BIRTHDAY DATE,
    UNIV_ID INTEGER);
```

4.5.8. Проверка ограничивающих условий с использованием составных полей. Ограничение **CHECK** можно использовать в качестве табличного ограничения, то есть при необходимости включить более одного поля строки в ограничивающее условие.

Предположим, что ограничение на размер стипендии (меньше 200) должно распространяться только на студентов, живущих в Воронеже. Это можно указать в запросе со следующим табличным ограничением **CHECK**:

```
CREATE TABLE STUDENT
   (STUDENT_ID INTEGER PRIMARY KEY,
    SURNAME CHAR(25) NOT NULL,
    NAME CHAR(10) NOT NULL,
    STIPEND INTEGER,
    KURS INTEGER,
    CITY CHAR(15),
    BIRTHDAY DATE,
    UNIV_ID INTEGER UNIQUE,
   CHECK ((STIPEND < 200 AND CITY = 'Воронеж') OR
                        NOT (CITY = 'Воронеж')));
```

или в несколько иной записи:

```
CREATE TABLE STUDENT
   (STUDENT_ID INTEGER PRIMARY KEY,
    SURNAME CHAR(25) NOT NULL,
    NAME CHAR(10) NOT NULL,
    STIPEND INTEGER,
    KURS INTEGER,
    CITY CHAR(15),
    BIRTHDAY DATE,
    UNIV_ID INTEGER UNIQUE,
   CONSTRAINT STUD_CHECK
     CHECK(STIPEND < 200 AND CITY = 'Воронеж') OR
                        NOT (CITY = 'Воронеж')));
```

4.5.9. Установка значений по умолчанию. В SQL имеется возможность при вставке в таблицу строки, не указывая значений неко-

торого поля, определять значение этого поля по умолчанию. Наиболее часто используемым значением по умолчанию является **NULL**. Это значение принимается по умолчанию для любого столбца, для которого не было установлено ограничение **NOT NULL**.

Значение поля по умолчанию указывается в команде **CREATE TABLE** тем же способом, что и ограничение столбца, с помощью ключевого слова

DEFAULT *<значение по умолчанию>;*

Опция **DEFAULT** не является ограничением, так как она не ограничивает значения, вводимые в поле, а просто конкретизирует значение поля в случае, если оно *не было задано*.

Предположим, что основная масса студентов, информация о которых находится в таблице STUDENT, проживает в Воронеже. Чтобы при задании атрибутов не вводить для большинства студентов название города 'Воронеж', можно установить его как значение поля CITY по умолчанию, определив таблицу STUDENT следующим образом:

```
CREATE TABLE STUDENT
   (STUDENT_ID INTEGER PRIMARY KEY,
    SURNAME CHAR(25) NOT NULL,
    NAME CHAR(10) NOT NULL,
    STIPEND INTEGER CHECK (STIPEND < 200),
    KURS INTEGER,
    CITY CHAR(15) DEFAULT 'Воронеж',
    BIRTHDAY DATE,
    UNIV_ID INTEGER);
```

Другая цель практического применения задания значения по умолчанию — это использование его как альтернативы для **NULL**. Как уже отмечалось выше, присутствие **NULL** в качестве возможных значений поля существенно усложняет интерпретацию операций сравнения, в которых участвуют значения таких полей, поскольку **NULL** представляет собой признак того, что фактическое значение поля *неизвестно* или *неопределенно*. Следовательно, строго говоря, сравнение с ним любого конкретного значения в рамках двузначной булевой логики некорректно, за исключением специальной операции сравнения **IS NULL**, которая определяет, является ли содержимое поля каким-либо значением или оно отсутствует. Действительно, каким образом в рамках двузначной логики ответить на вопрос, истинно или ложно условие: CITY = 'Воронеж', если текущее значение поля CITY неизвестно (содержит **NULL**)?

Во многих случаях использование вместо **NULL** значения, подставляемого в поле по умолчанию, может существенно упростить использование значений поля в предикатах.

Например, можно установить для столбца опцию **NOT NULL**, а для неопределенных значений числового типа установить значение

по умолчанию “равно нулю”, или для полей типа **CHAR** — пробел, использование которых в операциях сравнения не вызывает никаких проблем.

При использовании значений по умолчанию в принципе допустимо применять ограничения **UNIQUE** или **PRIMARY KEY** в соответствующем поле. При этом, однако, следует иметь в виду отсутствие в таком ограничении практического смысла, поскольку только одна строка в таблице сможет принять значение, совпадающее с этим значением по умолчанию.

УПРАЖНЕНИЯ

1. Создайте таблицу EXAM_MARKS так, чтобы не допускался ввод в таблицу двух записей об оценках одного студента по конкретным экзамену и предмету обучения, а также, чтобы не допускалось проведение двух экзаменов по любым предметам в один день.
2. Создайте таблицу предметов обучения SUBJECT так, чтобы количество отводимых на предмет часов по умолчанию было равно 36, не допускались записи с отсутствующим количеством часов, поле SUBJ_ID являлось первичным ключом таблицы, а значения семестров (поле SEMESTR) лежали в диапазоне от 1 до 12.
3. Создайте таблицу EXAM_MARKS таким образом, чтобы значения поля EXAM_ID были больше значений поля SUBJ_ID, а значения поля SUBJ_ID были больше значений поля STUDENT_ID; пусть также будут запрещены значения **NULL** в любом из этих трех полей.

4.6. Поддержка целостности данных

В таблицах рассматриваемой базы данных значения некоторых полей связаны друг с другом. Так, поле STUDENT_ID в таблице STUDENT и поле STUDENT_ID в таблице EXAM_MARKS связаны тем, что описывают одни и те же объекты, т.е. содержат идентификаторы студентов, информация о которых хранится в базе. Более того, значения идентификаторов студентов, которые допустимы в таблице EXAM_MARKS, должны выбираться только из списка значений STUDENT_ID, фактически присутствующих в таблице STUDENT, т.е. принадлежащих реально описанным в базе студентам. Аналогично, значения поля UNIV_ID таблицы STUDENT должны соответствовать идентификаторам университетов UNIV_ID, фактически присутствующим в таблице UNIVERSITY, а значения поля SUBJ_ID таблицы EXAM_MARKS должны соответствовать идентификаторам предметов обучения, фактически присутствующим в таблице SUBJECT.

Ограничения, накладываемые указанным типом связи, называются *ограничениями ссылочной целостности*. Они составляют важную часть описания характеристик предметной области, обеспечения кор-

ректности данных, хранящихся в таблицах. Команды описания таблиц DML имеют средства, позволяющие описывать ограничения ссылочной целостности и обеспечивать поддержание такой целостности при манипулировании значениями полей базы данных.

4.6.1. Внешние и родительские ключи. Когда каждое значение, присутствующее в одном поле таблицы, представлено в другом поле другой или этой же таблицы, говорят, что первое поле ссылается на второе. Это указывает на прямую связь между значениями двух полей. Поле, которое ссылается на другое поле, называется ***внешним ключом***, а поле, на которое ссылается другое поле, называется ***родительским ключом***. В качестве родительского ключа может выступать только поле, являющееся возможным (первичным или альтернативным) ключом отношения. Например, поле `UNIV_ID` таблицы `STUDENT` — это внешний ключ, ссылающийся на поле `UNIV_ID` таблицы `UNIVERSITY`, являющееся ее первичным ключом и выступающее в данном случае в качестве родительского ключа для этого внешнего ключа.

Хотя в приведенном примере имена внешнего и родительского ключей совпадают, они *не обязательно* должны быть одинаковыми, хотя часто их сознательно задают одинаковыми, чтобы соединение было более наглядным.

4.6.2. Составные внешние ключи. Подобно первичному ключу, внешний ключ может состоять как из одного, так и из нескольких полей. Внешний ключ и родительский ключ, на который он ссылается, конечно же, должны быть определены на одинаковом множестве полей (по количеству полей, типам полей и порядку следования полей). Внешние ключи, состоящие из одного поля — применяемые в типовых таблицах настоящего издания — наиболее часты на практике. Чтобы сохранить простоту обсуждения, будем говорить о внешнем ключе, как об одиночном столбце, хотя все, что будет излагаться о поле, которое является внешним ключом, справедливо и для составных внешних ключей, определенных на группе полей.

4.6.3. Смысл внешнего и родительского ключей. Когда поле является внешним ключом, оно определенным образом связано с таблицей, на которую этот ключ ссылается. Каждое значение в этом поле (внешнем ключе) непосредственно привязано к конкретному значению в другом поле (родительском ключе). Значения родительского ключа должны быть уникальными, так как он одновременно является ключом отношения. Значения внешнего ключа *не обязательно* должны быть уникальными, т.е. в отношении может быть любое число строк с одинаковым сочетанием значений атрибутов, составляющих внешний ключ. При этом строки, содержащие одинаковые значения внешнего ключа, должны *обязательно* ссылаться на конкретное, присутствующее в данный момент в таблице, значение родительского ключа или быть неопределенными (**`NULL`**). Ни в одной строке таблицы не должно быть значений внешнего ключа, для которых в текущий момент от-

сутствуют соответствующие значения родительского ключа. Другими словами, не должно быть так называемых "висячих" ссылок внешнего ключа. Если указанные требования выполняются в конкретный момент существования базы данных, то говорят, что данные находятся в *согласованном* состоянии, а сама база находится в состоянии *ссылочной целостности*.

4.6.4. Ограничение внешнего ключа (FOREIGN KEY). Для решения вопросов поддержания ссылочной целостности в SQL используется ограничение **FOREIGN KEY**. Назначение **FOREIGN KEY** — это ограничение допустимых значений поля множеством значений родительского ключа, ссылка на который указывается при описании данного ограничения **FOREIGN KEY**.

Проблемы обеспечения ссылочной целостности возникают как при вводе значений поля, являющегося внешним ключом, так и при модификации/удалении значений родительского ключа, т. е. поля, на которое ссылается этот ключ. Одно из действий ограничения **FOREIGN KEY** — это отклонение (блокировка) ввода значений внешнего ключа, отсутствующих в таблице с родительским ключом. Это ограничение воздействует также на возможность изменять или удалять значения родительского ключа.

Ограничение **FOREIGN KEY** используется в командах **CREATE TABLE** и **ALTER TABLE** при создании или модификации таблицы, содержащей поле, которое требуется объявить внешним ключом. В команде указывается имя родительского ключа, на который имеется ссылка в ограничении **FOREIGN KEY**.

4.6.5. Внешний ключ как ограничение таблицы. Синтаксис ограничения **FOREIGN KEY** имеет следующий вид:

```
FOREIGN KEY (<список столбцов>)
   REFERENCES <родительская таблица>
                [(<родительский ключ>)];
```

В этом предложении *<список столбцов>* — это список из одного или более столбцов таблицы, которые будут созданы или изменены командами **CREATE TABLE** или **ALTER TABLE** (должны быть отделены друг от друга запятыми). Параметр *<родительская таблица>* — это имя таблицы, содержащей родительский ключ. Это, в частности, может быть именем таблицы, которая создается или изменяется текущей командой. Параметр *<родительский ключ>* представляет собой список столбцов родительской таблицы, которые составляют собственно родительский ключ. Оба списка столбцов, определяющих внешний и родительский ключи, должны быть совместимы, а именно:

- списки должны содержать одинаковое число столбцов;
- последовательность (1-й, 2-й, 3-й и т. д.) столбцов списка внешнего ключа должны иметь типы данных и размеры, совпадающие

с соответствующими (1-м, 2-м, 3-м и т. д.) столбцами списка родительского ключа.

Создадим таблицу STUDENT с полем UNIV_ID, определенным в качестве внешнего ключа, ссылающегося на таблицу UNIVERSITY:

```
CREATE TABLE STUDENT
   (STUDENT_ID INTEGER PRIMARY KEY,
    SURNAME CHAR(25),
    NAME CHAR(10),
    STIPEND INTEGER,
    KURS INTEGER,
    CITY CHAR(15),
    BIRTHDAY DATE,
    UNIV_ID INTEGER,
CONSTRAINT UNIV_FOR_KEY FOREIGN KEY (UNIV_ID)
                    REFERENCES UNIVERSITY (UNIV_ID));
```

Если ограничение **FOREIGN KEY** устанавливается в уже существующей таблице с помощью оператора **ALTER TABLE**, то имеющиеся в этой таблице значения внешнего ключа и значения родительского ключа таблицы, на которую ссылается устанавливаемый внешний ключ, должны находиться в состоянии ссылочной целостности. В противном случае команда будет отклонена.

Синтаксис команды **ALTER TABLE** в этом случае имеет следующий вид:

```
ALTER TABLE <имя таблицы>
     ADD CONSTRAINT <имя ограничения>
   FOREIGN KEY (<список столбцов внешнего ключа>)
     REFERENCES <имя родительской таблицы>
              [(<список столбцов родительского ключа>)];
```

Например, команда

```
ALTER TABLE STUDENT
     ADD CONSTRAINT STUD_UNIV_FOR_KEY
   FOREIGN KEY (UNIV_ID)
   REFERENCES UNIVERSITY (UNIV_ID);
```

добавляет ограничение внешнего ключа для таблицы STUDENT.

4.6.6. Внешний ключ как ограничение столбцов. Если определяемый внешний ключ не является составным, а состоит из единственного столбца, то ограничение внешнего ключа может устанавливаться непосредственно в строке, описывающей этот столбец. При таком варианте, называемом *ссылочным ограничением столбца*, ключевое слово **FOREIGN KEY** фактически не используется. Просто используется ключевое слово **REFERENCES**, и далее указывается имя родительского ключа, подобно следующему примеру:

```
CREATE TABLE STUDENT
   (STUDENT_ID INTEGER PRIMARY KEY,
    SURNAME CHAR(25),
    NAME CHAR(10),
    STIPEND INTEGER,
    KURS INTEGER,
    CITY CHAR(15),
    BIRTHDAY DATE,
    UNIV_ID INTEGER REFERENCES UNIVERSITY(UNIV_ID));
```

Команда определяет поле STUDENT.UNIV_ID как внешний ключ, использующий в качестве родительского ключа поле UNIVERSITY.UNIV_ID, являющееся ключом таблицы UNIVERSITY.

Эта форма эквивалентна следующему ограничению таблицы STUDENT:

```
FOREIGN KEY (UNIV_ID)
   REGERENCES UNIVERSITY (UNIV_ID)
```

или, в другой записи,

```
CONSTRAINT UNIV_FOR_KEY FOREIGN KEY (UNIV_ID)
                    REFERENCES UNIVERSITY (UNIV_ID).
```

Если в родительской таблице у родительского ключа указано ограничение **PRIMARY KEY**, то при указании ограничения **FOREIGN KEY**, накладываемого на таблицу или на столбцы, *можно не указывать список столбцов родительского ключа*. Естественно, в случае использования ключей со многими полями порядок столбцов в соответствующих внешних и первичных ключах должен совпадать, и в любом случае должен быть соблюден принцип совместимости между двумя ключами.

Например, если ограничение **PRIMARY KEY** размещено в поле UNIV_ID таблицы UNIVERSITY:

```
CREATE TABLE UNIVERSITY
   (UNIV_ID INTEGER PRIMARY KEY,
    UNIV_NAME CHAR(10),
    RATING INTEGER,
    CITY CHAR(15));
```

то в таблице STUDENT поле UNIV_ID можно использовать в качестве внешнего ключа, не указывая в ссылке имя родительского ключа:

```
CREATE TABLE STUDENT
   (STUDENT_ID INTEGER PRIMARY KEY,
    SURNAME CHAR(25),
    NAME CHAR(10),
    STIPEND INTEGER,
    KURS INTEGER,
```

```
CITY CHAR(15),
BIRTHDAY DATE,
UNIV_ID INTEGER REFERENCES UNIVERSITY);
```

Такая возможность встроена в язык для обеспечения использования первичных ключей в качестве родительских.

4.6.7. Поддержание ссылочной целостности и ограничения значений родительского ключа. Поддержание ссылочной целостности требует выполнения некоторых ограничений на значения, которые могут быть заданы в полях, объявленных как внешний ключ и родительский ключ. Набор значений родительского ключа должен быть таким, чтобы каждому значению внешнего ключа в родительской таблице обязательно соответствовала одна и только одна строка, указанная соответствующим родительским ключом. Это означает, что родительский ключ должен быть *уникальным*. Следовательно, при объявлении внешнего ключа необходимо убедиться, что все поля, которые используются как родительские ключи, имеют или ограничение **PRIMARY KEY**, или ограничения **UNIQUE**.

4.6.8. Использование первичного ключа в качестве уникального внешнего ключа. Ссылка внешних ключей только на первичные ключи считается хорошим стилем программирования SQL-запросов. В этом случае используемые внешние ключи связываются не просто с родительскими ключами, на которые они ссылаются, а с одной конкретной строкой родительской таблицы, в которой будет найдено соответствующее значение родительского ключа. Сам по себе родительский ключ не обеспечивает никакой информации, которая бы не была уже представлена во внешнем ключе. Внешний ключ — это не просто связь между двумя идентичными значениями столбцов двух таблиц, но связь *между двумя строками двух таблиц*.

Так как назначение первичного ключа состоит именно в том, чтобы однозначно идентифицировать строку, то использование ссылки на него в качестве внешнего ключа является более логичным и более однозначным выбором для внешнего ключа. Внешний ключ, который не имеет никакой другой цели, кроме связывания строк, напоминает первичный ключ, используемый исключительно для идентификации строк, и является хорошим средством сохранения наглядности и простоты структуры базы данных.

4.6.9. Ограничения значений внешнего ключа. Внешний ключ может содержать только те значения, которые фактически представлены в родительском ключе или являются пустыми (**NULL**). Попытка ввести другие значения в этот ключ должна быть отклонена, поэтому объявление внешнего ключа как **NOT NULL** не является обязательным.

4.6.10. Действие ограничений внешнего и родительского ключей при использовании команд модификации. Как уже говорилось, при использовании команд **INSERT** и **UPDATE** для модификации

значений столбца, объявленного как *внешний ключ*, вновь вводимые значения должны уже быть обязательно представлены в фактически присутствующих значениях столбца, объявленного родительским ключом. При этом можно помещать в эти поля пустые (**NULL**) значения, несмотря на то, что значения **NULL** недопустимы в родительских ключах. Можно также удалять (**DELETE**) любые строки с внешними ключами из таблицы, в которой эти ключи объявлены.

При необходимости модификации значений *родительского ключа* дело обстоит иначе. Использование команды **INSERT**, которая осуществляет ввод новой записи, не вызывает никаких особенностей, при которых возможно нарушение ссылочной целостности. Однако команда **UPDATE**, изменяющая значение родительского ключа, и команда **DELETE**, удаляющая строку, содержащую такой ключ, содержат возможность нарушения согласованности значений родительского и ссылающихся на него внешних ключей. Например, может возникнуть так называемая "*висячая*" ссылка внешнего ключа на несуществующее значение родительского ключа, что совершенно недопустимо. Чтобы при применении команд **UPDATE** и **DELETE** к полю, являющемуся родительским ключом, не нарушалась целостность ссылки, возможны следующие варианты действий.

- Любые изменения значений родительского ключа *запрещаются* и при попытке их совершения отвергаются (ограничение **NO ACTION** или **RESTRICT**). Эта спецификация действия применяется по умолчанию.
- Изменения значений родительского ключа *разрешаются*, но при этом автоматически осуществляется *коррекция* всех значений внешних ключей, ссылающихся на модифицируемое значение родительского ключа. Это называется *каскадным изменением* (ограничение **CASCADE**).
- Изменения значений родительского ключа *разрешаются*, но при этом соответствующие значения внешнего ключа автоматически *удаляются*, т. е. заменяются значением **NULL** (ограничение **SET NULL**).
- Изменения значений родительского ключа *разрешаются*, но при этом соответствующие значения внешнего ключа автоматически *заменяются* значением по умолчанию (ограничение **SET DEFAULT**).

При описании внешнего ключа должно указываться, какой из приведенных вариантов действий следует применять, причем в общем случае это должно быть указано раздельно для каждой из команд **UPDATE** и **DELETE**. В качестве примера использования ограничений, накладываемых на операции модификации родительских ключей, можно привести следующий запрос:

```
CREATE TABLE NEW_EXAM_MARKS
   (STUDENT_ID INTEGER NOT NULL,
```

```
      SUBJ_ID INTEGER NOT NULL,
      MARK INTEGER,
      DATA DATE,
CONSTRAINT EXAM_PR_KEY
   PRIMARY KEY(STUDENT_ID, SUBJ_ID),
CONSTRAINT SUBJ_ID_FOR_KEY FOREIGN KEY (SUBJ_ID)
                    REFERENCES SUBJECT,
CONSTRAINT STUDENT_ID_FOR_KEY FOREIGN KEY(STUDENT_ID)
        REFERENCES STUDENT ON UPDATE CASCADE
                           ON DELETE NO ACTION);
```

В этом примере при попытке изменения значения поля STUDENT_ID таблицы STUDENT будет автоматически обеспечиваться каскадная корректировка этих значений в таблице EXAM_MARKS, т. е. при изменении идентификатора студента STUDENT_ID в таблице STUDENT сохранятся все ссылки на его оценки. Однако любая попытка удаления (**DELETE**) записи о студенте из таблицы STUDENT будет отвергаться, если в таблице EXAM_MARKS существуют записи об оценках данного студента.

УПРАЖНЕНИЯ

1. Создайте таблицу с именем SUBJECT_1, с теми же полями, что и в таблице SUBJECT (предмет обучения). Поле SUBJ_ID является первичным ключом.
2. Создайте таблицу с именем SUBJ_LECT_1 (учебные дисциплины преподавателей), с полями LECTURER_ID (идентификатор преподавателя) и SUBJ_ID (идентификатор предмета обучения). Первичным ключом (составным) таблицы является пара атрибутов LECTURER_ID и SUBJ_ID; кроме того, поле LECTURER_ID является внешним ключом, ссылающимся на таблицу LECTURER_1, аналогичную таблице LECTURER (преподаватель), а поле SUBJ_ID является внешним ключом, ссылающимся на таблицу SUBJECT_1, аналогичную таблице SUBJECT.
3. Создайте таблицу с именем SUBJ_LECT_1 как в предыдущем задании, но добавьте для всех ее внешних ключей режим обеспечения ссылочной целостности, запрещающий обновление и удаление соответствующих родительских ключей.
4. Создайте таблицу с именем LECTURER_1, с теми же полями, что и в таблице LECTURER. Первичным ключом таблицы является атрибут LECTURER_ID; кроме того, поле UNIV_ID является внешним ключом, ссылающимся на таблицу UNIVERSITY_1 (аналог UNIVERSITY). Для этого поля установите каскадные режимы обеспечения целостности для команд **UPDATE** и **DELETE**.
5. Создайте таблицу с именем UNIVERSITY_1, с теми же полями, что и в таблице UNIVERSITY (университеты). Поле UNIV_ID является первичным ключом.

6. Создайте таблицу с именем EXAM_MARKS_1. Она должна содержать те же поля, что и таблица EXAM_MARKS (экзаменационные оценки). Комбинация полей EXAM_ID, STUDENT_ID и SUBJ_ID является первичным ключом. Кроме того, поля STUDENT_ID и SUBJ_ID являются внешним ключами, ссылающимися соответственно на таблицы STUDENT_1 и SUBJECT_1. Для этих полей установите режим каскадного обеспечения ссылочной целостности при операции обновления соответствующих первичных ключей и режим блокировки операции удаления родительского ключа при наличии ссылки на него.
7. Создайте таблицу с именем STUDENT_1. Она должна содержать те же поля, что и таблица STUDENT, и новое поле SENIOR_STUDENT (староста), значением которого должен быть идентификатор студента, являющегося старостой группы, в которой учится данный студент. Укажите необходимые для этого ограничения ссылочной целостности.
8. Создайте таблицу STUDENT_2, аналогичную таблице STUDENT, в которой поле UNIV_ID (идентификатор университета) является внешним ключом, ссылающимся на таблицу UNIVERSITY_1, таким образом, чтобы при удалении из таблицы UNIVERSITY_1 строки с информацией о каком-либо университете в соответствующих записях таблицы STUDENT_2 поле UNIV_ID очищалось (замещалось на **NULL**).
9. С помощью команды **CREATE TABLE** создайте запросы для формирования таблиц учебной базы данных, представленной в разделе 1.7, с указанием первичных ключей, но без указания ограничений внешних ключей. Затем с помощью команды **ALTER TABLE** укажите для сформированных таблиц все ограничения, в том числе и ограничения ссылочной целостности.

Глава 5

ПРЕДСТАВЛЕНИЯ (VIEW)

5.1. Представления — именованные запросы

До сих пор речь шла о таблицах, обычно называемых базовыми таблицами. Это — таблицы, которые содержат данные. Однако имеется и другой вид таблиц, называемый **VIEW** (ПРЕДСТАВЛЕНИЯ). Таблицы-представления не содержат никаких собственных данных. Фактически ***представление*** — *это* ***именованная*** *таблица*, *содержимое которой является результатом запроса*, *заданного при описании представления*, причем данный запрос выполняется всякий раз, когда таблица-представление становится объектом выражения SQL. При этом в каждый момент вывод запроса становится содержанием представления. Представления позволяют:

- ограничивать число столбцов, из которых пользователь выбирает или в которые вводит данные;
- ограничивать число строк, из которых пользователь выбирает или в которые вводит данные;
- выводить дополнительные столбцы, преобразованные из других столбцов базовой таблицы;
- выводить группы строк таблицы.

Благодаря этому представления дают возможность гибкой настройки выводимой из таблиц информации в соответствии с требованиями конкретных пользователей, позволяют обеспечивать защиту информации на уровне строк и столбцов, упрощают формирование сложных отчетов и выходных форм.

Представление определяется с помощью команды **CREATE VIEW** (СОЗДАТЬ ПРЕДСТАВЛЕНИЕ). Например:

```
CREATE VIEW MOSC_STUD AS
   SELECT *
   FROM STUDENT
   WHERE CITY = 'Москва';
```

Данные из базовой таблицы, предъявляемые пользователю в представлении, зависят от условия (предиката), описанного в **SELECT**-запросе при определении представления.

В созданную в результате приведенного выше запроса таблицу-представление MOSC_STUD передаются данные из базовой таблицы STUDENT, но не все, а только записи о студентах, для которых значение поля CITY равно 'Москва'. К таблице MOSC_STUD можно теперь обращаться с помощью запросов так же, как и к любой другой таблице базы данных. Например, запрос для просмотра представления MOSC_STUD имеет вид:

```
SELECT *
   FROM MOSC_STUD;
```

Представление, в котором выбираются *все* строки и столбцы базовой таблицы, например,

```
CREATE VIEW NEW_STUD_TAB AS
   SELECT *
   FROM STUDENT;
```

по сути эквивалентно применению синонима, но менее эффективно, поэтому применяется редко.

В следующем примере представление выбирает все строки и столбцы; кроме того, в качестве имен столбцов применяются псевдонимы:

```
CREATE VIEW NEW_STUDENT
   (NEW_STUDENT_ID, NEW_SURNAME, NEW_NAME,
    NEW_STIPEND, NEW_KURS, NEW_CITY,
    NEW_BIRTHDAY, NEW_UNIV_ID)
AS SELECT STUDENT_ID, SURNAME, NAME, STIPEND,
          KURS, CITY, BIRTHDAY, UNIV_ID
   FROM STUDENT;
```

Такое представление является простым способом организации общей таблицы для группы пользователей или прикладных задач, которые используют собственные имена полей и таблицы.

5.2. Модификация представлений

Данные, предъявляемые пользователю через представление, могут изменяться с помощью команд модификации DML, но при этом фактическая модификация данных будет осуществляться не в само́й виртуальной таблице-представлении, а будет перенаправлена к соответствующей базовой таблице. Например, запрос на обновление представления NEW_STUDENT

```
UPDATE NEW_STUDENT
   SET CITY = 'Москва'
```

```
WHERE STUDENT_ID = 1004;
```

эквивалентен выполнению команды **UPDATE** над базовой таблицей STUDENT. Следует, однако, обратить внимание на то, что в общем случае, из-за того, что обычно в представлении данные из базовой таблицы отображаются в *преобразованном* или *усеченном* виде, применение команд модификации к таблицам-представлениям имеет некоторые особенности, рассматриваемые ниже.

5.3. Маскирующие представления

5.3.1. Представления, маскирующие столбцы. Данный вид представлений ограничивает число столбцов базовой таблицы, к которым возможен доступ. Например, представление

```
CREATE VIEW STUD AS
   SELECT STUDENT_ID, SURNAME, CITY
      FROM STUDENT;
```

дает пользователю доступ к полям STUDENT_ID, SURNAME, CITY базовой таблицы STUDENT, полностью скрывая от него как содержимое, так и сам факт наличия в базовой таблице полей NAME, STIPEND, KURS, BIRTHDAY и UNIV_ID.

5.3.2. Операции модификации в представлениях, маскирующих столбцы. Представления, как уже отмечалось выше, могут изменяться с помощью команд модификации DML, но при этом модификация данных будет осуществляться не в самой таблице-представлении, а в соответствующей базовой таблице. Поэтому с представлениями, маскирующими столбцы, функции вставки и удаления работают несколько иначе, чем с обычными таблицами. Оператор **INSERT**, примененный к представлению, фактически осуществляет вставку строки в соответствующую базовую таблицу, причем *во все* столбцы этой таблицы независимо от того, видны они пользователю через представление или скрыты от него. В связи с этим в столбцах, не включенных в представление, устанавливается **NULL**-значение или значение по умолчанию. Если для не включенного в представление столбца действует ограничение **NOT NULL** а значение по умолчанию не определено, то генерируется сообщение об ошибке.

Любое применение оператора **DELETE** удаляет строки базовой таблицы независимо от их значений.

5.3.3. Представления, маскирующие строки. Представления могут также ограничивать доступ к строкам. Охватываемые представлением строки базовой таблицы задаются условием (предикатом) в конструкции **WHERE** при описании представления. Доступ через представление возможен только к строкам, удовлетворяющим условию.

Например, представление

```
CREATE VIEW MOSC_STUD AS
   SELECT *
   FROM STUDENT
   WHERE CITY = 'Москва';
```

показывает пользователю только те строки таблицы STUDENT, для которых значение поля CITY равно 'Москва'.

5.3.4. Операции модификации в представлениях, маскирующих строки. Каждая включенная в представление строка доступна для вывода, обновления и удаления. Любая допустимая для базовой таблицы строка вставляется в базовую таблицу независимо от ее включения в представление. При этом может возникнуть проблема, состоящая в том, что значения, введенные пользователем в базовую таблицу через представление значений, будут отсутствовать в представлении, оставаясь при этом в базовой таблице. Рассмотрим такой случай:

```
CREATE VIEW HIGH_RATING AS
   SELECT *
      FROM UNIVERSITY
      WHERE RATING = 300;
```

Это представление является обновляемым. Оно просто ограничивает доступ пользователя к определенным столбцам и строкам в таблице UNIVERSITY. Предположим, необходимо вставить с помощью команды **INSERT** следующую строку:

```
INSERT INTO HIGH_RATING
   VALUES (180, 'Новый университет',
           200, 'Воронеж');
```

Команда **INSERT** допустима в этом представлении. С помощью представления HIGH_RATING строка будет вставлена в базовую таблицу UNIVERSITY. Однако после появления этой строки в базовой таблице *из самого представления она исчезнет*, поскольку значение поля RATING не равно 300, и, следовательно, эта строка не удовлетворяет условию предложения **WHERE** для отбора строк в представление. Для пользователя такое исчезновение только что введенной строки является неожиданным. Действительно, непонятно, почему после ввода строки в таблицу ее нельзя увидеть и, например, тут же удалить. Тем более, что пользователь может вообще не знать, работает ли он в данный момент с базовой таблицей или с таблицей-представлением.

Аналогичная ситуация возникнет, если в какой-либо существующей записи представления HIGH_RATING изменить значение поля RATING на значение, отличное от 300.

Подобные проблемы можно устранить путем включения в определение представления опции **WITH CHECK OPTION**. Эта опция распространяет условие **WHERE** для запроса на операции обновления и вставки в описание представления. Например:

```
CREATE VIEW HIGH_RATUNG AS
   SELECT *
      FROM UNIVERSITY
      WHERE RATING = 300
   WITH CHECK OPTION;
```

В этом случае вышеупомянутые операции вставки строки или коррекции поля RATING будет отклонены.

Опция **WITH CHECK OPTION** помещается в определение представления, а не в команду **DML**, так что *все* команды модификации в представлении будут проверяться. Рекомендуется использовать эту опцию во всех случаях, когда нет причины разрешать представлению помещать в таблицу значения, которые в нем самом не могут быть видны.

5.3.5. Операции модификации в представлениях, маскирующих строки и столбцы. Рассмотренная выше проблема возникает и при вставке строк в представление с предикатом, использующим поля базовой таблицы, не присутствующие в самом представлении. Например, рассмотрим представление

```
CREATE VIEW MOSC_STUD AS
   SELECT STUDENT_ID, SURNAME, STIPEND
   FROM STUDENT
   WHERE CITY = 'Москва';
```

Видно, что в данное представление не включено поле CITY таблицы STUDENT.

При попытке вставки строки в это представление операция не будет выполнена, если столбец CITY в таблице STUDENT определен с ограничением **NOT NULL** и без значения по умолчанию. Если для столбца CITY определено значение по умолчанию, отличное от **NULL**, то это значение и попадет в столбец CITY. Так как в этом случае значение поля CITY базовой таблицы STUDENT не будет равняться значению 'Москва', вставляемая строка будет исключена из самого представления и поэтому не будет видна пользователю. Это будет происходить для *любой* вставляемой в представление MOSC_STUD строки. Другими словами, пользователь вообще не сможет видеть строки, вводимые им в это представление. Данная проблема не решается и в том случае, когда в определение представления добавляется опция **WITH CHECK OPTION**, если только для столбца CITY не определено значение по умолчанию 'Москва':

```
CREATE VIEW MOSC_STUD AS
   SELECT STUDENT_ID, SURNAME, STIPEND
   FROM STUDENT
   WHERE CITY = 'Москва'
   WITH CHECK OPTION;
```

Таким образом, в определенном указанными способами представлении можно модифицировать значения полей или удалять строки, но нельзя *вставлять* строки. Исходя из этого, рекомендуется даже в тех случаях, когда этого не требуется по соображениям полезности (и даже безопасности) информации, при определении представления включать в него *все* поля, на которые имеется ссылка в предикате. Если эти поля не должны отображаться в выводе таблицы, всегда можно исключить их уже в запросе к представлению. Другими словами, можно было бы определить представление MOSC_STUD, например, так:

```
CREATE VIEW MOSC_STUD AS
   SELECT *
   FROM STUDENT
   WHERE CITY = 'Москва'
   WITH CHECK OPTION;
```

Эта команда заполнит в представлении поле CITY одинаковыми значениями, которые можно просто исключить из вывода с помощью другого запроса уже к этому сформированному представлению, указав в запросе только поля, необходимые для вывода:

```
SELECT STUDENT_ID, SURNAME, STIPEND
   FROM MOSC_STUD;
```

5.4. Агрегированные представления

Создание представлений с использованием агрегированных функций и предложения **GROUP BY** является удобным инструментом для непрерывной обработки и интерпретации извлекаемой информации. Предположим, необходимо следить за количеством студентов, сдающих экзамены, количеством сданных экзаменов, количеством сданных предметов, средним баллом по каждому предмету. Для этого можно сформировать следующее представление:

```
CREATE VIEW TOTALDAY AS
   SELECT EXAM_DATE, COUNT(DISTINCT SUBJ_ID)
          AS SUBJ_CNT,
      COUNT(STUDENT_ID) AS STUD_CNT,
      COUNT(MARK) AS MARK_CNT,
      AVG(MARK) AS MARK_AVG, SUM(MARK) AS MARK_SUM
   FROM EXAM_MARKS
   GROUP BY EXAM_DATE;
```

Теперь требуемую информацию можно увидеть с помощью простого запроса к представлению:

```
SELECT * FROM TOTALDAY;
```

5.5. Представления, основанные на нескольких таблицах

Представления часто используются для объединения нескольких таблиц (базовых и/или других представлений) в одну большую виртуальную таблицу. Такое решение имеет ряд преимуществ:

- представление, объединяющее несколько таблиц, при формировании сложных отчетов может использоваться как промежуточный макет, скрывающий детали объединения большого количества исходных таблиц;
- предварительно объединенные поисковые и базовые таблицы обеспечивают наилучшие условия для транзакций, позволяют использовать компактные схемы кодов, устраняя необходимость написания длинных объединяющих процедур для каждого отчета;
- позволяет использовать при формировании отчетов более надежный модульный подход;
- предварительно объединенные и проверенные представления уменьшают вероятность ошибок, связанных с неполным выполнением условий объединения.

Можно, например, создать представление, которое показывает имена и названия сданных предметов для каждого студента:

```
CREATE VIEW STUD_SUBJ AS
   SELECT A.STUDENT_ID, C.SUBJ_ID,
          A.SURNAME, C.SUBJ_NAME
      FROM STUDENT A, EXAM_MARKS B, SUBJECT C
      WHERE A.STUDENT_ID = B.STUDENT_ID
        AND B.SUBJ_ID = C.SUBJ_ID;
```

Теперь все предметы, сданные студентом, или всех студентов, сдавших данный предмет, можно выбрать с помощью простого запроса. Например, чтобы увидеть все предметы, сданные студентом Ивановым, подается запрос:

```
SELECT SUBJ_NAME
   FROM STUD_SUBJ
   WHERE SURNAME = 'Иванов';
```

5.6. Представления и подзапросы

При создании представлений могут также использоваться подзапросы, включая связанные подзапросы. Предположим, предусматривается премия для тех студентов, которые имеют самый высокий балл на любую заданную дату. Получить такую информацию можно с помощью представления:

```
CREATE VIEW ELITE_STUD
   AS SELECT B.EXAM_DATE, A.STUDENT_ID, A.SURNAME
   FROM STUDENT A, EXAM_MARKS B
      WHEREA.STUDENT_ID = B.STUDENT_ID
      AND B.MARK =
         (SELECT MAX (MARK)
          FROM EXAM_MARKS C
          WHERE C.EXAM_DATE = B.EXAM_DATE);
```

Если, с другой стороны, премия будет назначаться только студенту, который имел самый высокий балл, причем не меньше 10 раз, то необходимо использовать другое представление, основанное на первом:

```
CREATE VIEW BONUS
   AS SELECT DISTINCT STUDENT_ID, SURNAME
      FROM ELITE_STUD A
         WHERE 10 <=
            (SELECT COUNT(*)
             FROM ELITE_STUD B
             WHERE A.STUDENT_ID = B.STUDENT_ID);
```

Извлечение из этой таблицы записей о студентах, которые будут получать премию, выполняется простым запросом:

```
SELECT * FROM BONUS;
```

5.7. Удаление представлений

Синтаксис удаления представления из базы данных подобен синтаксису удаления базовых таблиц:

DROP VIEW *<имя представления>*

УПРАЖНЕНИЯ

1. Создайте представление для получения сведений обо всех студентах, имеющих только отличные оценки.
2. Создайте представление для получения сведений о количестве студентов в каждом городе.
3. Создайте представление для получения следующих сведений по каждому студенту: его идентификатор, фамилия, имя, средний и общий баллы.
4. Создайте представление для получения сведений о количестве экзаменов, которые сдавал каждый студент.

5.8. Изменение значений в представлениях

Как уже говорилось, использование команд модификации языка SQL — **INSERT** (ВСТАВИТЬ), **UPDATE** (ЗАМЕНИТЬ), и **DELETE** (УДАЛИТЬ) — применительно к представлениям имеет ряд особенностей. В дополнение к аспектам, рассмотренным выше, следует отметить, что не все представления могут модифицироваться.

Если в представлении могут выполняться команды модификации, то представление является обновляемым (модифицируемым); в противном случае оно предназначено только для чтения при запросе. Каким образом можно определить, является ли представление модифицируемым? Критерии обновляемости представления можно сформулировать следующим образом.

- Представление строится на основе одной и только одной базовой таблицы.
- Представление должно содержать первичный ключ базовой таблицы.
- Представление не должно иметь никаких полей, которые представляют собой агрегирующие функции.
- Представление не должно содержать **DISTINCT** в своем определении.
- Представление не должно использовать **GROUP BY** или **HAVING** в своем определении.
- Представление не должно использовать подзапросы.
- Представление *может быть* использовано в другом представлении, но это представление должно быть также модифицируемым.
- Представление не должно использовать в качестве полей вывода константы или выражения значений.

Суть этих ограничений в том, что обновляемые представления фактически подобны окнам в базовых таблицах. Они показывают информацию из базовой таблицы, ограничивая определенные ее строки (использованием соответствующих предикатов) или специально именованные столбцы (с исключениями). Но при этом представления выводят значения *без их обработки* с использованием агрегирующих функций и группировки. Они также не сравнивают строки таблиц друг с другом (как это имеет место в объединениях и подзапросах или при использовании **DISTINCT**).

Различия между модифицируемыми (обновляемыми) представлениями и представлениями "*только для чтения*" не случайны. Обновляемые представления в основном используются аналогично базовым таблицам. Пользователи могут даже не знать, является ли запрашиваемый ими объект базовой таблицей или представлением. Это превосходный механизм защиты для скрытия частей таблицы, которые конфиденциальны или не предназначены данному пользователю.

Немодифицируемые представления, с другой стороны, позволяют более рационально получать и переформатировать данные. С их помощью формируются библиотеки сложных запросов, которые могут затем использоваться в запросах для получения информации самостоятельно (например, в объединениях). 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 оценки студента), не показывая конкретных значений оценок и тем более не позволяя их модифицировать.

5.9. Примеры обновляемых и необновляемых представлений

Пример 1.

```
CREATE VIEW DATEEXAM (EXAM_DATE, QUANTITY)
   AS SELECT EXAM_DATE, COUNT (*)
      FROM EXAM_MARKS
      GROUP BY EXAM_DATE;
```

Данное представление — *необновляемое* из-за присутствия в нем агрегирующей функции и **GROUP BY**.

Пример 2.

```
CREATE VIEW LCUSTT
   AS SELECT *
      FROM UNIVERSITY
      WHERE CITY = 'Москва';
```

Это — *обновляемое* представление.

Пример 3.

```
CREATE VIEW SSTUD (SURNAME1, NUMB, KUR)
   AS SELECT SURNAME, STUDENT_ID, KURS*2
      FROM STUDENT
      WHERE CITY = 'Москва';
```

Это представление — *немодифицируемое* из-за наличия выражения "KURS*2".

Пример 4.

```
CREATE VIEW STUD3
   AS SELECT *
      FROM STUDENT
      WHERE STUDENT_ID IN
```

```
(SELECT MARK
 FROM EXAM_MARKS
 WHERE EXAM_DATE = '10/02/1999');
```

Из-за наличия подзапроса представление лучше отнести к *немодифицируемым*, хотя некоторые СУБД могут допускать его обновление.

Пример 5.

```
CREATE VIEW SOMEMARK
   AS SELECT STUDENT_ID, SUBJ_ID, MARK
      FROM EXAM_MARKS
      WHERE EXAM_DATE
         IN ('10/02/1999', '10/06/1999');
```

Это — *обновляемое* представление.

УПРАЖНЕНИЯ

1. Какие из представленных ниже представлений являются обновляемыми?

а)
```
CREATE VIEW DAILYEXAM AS
    SELECT DISTINCT STUDENT_ID, SUBJ_ID, MARK,
                    EXAM_DATE
    FROM EXAM_MARKS;
```

б)
```
CREATE VIEW CUSTALS AS
    SELECT SUBJECT.SUBJ_ID, SUM (MARK) AS MARK1
    FROM SUBJECT, EXAM_MARKS
    WHERE SUBJECT.SUBJ_ID = EXAM_MARKS.SUBJ_ID
    GROUP BY SUBJECT.SUBJ_ID;
```

в)
```
CREATE VIEW THIRDEXAM AS SELECT *
    FROM DAILYEXAM
    WHERE EXAM_DATE = '10/02/1999';
```

г)
```
CREATE VIEW NULLCITIES
    AS SELECT STUDENT_ID, SURNAME, CITY
    FROM STUDENT
    WHERE CITY IS NULL
    OR SURNAME BETWEEN 'А' AND 'Д';
```

2. Создайте представление таблицы STUDENT с именем STIP, включающее поля STIPEND и STUDENT_ID и позволяющее вводить или изменять значение поля STIPEND (стипендия), но только в пределах от 100 до 200.

Глава 6

ОПРЕДЕЛЕНИЕ ПРАВ ДОСТУПА ПОЛЬЗОВАТЕЛЕЙ К ДАННЫМ

6.1. Пользователи и привилегии

Каждый, кто имеет доступ к базе данных, называется *пользователем*. SQL используется обычно в многопользовательских средах, которые требуют разграничения прав пользователей с точки зрения доступа к данным и прав на выполнение с ними тех или иных манипуляций. Для этих целей в SQL реализованы средства, позволяющие устанавливать и контролировать привилегии пользователей базы данных.

Каждый пользователь в среде SQL имеет специальное имя, или идентификатор, с помощью которого осуществляется идентификация пользователя для установки и определения его прав с точки зрения доступа к данным. Каждая посланная к СУБД команда SQL-запроса ассоциируется СУБД с идентификатором и правами доступа соответствующего пользователя.

Пользователь может быть определен с помощью следующего оператора:

```
CREATE USER <имя_пользователя>
   IDENTIFIED BY <пароль>;
```

После выполнения этого оператора пользователь становится известен базе данных, но пока не может выполнять никаких операций.

Для удаления пользователя используется оператор

```
DROP USER <имя_пользователя>;
```

Назначаемые пользователю привилегии — это то, что определяет, может ли указанный пользователь выполнить данную команду над определенным объектом базы данных или нет. Имеется несколько типов привилегий, соответствующих нескольким типам операций. Привилегии даются и отменяются двумя командами SQL, соответственно:

GRANT — установка привилегий,

REVOKE — отмена привилегий.

6.2. Стандартные привилегии

Привилегии, определенные SQL, — это привилегии объекта. Это означает, что пользователь имеет привилегию (право) на выполнение данной команды только на определенном объекте в базе данных. Привилегии объекта связаны одновременно и с пользователями, и с таблицами базы данных, т.е. привилегия дается определенному пользователю в указанной таблице. Это может быть как базовая таблица, так и представление.

Пользователь, создавший таблицу (любого вида), является *владельцем* этой таблицы. Это означает, что этот пользователь имеет *все* привилегии, относящиеся к этой таблице, в том числе он может передавать привилегии на работу с этой таблицей другим пользователям.

Пользователю могут быть назначены следующие привилегии:

- **SELECT** — пользователь может выполнять запросы к таблице;
- **INSERT** — пользователь может выполнять в таблице команду **INSERT**;
- **UPDATE** — пользователь может выполнять в таблице команду **UPDATE**. Эта привилегия может быть ограничена для определенных столбцов таблицы;
- **DELETE** — пользователь может выполнять в таблице команду **DELETE**;
- **REFERENCES** — пользователь может определить внешний ключ, который использует в качестве родительского ключа один или более столбцов этой таблицы. Возможно ограничение этой привилегии для определенных столбцов.

Кроме того, могут быть нестандартные привилегии объекта, такие, как:

- **INDEX** — пользователь имеет право создавать индекс в таблице;
- **SYNONYM** — пользователь имеет право создавать синоним для объекта;
- **ALTER** — пользователь имеет право выполнять команду **ALTER TABLE** в таблице;
- **EXECUTE** — позволяет выполнять процедуру.

Назначение пользователям этих привилегий осуществляется с помощью команды **GRANT**.

6.3. Команда GRANT

Пользователь, являющийся владельцем таблицы `STUDENT`, может передать другому пользователю (пусть это будет пользователь с именем `IVANOV`) привилегию **SELECT** с помощью следующей команды.

```
GRANT SELECT ON STUDENT TO IVANOV;
```

Теперь пользователь с именем `IVANOV` может выполнять **SELECT**-запросы к таблице `STUDENT`. При отсутствии других приви-

легий он может только *выбирать* значения, но не может выполнять никаких действий, которые воздействовали бы на значения в таблице STUDENT, включая ее использование в качестве родительской таблицы внешнего ключа. Когда SQL получает команду **GRANT**, проверяются привилегии пользователя, давшего эту команду, чтобы определить допустимость команды **GRANT** для этого пользователя. Пользователь IVANOV самостоятельно не может задать эту команду. Он также не может предоставить право **SELECT** другому пользователю, так как таблица принадлежит не ему (ниже будет показано, как владелец таблицы может передать другому пользователю право предоставления привилегий).

Команда

```
GRANT INSERT ON EXAM_MARKS TO IVANOV;
```

предоставляет пользователю IVANOV право вводить в таблицу EXAM_MARKS новые строки.

В команде **GRANT** допустимо указывать через запятые список предоставляемых привилегий и список пользователей, которым они предоставляются. Например:

```
GRANT SELECT, INSERT ON SUBJECT TO IVANOV, PETROV;
```

При этом все указанные в списке привилегии предоставляются всем указанным пользователям. В строгой ANSI-интерпретации невозможно предоставить привилегии для нескольких таблиц одной командой **GRANT**.

6.4. Использование аргументов ALL и PUBLIC

Аргумент **ALL PRIVILEGES** (все привилегии) или просто **ALL** используется вместо имен привилегий в команде **GRANT**, чтобы предоставить все привилегии в таблице. Например, команда

```
GRANT ALL PRIVILEGES ON STUDENT TO IVANOV;
```

или более коротко:

```
GRANT ALL ON STUDENT TO IVANOV;
```

передает пользователю IVANOV ***весь*** набор привилегий в таблице STUDENT.

Аргумент **PUBLIC** используется для передачи указанных в команде привилегий *всем* остальным пользователям. Наиболее часто это применяется для привилегии **SELECT** в определенных базовых таблицах или представлениях, которые необходимо сделать доступными для любого пользователя. Например, чтобы позволить любому пользователю получать информацию из таблицы EXAM_MARKS, можно использовать команду

```
GRANT SELECT ON EXAM_MARKS TO PUBLIC;
```

Предоставление *всех* привилегий к таблице *всем* пользователям обычно является нежелательным. Все привилегии, за исключением **SELECT**, позволяют пользователю изменять содержание таблицы, поэтому разрешение всем пользователям изменять содержание таблиц может вызвать определенные проблемы обеспечения безопасности и защиты данных. Тем более, что привилегия **PUBLIC** не ограничена в передаче прав только текущим пользователям. Любой новый пользователь, добавляемый к системе, автоматически получает в этом случае полный набор привилегий, назначенный ранее всем пользователям. Поэтому для ограничения доступа к таблице всем и всегда лучше всего предоставить привилегии, отличные от **SELECT**, только индивидуальным пользователям.

6.5. Отмена привилегий

Отмена привилегии осуществляется с помощью команды **REVOKE** , которая имеет синтаксис, аналогичный команде **GRANT**.

Например, команда

```
REVOKE INSERT ON STUDENT FROM PETROV;
```

отменяет привилегию **INSERT** в таблице STUDENT для пользователя PETROV. Возможно использование в команде **REVOKE** списков привилегий и пользователей. Например:

```
REVOKE INSERT, DELETE ON STUDENT
   FROM PETROV, SIDOROV;
```

Следует иметь в виду, что привилегии отменяются тем пользователем, который их предоставил, и при этом отмена автоматически распространяется на всех пользователей, получивших от него эту привилегию.

6.6. Использование представлений для фильтрации привилегий

Действия привилегий можно сделать более точными, используя представления. Привилегия, передаваемая пользователю в базовой таблице, автоматически распространяется на все строки, а при использовании возможных исключений **UPDATE** и **REFERENCES** — и на все столбцы таблицы. Создавая представление, которое ссылается на базовую таблицу, и затем передавая привилегию уже на это представление, можно ограничить эти привилегии любыми выражениями в запросе, содержащемся в представлении. Такой метод расширяет возможности команды **GRANT**.

Для создания представлений пользователь должен обладать привилегией **SELECT** во всех таблицах, на которые он ссылается в представлении. Если представление модифицируемое, то любая из привилегий **INSERT**, **UPDATE** и **DELETE** , которая предоставлена пользователю в базовой таблице, будет автоматически распространяться на представление. Если привилегии на обновление отсутствуют, то их невозможно получить и в созданных представлениях, даже если сами эти представления обновляемые. Так как внешние ключи не применяются в представлениях, то и привилегия **REFERENCES** никогда не используется при создании представлений.

6.6.1. Ограничение привилегии SELECT для определенных столбцов. Предположим, необходимо обеспечить пользователю PETROV возможность доступа только к столбцам STUDENT_ID и SURNAME таблицы STUDENT. Это можно сделать, поместив имена этих столбцов в представление

```
CREATE VIEW STUDENT_VIEW AS
   SELECT STUDENT_ID, SURNAME
   FROM STUDENT;
```

и предоставив пользователю PETROV привилегию **SELECT** в созданном представлении, а не в самой таблице STUDENT:

```
GRANT SELECT ON STUDENT_VIEW TO PETROV;
```

Для столбцов можно создать различные привилегии, однако следует иметь в виду, что для команды **INSERT** это будет означать вставку значений по умолчанию, а для команды **DELETE** ограничение столбца вообще не будет иметь значения.

6.6.2. Ограничение привилегий для определенных строк. Представления позволяют ограничивать (фильтровать) привилегии для определенных строк таблицы. Для этого естественно использовать в представлении предикат, который определит, какие строки включены в представление. Чтобы предоставить пользователю PETROV привилегию вида **UPDATE** в таблице UNIVERSITY для всех записей о московских университетах, можно создать следующее представление:

```
CREATE VIEW MOSC_UNIVERSITY AS
   SELECT * FROM UNIVERSITY
   WHERE CITY = 'Москва'
   WITH CHECK OPTION;
```

Затем можно передать привилегию **UPDATE** в этой таблице пользователю PETROV:

```
GRANT UPDATE ON MOSC_UNIVERSITY TO PETROV;
```

В отличие от привилегии **UPDATE** для определенных столбцов, которая распространена на все строки таблицы UNIVERSITY, данная

привилегия относится только к строкам, для которых значение поля CITY равно 'Москва'. Предложение **WITH CHECK OPTION** предохраняет пользователя PETROV от замены значения поля CITY на любое значение, кроме значения 'Москва'.

6.6.3. Предоставление доступа только к извлеченным данным. Другая возможность состоит в том, чтобы устанавливать пользователям привилегии на доступ к уже извлеченным данным, а не к значениям в таблице. Для этого удобно использовать агрегирующие функции. Например, создадим представление, которое дает информацию о количестве оценок, среднем и общем баллах для студентов на каждый день:

```
CREATE VIEW DATETOTALS AS
   SELECT EXAM_DATE, COUNT (*) AS KOL,
          SUM (MARK) AS SUMMA, AVG (MARK) AS TOT
   FROM EXAM_MARKS
   GROUP BY EXAM_DATE;
```

Теперь можно передать пользователю PETROV привилегию **SELECT** в созданном представлении DATETOTALS с помощью запроса:

```
GRANT SELECT ON DATETOTALS TO PETROV;
```

6.6.4. Использование представлений в качестве альтернативы ограничениям. Представления с **WITH CHECK OPTION** могут использоваться в качестве альтернативы ограничениям. Например, необходимо удостовериться, что все значения поля CITY в таблице STUDENT равны названиям конкретных городов. Для этого можно установить ограничение **CHECK** непосредственно на столбец CITY. Однако позже его изменение будет затруднено. В качестве альтернативы можно создать представление, исключающее неправильные значения CITY:

```
CREATE VIEW CURCITYES AS
   SELECT *
   FROM STUDENT
   WHERE CITY IN ('Москва', 'Воронеж')
   WITH CHECK OPTION;
```

Теперь вместо того, чтобы предоставлять пользователям привилегии обновления в таблице STUDENT, можно предоставить соответствующие привилегии в представлении CURCITYES. Преимущество такого подхода состоит в том, что при необходимости изменения можно удалить это представление, создать новое и предоставить в этом новом представлении привилегии пользователям. Такая операция выполняется проще, чем изменение ограничений в таблице. Недостатком этого метода является то, что владелец таблицы STUDENT тоже должен использовать это новое представление, иначе его собственные команды также не будут приняты.

6.7. Другие типы привилегий

До сих пор не рассмотрены вопросы установки целого ряда других привилегий, а именно:

- Кто имеет право создавать таблицы?
- Кто имеет право изменять, удалять или ограничивать таблицы?
- Должны ли права создания базовых таблиц отличаться от прав создания представлений?
- Должен ли существовать *суперпользователь*, т.е. пользователь, отвечающий за поддержание базы данных и, следовательно, имеющий наибольшие, или полные привилегии, которые не предоставляются обычному пользователю?

Привилегии, которые не определяются в терминах специальных объектов данных, называются *привилегиями системы*, или *правами базы данных*. Эти привилегии включают в себя право создавать объекты данных, отличающиеся от базовых таблиц (обычно создаваемых несколькими пользователями) и представлений (обычно создаваемых большинством пользователей). Привилегии системы для создания представлений должны дополнять, а не заменять привилегии объекта, которые стандарт требует от создателей представлений (описаны ранее). Кроме того, в любой системе всегда имеются некоторые типы суперпользователей, т.е. пользователей, которые имеют большинство или все привилегии и могут передать свой статус суперпользователя кому-либо с помощью привилегии или группы привилегий. Такого рода пользователем является так называемый *администратор базы данных*, или DBA (Database Administrator).

6.8. Типичные привилегии системы

При общем подходе имеются три базовых привилегии системы:

- **CONNECT** (ПОДКЛЮЧИТЬ),
- **RESOURCE** (РЕСУРС),
- **DBA** (АДМИНИСТРАТОР БАЗЫ ДАННЫХ).

Привилегия **CONNECT** состоит из права зарегистрироваться и права создавать представления и синонимы, если переданы привилегии объекта.

Привилегия **RESOURCE** состоит из права создавать базовые таблицы.

Привилегия **DBA** — это привилегия администратора базы данных, т.е. суперпользователя, которому предоставляются самые высокие полномочия при работе с базой данных. Эту привилегию могут иметь пользователи с функциями администратора базы данных. Команда **GRANT** (в измененной форме) может применяться как с привилегиями объекта, так и с системными привилегиями.

6.9. Создание и удаление пользователей

В большинстве реализаций SQL нового пользователя создает пользователь с привилегией **DBA**, т.е. администратор базы данных, который автоматически предоставляет новому пользователю привилегию **CONNECT**. В этом случае обычно добавляется предложение **IDENTIFIED BY**, указывающее пароль для этого пользователя. Например, команда

```
GRANT CONNECT TO PETROV
   IDENTIFIED BY 'PETROVPASSWORD';
```

приведет к созданию пользователя с именем PETROV, предоставит ему право регистрироваться в базе данных, и назначит ему пароль 'PETROVPASSWORD'. После этого, так как PETROV уже является зарегистрированным пользователем, он (или пользователь **DBA**) может использовать эту же команду для изменения данного пароля 'PETROVPASSWORD'.

Когда пользователь A предоставляет привилегию **CONNECT** другому пользователю B, говорят, что пользователь A "*создает*" пользователя B. При этом пользователь A обязательно должен иметь привилегию **DBA**. Если пользователь B будет создавать базовые таблицы (а не только представления), то ему также должна быть предоставлена привилегия **RESOURCE**. Но при этом возникает другая проблема. При попытке удаления пользователем A привилегии **CONNECT** пользователя B, который уже имеет созданные им таблицы, эта команда удаления привилегии будет отклонена, поскольку ее действие оставит эти таблицы без владельца, что не допускается. Поэтому, прежде чем удалить привилегию **CONNECT** у какого-либо пользователя, сначала необходимо удалить из базы данных все созданные этим пользователем таблицы. Привилегию **RESOURCE** удалять отдельно не требуется, достаточно удалить **CONNECT**, чтобы удалить пользователя.

УПРАЖНЕНИЯ

1. Передайте пользователю PETROV право на изменение оценок студентов в базе данных.
2. Передайте пользователю SIDOROV право передавать другим пользователям права на осуществление запросов к таблице EXAM_MARKS.
3. Отмените привилегию **INSERT** по отношению к таблице STUDENT у пользователя IVANOV и у всех других пользователей, которым привилегия, в свою очередь, была предоставлена этим пользователем IVANOV.
4. Передайте пользователю SIDOROV право выполнять операции вставки или обновления в таблице UNIVERSITY, но только для записей об университетах, значения рейтингов которых лежат в диапазоне от 300 до 400.

5. Разрешите пользователю PETROV делать запросы к таблице EXAM_MARKS, но запретите ему изменять в этой таблице значения оценок студентам, имеющим неудовлетворительные (=2) оценки.

6.10. Создание синонимов (SYNONYM)

Каждый раз при ссылке к базовой таблице или представлению, не являющимся собственностью пользователя, требуется установить в качестве префикса к имени этой таблицы имя ее владельца, так как у разных пользователей могут оказаться таблицы с одинаковыми именами и в этом случае система не сможет определить местонахождение таблицы. Использование длинных имен с префиксами может оказаться неудобным. Поэтому большинство реализаций SQL позволяют создавать для таблиц синонимы. *Синоним*— это альтернативное имя таблицы. При создании синонима пользователь становится его собственником, поэтому необходимость использования префикса к имени таблицы для него отпадает. Пользователь имеет право создавать синоним для таблицы, если он имеет по крайней мере одну привилегию в одном или более столбцах этой таблицы.

С помощью команды **CREATE SYNONYM** пользователь IVANOV может для таблицы с именем PETROV.STUDENT создать синоним с именем CLIENTS следующим образом:

```
CREATE SYNONYM CLIENTS FOR PETROV.STUDENT;
```

Теперь пользователь IVANOV может использовать таблицу с именем CLIENTS в команде точно так же, как имя PETROV.STUDENT.

Как уже говорилось, префикс пользователя — это фактически часть имени любой таблицы. Всякий раз, когда пользователь не указывает собственное имя вместе с именем своей таблицы, SQL по умолчанию подставляет идентификатор пользователя в качестве префикса имени таблицы. Следовательно, два одинаковых имени таблицы, но связанные с различными владельцами, становятся неидентичными и, следовательно, не приводят к какой-либо путанице в запросах. Таким образом, два пользователя могут создавать две полностью независимые таблицы с одинаковыми именами, но это также означает, что один пользователь может создать представление, основанное на имени, стоящем после имени таблицы и используемом другим пользователем. Это иногда делается в случаях, когда представление используется как замена самой исходной таблицы, например, если представление просто использует **CHECK OPTION** как заменитель ограничения **CHECK** в базовой таблице. Можно также создавать собственные синонимы пользователя, имена которых будут такими же, как и первоначальные имена таблиц. Например, пользователь PETROV может определить имя STUDENT как свой синоним для таблицы IVANOV.STUDENT с помощью запроса:

```
CREATE SYNONYM STUDENT FOR IVANOV.STUDENT;
```

С точки зрения SQL, теперь имеются два разных имени одной таблицы: IVANOV.STUDENT и PETROV.STUDENT. Однако каждый из соответствующих пользователей может обращаться к данной таблице, используя имя STUDENT. SQL, как говорилось выше, сам добавит к этому имени недостающие имена пользователей в качестве префиксов.

6.11. Синонимы общего пользования (PUBLIC)

Если планируется использовать таблицу STUDENT большим числом пользователей, то удобнее, чтобы все пользователи ссылались к ней с помощью одного и того же имени. Это даст возможность, например, использовать указанное имя без ограничений в прикладных программах. Чтобы создать единое имя для всех пользователей, создается общий синоним.

Например, если все пользователи будут вызывать таблицу STUDENT с данными о студентах, можно присвоить ей *общий* синоним STUDENT следующим образом:

```
CREATE PUBLIC SYNONYM STUDENT FOR STUDENT;
```

Общие синонимы в основном создаются владельцами объектов или пользователями с привилегиями администратора базы данных (пользователь **DBA**). Другим пользователям при этом должны быть предоставлены соответствующие привилегии в таблице STUDENT, чтобы она была им доступна. Даже если имя является общим, сама таблица общей не является.

6.12. Удаление синонимов

Общие и другие синонимы могут удаляться командой **DROP SYNONYM**. Синонимы могут удаляться только их владельцами, кроме общих синонимов, которые могут удаляться соответствующими привилегированными пользователями (обычно это пользователи **DBA**). Чтобы удалить, например, синоним CLIENTS, когда вместо него уже появился общий синоним STUDENT, пользователь может ввести команду

```
DROP SYNONYM CLIENTS;
```

УПРАЖНЕНИЯ

1. Пользователь IVANOV передал Вам право **SELECT** в таблице EXAM_MARKS. Запишите команду, позволяющую Вам обращаться к этой таблице, используя имя EXAM_MARKS без префикса.
2. Вы передали право **SELECT** в таблице EXAM_MARKS пользователю IVANOV. Запишите команду, позволяющую ему обращаться к этой таблице, используя имя EXAM_MARKS без префикса.

Глава 7

УПРАВЛЕНИЕ ТРАНЗАКЦИЯМИ

В процессе выполнения последовательности команд SQL таблицы базы данных не всегда могут находиться в согласованном состоянии. В случае возникновения каких-либо сбоев, когда логически связанная последовательность запросов не доведена до конца, возможно нарушение целостности данных в базе. Для обеспечения целостности данных логически связанные последовательности запросов, неделимые с точки зрения воздействия на базу данных, объединяют в так называемые *транзакции*. При этом либо запросы, составляющие транзакцию, должны выполняться все полностью — с первого до последнего, и тогда транзакция завершается командой **COMMIT**, либо, если в силу каких-либо внешних причин это оказывается невозможным, внесенные запросами транзакции изменения в базе данных должны аннулироваться командой **ROLLBACK**. Во втором случае база данных возвращается в целостное состояние на момент, предшествующий началу транзакции. Это называют *откатом* транзакции.

Новая транзакция начинается после каждой команды **COMMIT** или **ROLLBACK**.

В большинстве реализаций можно установить параметр, называемый **AUTOCOMMIT**. Он будет автоматически запоминать все выполняемые действия над данными. Действия, которые приведут к ошибке при незавершенной транзакции, всегда будут автоматически “откатаны” обратно.

Имеется возможность установки режима **AUTOCOMMIT** автоматически при регистрации. Если сеанс пользователя завершается аварийно, например, произошел сбой системы или выполнена перезагрузка пользователя, то текущая транзакция выполнит автоматический откат изменений. Это — одна из возможностей управления выполнением диалоговой обработки запросов путем разделения команд на большое количество различных транзакций. Одиночная транзакция не должна содержать слишком много несвязанных команд, на практике она часто состоит из единственной команды. Хорошее правило, которому можно

следовать — это создавать транзакции из одной команды или нескольких близко связанных команд.

Например, требуется удалить сведения о студенте по фамилии Иванов из базы данных. Прежде, чем сведения будут удалены из таблицы STUDENT, требуется осуществить определенные действия с данными об этом студенте в других таблицах, в частности, с данными о его оценках. Необходимо установить в **NULL** соответствующее этому студенту поле STUDENT_ID в таблице EXAM_MARKS. После этого можно удалить запись об этом студенте из таблицы STUDENT. Эти действия выполняются с помощью двух запросов:

```
UPDATE EXAM_MARKS
   SET STUDENT_ID = NULL
   WHERE STUDENT_ID = 1004;

DELETE FROM STUDENT
   WHERE STUDENT_ID = 1004;
```

Если возникает проблема с удалением записи о студенте с фамилией Иванов (возможно, имеется другой внешний ключ, ссылающийся на него, о котором не было известно и который, соответственно, не учтен при удалении), то можно было бы отменить все сделанные изменения, по крайней мере до тех пор, пока проблема не будет решена. Для этого приведенную группу команд следует обрабатывать как одиночную транзакцию, предусматривая ее завершение с помощью команды **COMMIT** или **ROLLBACK** — в зависимости от результата.

ОТВЕТЫ К УПРАЖНЕНИЯМ

Раздел 2.1

```
1. SELECT SUBJ_ID, SUBJ_NAME, SEMESTR, HOUR
     FROM SUBJECT;
2.  SELECT *
     FROM EXAM_MARKS
     WHERE SUBJ_ID = 12;
3.  SELECT KURS, SURNAME, NAME, STIPEND
     FROM STUDENT;
4.  SELECT SUBJ_NAME, HOUR
     FROM SUBJECT
     WHERE SEMESTR = 4;
5.  SELECT DISTINCT MARK
     FROM EXAM_MARKS;
6.  SELECT SURNAME
     FROM STUDENT
     WHERE KURS >= 3;
7.  SELECT SURNAME, NAME, KURS
     FROM STUDENT
     WHERE STIPEND > 140;
8.  SELECT SUBJ_NAME
     FROM SUBJECT
     WHERE HOUR > 30;
9.  SELECT *
     FROM UNIVERSITY
     WHERE RATING > 300;
10. SELECT KURS, SURNAME, NAME
     FROM STUDENT
     WHERE STIPEND >= 100
       AND CITY = 'Воронеж';
   SELECT *
     FROM STUDENT
```

```
      WHERE BIRTHDAY < '01.09.1980' AND KURS = 1;
```

Дата приведена для примера; конкретная задаваемая дата должна быть на 25 лет раньше текущей.

```
13. SELECT *
      FROM SUBJECT
      WHERE SEMESTER = 1 AND HOUR > 100;
14. SELECT *
      FROM LECTURER
      WHERE CITY = 'Воронеж';
15. SELECT *
      FROM UNIVERSITY
      WHERE CITY = 'Москва' AND RATING < 296;
16. SELECT *
      FROM STUDENT
      WHERE CITY = 'Воронеж' AND STIPEND = 0;
17. SELECT *
      FROM STUDENT
      WHERE BIRTHDAY > '01.09.1985';
```

Дата приведена для примера; конкретная задаваемая дата должна быть на 20 лет раньше текущей.

```
18. SELECT *
      FROM STUDENT
      WHERE CITY IS NULL;
```

Раздел 2.2

```
1. SELECT SUBJ_ID
     FROM EXAM_MARKS
     WHERE EXAM_DATE
       BETWEEN '10/01/2005' AND '20/01/2005';
2. SELECT SUBJ_NAME
     FROM SUBJECT
     WHERE STUDENT_ID IN (12, 32);
3. SELECT SUBJ_NAME
     FROM SUBJECT
     WHERE SUBJ_NAME LIKE 'И%';
4. SELECT SURNAME,NAME
     FROM STUDENT
     WHERE NAME LIKE 'И%' OR NAME LIKE 'С%';
5. SELECT *
     FROM EXAM_MARKS
     WHERE MARK IS NULL;
6. SELECT *
     FROM EXAM_MARKS
     WHERE MARK IS NOT NULL;
```

```
 7. SELECT *
      FROM LECTURER
      WHERE CITY LIKE '%-%';
 8. SELECT *
      FROM UNIVERSITY
      WHERE UNIV_NAME LIKE '%"%';
 9. SELECT *
      FROM SUBJECT
      WHERE SUBJ_NAME LIKE '%ия';
10. SELECT *
      FROM UNIVERSITY
      WHERE UNIV_NAME LIKE '%университет%';
11. SELECT *
      FROM STUDENT
      WHERE SURNAME LIKE 'Ков%' OR
            SURNAME LIKE 'Куз%';
12. SELECT *
      FROM SUBJECT
      WHERE SUBJ_NAME LIKE '% %';
13. SELECT *
      FROM UNIVERSITY
      WHERE UNIV_NAME LIKE '% % % % % % %';
14. SELECT *
      FROM STUDENT
      WHERE SURNAME LIKE '___';
```

Раздел 2.3

```
1. SELECT STUDENT_ID || ';' || UPPER(SURNAME) || ';' ||
       UPPER(NAME) || ';' || STIPEND || ';' ||
       KURS || ';' || UPPER(CITY) || ';' ||
       TO_CHAR(BIRTHDAY,'DD.MM.YYYY') || ';' ||
       UNIV_ID
     FROM STUDENT;
2. SELECT SUBSTR(NAME,1,1) || '.' || UPPER(SURNAME) ||
       '; место жительства – ' || UPPER(CITY) ||
       '; родился – ' ||
       TO_CHAR(BIRTHDAY, 'DD.MM.YY')
     FROM STUDENT;
3. SELECT LOWER(SUBSTR(NAME,1,1) || '.' || SURNAME) ||
       '; место жительства – ' || LOWER(CITY) ||
       '; родился: ' ||
       TO_CHAR(BIRTHDAY, 'DD-mon-YYYY')
     FROM STUDENT;
4. SELECT NAME || ' ' || SURNAME || ' родился в ' ||
       TO_CHAR(BIRTHDAY, 'YYYY') || ' году'
     FROM STUDENT;
```

```
5. SELECT SURNAME, NAME, STIPEND*100
     FROM STUDENT;
6. SELECT UPPER(NAME || ' ' || SURNAME) ||
        ' родился в ' ||
       TO_CHAR(BIRTHDAY, 'YYYY') || ' году'
     FROM STUDENT
     WHERE KURS IN(1, 2, 4);
7. SELECT 'Код-' || UNIV_ID || '; ' || UNIV_NAME ||
        '-г. ' || UPPER(CITY) || '; Рейтинг=' || RATING
     FROM UNIVERSITY;
8. SELECT 'Код-' || UNIV_ID || '; ' || UNIV_NAME ||
        '-г. ' || UPPER(CITY) ||
        '; Рейтинг=' || ROUND(RATING,-2)
     FROM UNIVERSITY;
```

Раздел 2.4

```
1. SELECT COUNT(*)
     FROM EXAM_MARKS
     WHERE SUBJ_ID = 20;
2. SELECT COUNT(DISTINCT SUBJ_ID)
     FROM EXAM_MARKS;
3. SELECT STUDENT_ID, MIN(MARK)
     FROM EXAM_MARKS
     GROUP BY STUDENT_ID;
4. SELECT STUDENT_ID, MAX(MARK)
     FROM EXAM_MARKS
     GROUP BY STUDENT_ID;
5. SELECT MIN(SURNAME)
     FROM STUDENT
     WHERE SURNAME LIKE 'И%';
6. SELECT SUBJ_NAME, MAX(SEMESTR)
     FROM SUBJECT
     GROUP BY SUBJ_NAME;
7. SELECT EXAM_DATE, COUNT(DISTINCT STUDENT_ID)
     FROM EXAM_MARKS
     GROUP BY EXAM_DATE;
8. SELECT STUDENT_ID, AVG(MARK)
     FROM EXAM_MARKS
     GROUP BY STUDENT_ID;
9. SELECT EXAM_ID, AVG(MARK)
     FROM EXAM_MARKS
     GROUP BY EXAM_ID;
10. SELECT EXAM_ID, COUNT(STUDENT_ID)
     FROM EXAM_MARKS
     GROUP BY EXAM_ID;
```

```
11. SELECT CEIL(SEMESTER/2), COUNT(*)
      FROM SUBJECT
      GROUP BY CEIL(SEMESTER/2);
12. SELECT UNIV_ID, SUM(STIPEND)
      FROM STUDENT
      GROUP BY UNIV_ID
      ORDER BY SUM(STIPEND);
13. SELECT SEMESTER, SUM(HOUR)
      FROM SUBJECT
      GROUP BY SEMESTER;
14. SELECT STUDENT_ID, AVG(MARK)
      FROM EXAM_MARKS
      GROUP BY STUDENT_ID;
15. SELECT STUDENT_ID, SUBJ_ID, AVG(MARK)
      FROM EXAM_MARKS
      GROUP BY STUDENT_ID, SUBJ_ID;
16. SELECT CITY, COUNT(STUDENT_ID)
      FROM STUDENT
      GROUP BY CITY
      ORDER BY COUNT(STUDENT_ID) DESC;
17. SELECT UNIV_ID, COUNT(STUDENT_ID)
      FROM STUDENT
      GROUP BY UNIV_ID
      ORDER BY COUNT(STUDENT_ID);
18. SELECT UNIV_ID, COUNT(LECTURER_ID)
      FROM LECTURER
      GROUP BY UNIV_ID
      ORDER BY COUNT(LECTURER_ID);
19. SELECT UNIV_ID, KURS, SUM(STIPEND)
      FROM STUDENT
      GROUP BY UNIV_ID, KURS;
20. SELECT CITY, MAX(RATING)
      FROM UNIVERSITY
      GROUP BY CITY
      ORDER BY MAX(RATING);
21. SELECT EXAM_DATE, AVG(MARK)
      FROM EXAM_MARKS
      GROUP BY EXAM_DATE;
22. SELECT EXAM_DATE, SUBJ_ID, MAX(MARK)
      FROM EXAM_MARKS
      GROUP BY EXAM_DATE, SUBJ_ID;
23. SELECT EXAM_DATE, COUNT(DISTINCT STUDENT_ID)
      FROM EXAM_MARKS
      GROUP BY EXAM_DATE;
```

```
24. SELECT EXAM_DATE, STUDENT_ID, COUNT(*)
      FROM EXAM_MARKS
      GROUP BY EXAM_DATE, STUDENT_ID;
25. SELECT LECTURER_ID, COUNT(SUBJ_ID)
      FROM SUBJ_LECT
      GROUP BY LECTURER_ID;
26. SELECT SUBJ_ID, COUNT(LECTURER_ID)
      FROM SUBJ_LECT
      GROUP BY SUBJ_ID;
27. SELECT COUNT(*)
      FROM (SELECT STUDENT_ID, MIN(MARK)
            FROM EXAM_MARKS
            GROUP BY STUDENT_ID
            HAVING MIN(MARK)=5);
28. SELECT COUNT(*)
      FROM EXAM_MARKS
      WHERE MARK>2 AND STUDENT_ID=32;
```

Раздел 2.7

```
1. a) SELECT STUDENT_ID, SURNAME, STIPEND*1.2
        FROM STUDENT
        ORDER BY 3;
2. a) SELECT STUDENT_ID, MAX(MARK)
        FROM EXAM_MARKS
        GROUP BY STUDENT_ID;
3. a) SELECT SEMESTR, SUBJ_NAME, SUBJ_ID
        FROM SUBJECT
        ORDER BY SEMESTR DESC;
4. a) SELECT SUM(MARK), EXAM_DATE
        FROM EXAM_MARKS
        GROUP BY EXAM_DATE
        ORDER BY 1 DESC;
5. a) SELECT EXAM_DATE AVG(MARK),
        FROM EXAM_MARKS
        GROUP BY EXAM_DATE
        ORDER BY 2 DESC;
```

Раздел 2.8

```
1. SELECT *
     FROM STUDENT
     WHERE STIPEND =
       (SELECT MAX(STIPEND) FROM STUDENT)
     ORDER BY SURNAME;
2. SELECT *
     FROM STUDENT
```

```
     WHERE STIPEND >
       (SELECT AVG(STIPEND) FROM STUDENT);
3. SELECT *
     FROM STUDENT
     WHERE UNIV_ID IN
     (SELECT UNIV_ID FROM UNIVERSITY
       WHERE CITY='Воронеж')
     ORDER BY UNIV_ID, KURS;
4. SELECT *
     FROM SUBJECT
     WHERE HOUR=(SELECT MAX(HOUR) FROM SUBJECT);
5. SELECT SURNAME, NAME
     FROM STUDENT S
     WHERE CITY<>(SELECT CITY FROM UNIVERSITY
                  WHERE UNIV_ID=S.UNIV_ID);
6. SELECT *
     FROM UNIVERSITY WHERE CITY='Москва'
     AND RATING<(SELECT RATING FROM UNIVERSITY
                  WHERE UNIV_NAME='ВГУ');
```

Раздел 2.9

```
1.  SELECT * FROM STUDENT S
     WHERE CITY=(SELECT CITY FROM UNIVERSITY
                 WHERE UNIV_ID=S.UNIV_ID);
2.  SELECT * FROM STUDENT S
     WHERE CITY<>(SELECT CITY FROM UNIVERSITY
                  WHERE UNIV_ID=S.UNIV_ID)
     ORDER BY UNIV_ID, KURS;
3. SELECT * FROM LECTURER L
     WHERE CITY<>(SELECT CITY FROM UNIVERSITY
                  WHERE UNIV_ID=L.UNIV_ID)
     ORDER BY UNIV_ID, CITY;
4. SELECT * FROM SUBJECT S
     WHERE HOUR = (SELECT MAX(HOUR) FROM SUBJECT
                  WHERE SEMESTER=S.SEMESTER)
     ORDER BY SEMESTER;
5. SELECT * FROM STUDENT S
     WHERE STIPEND >
        (SELECT AVG(STIPEND) FROM STUDENT
         WHERE KURS=S.KURS);
6. SELECT * FROM STUDENT S
     WHERE STIPEND =
       (SELECT MIN(STIPEND) FROM STUDENT
         WHERE UNIV_ID=S.UNIV_ID)
     ORDER BY UNIV_ID, STIPEND;
```

```
7. SELECT * FROM UNIVERSITY U
     WHERE 50<(SELECT COUNT(*) FROM STUDENT
                  WHERE UNIV_ID=U.UNIV_ID)
     ORDER BY RATING;
8. SELECT * FROM UNIVERSITY U
     WHERE 5<(SELECT COUNT(*) FROM LECTURER
                 WHERE UNIV_ID=U.UNIV_ID)
     ORDER BY RATING;
9. SELECT * FROM STUDENT S
     WHERE 5=(SELECT MIN(MARK) FROM EXAM_MARKS
                 WHERE STUDENT_ID=S.STUDENT_ID)
     ORDER BY UNIV_ID, KURS;
10. SELECT * FROM STUDENT S
     WHERE 3>(SELECT MAX(MARK) FROM EXAM_MARKS
                 WHERE STUDENT_ID=S.STUDENT_ID)
     ORDER BY UNIV_ID, KURS;
11. SELECT * FROM STUDENT S
     WHERE 4<(SELECT AVG(MARK) FROM EXAM_MARKS
                 WHERE STUDENT_ID=S.STUDENT_ID);
12. SELECT DISTINCT SURNAME FROM STUDENT S
     WHERE 5=(SELECT MIN(MARK) FROM EXAM_MARKS
                 WHERE STUDENT_ID=S.STUDENT_ID)
       AND CITY<>(SELECT CITY FROM UNIVERSITY
                    WHERE UNIV_ID=S.UNIV_ID);
```

Раздел 2.10

```
1. SELECT *
     FROM EXAM_MARKS
     WHERE STUDENT_ID =
        (SELECT STUDENT_ID FROM STUDENT
         WHERE SURNAME = 'Иванов');
2. SELECT DISTINCT NAME
     FROM STUDENT, EXAM_MARKS
     WHERE MARK > (SELECT AVG(MARK) FROM EXAM_MARKS)
     AND STUDENT.STUDENT_ID = EXAM_MARKS.STUDENT_ID
     AND SUBJ_ID = 101;
3. SELECT DISTINCT NAME
     FROM STUDENT, EXAM_MARKS
     WHERE MARK < (SELECT AVG(MARK) FROM EXAM_MARKS)
     AND STUDENT.STUDENT_ID = EXAM_MARKS.STUDENT_ID
     AND SUBJ_ID = 102;
4. SELECT COUNT(SUBJ_ID)
     FROM EXAM_MARKS
     GROUP BY STUDENT_ID
     HAVING COUNT(SUBJ_ID) > 20;
```

```
5. SELECT NAME, STUDENT_ID
     FROM STUDENT A
     WHERE STIPEND = (SELECT MAX(STIPEND)
                        FROM STUDENT B
                        WHERE A.CITY = B.CITY);
6. SELECT NAME, STUDENT_ID
     FROM STUDENT A
     WHERE A.CITY IS NOT NULL
     AND A.CITY NOT IN
       (SELECT B.CITY FROM UNIVERSITY B);
7. SELECT NAME, STUDENT_ID
     FROM STUDENT A
     WHERE A.CITY IS NOT NULL
       AND A.UNIV_ID IS NOT NULL
       AND A.CITY NOT IN
         (SELECT B.CITY
          FROM UNIVERSITY B
          WHERE A.UNIV_ID = B.UNIV_ID);

   SELECT DISTINCT NAME, A.UNIV_ID
     FROM STUDENT A, UNIVERSITY B
     WHERE A.CITY <> B.CITY
       AND A.CITY IS NOT NULL
       AND A.UNIV_ID = B.UNIV_ID;
```

Раздел 2.11

```
1. SELECT *
     FROM STUDENT A
     WHERE EXISTS (SELECT * FROM UNIVERSITY B
     WHERE A.UNIV_ID = B.UNIV_ID AND RATING > 300);
2. SELECT NAME, A.UNIV_ID, STUDENT_ID, A.CITY
     FROM STUDENT A, UNIVERSITY B
     WHERE A.UNIV_ID = B.UNIV_ID AND RATING > 300;
3. SELECT * FROM STUDENT A
     WHERE EXISTS (SELECT * FROM UNIVERSITY B
                     WHERE A.UNIV_ID <> B.UNIV_ID
                       AND A.CITY = B.CITY);
4. SELECT SUBJ_NAME FROM SUBJECT A
     WHERE EXISTS (SELECT * FROM EXAM_MARKS B
       WHERE A.SUBJ_ID = B.SUBJ_ID AND B.MARK > 2
         AND 1 < (SELECT COUNT(*) FROM EXAM_MARKS C
                    WHERE C.SUBJ_ID = A.SUBJ_ID
         AND B.MARK > 2
         AND C.STUDENT_ID <> A.STUDENT_ID));
5. SELECT DISTINCT CITY FROM STUDENT S WHERE EXISTS
     (SELECT * FROM UNIVERSITY WHERE CITY=S.CITY);
```

```
6. SELECT DISTINCT CITY FROM STUDENT S
     WHERE NOT EXISTS (SELECT * FROM UNIVERSITY
                         WHERE CITY=S.CITY);
7. SELECT * FROM SUBJECT S WHERE NOT EXISTS
     (SELECT * FROM SUBJ_LECT
       WHERE SUBJ_ID=S.SUBJ_ID);
8. SELECT SUBJ_NAME, SEMESTER FROM SUBJECT S
     WHERE NOT EXISTS (SELECT * FROM SUBJECT
                        WHERE SUBJ_NAME=S.SUBJ_NAME
                        AND SUBJ_ID<>S.SUBJ_ID);
9. SELECT SUBJ_NAME, SEMESTER FROM SUBJECT S
     WHERE EXISTS (SELECT * FROM SUBJECT
                       WHERE SUBJ_NAME=S.SUBJ_NAME
                       AND SUBJ_ID<>S.SUBJ_ID);
10. SELECT UNIV_NAME FROM UNIVERSITY U
     WHERE NOT EXISTS (SELECT * FROM LECTURER
                         WHERE UNIV_ID=U.UNIV_ID);
11. SELECT UNIV_NAME FROM UNIVERSITY U
     WHERE NOT EXISTS (SELECT * FROM STUDENT
                         WHERE UNIV_ID=U.UNIV_ID);
12. SELECT * FROM SUBJECT S
     WHERE NOT EXISTS (SELECT * FROM EXAM_MARKS
       WHERE SUBJ_ID=S.SUBJ_ID AND MARK<3);
13. SELECT * FROM SUBJECT S
     WHERE EXISTS (SELECT * FROM EXAM_MARKS
              WHERE SUBJ_ID=S.SUBJ_ID AND MARK<3);
14. SELECT * FROM STUDENT S
     WHERE NOT EXISTS (SELECT * FROM EXAM_MARKS
       WHERE STUDENT_ID=S.STUDENT_ID AND MARK<3);
15. SELECT * FROM STUDENT S
     WHERE EXISTS (SELECT * FROM EXAM_MARKS
       WHERE STUDENT_ID=S.STUDENT_ID AND MARK<3);
16. SELECT SURNAME, NAME FROM STUDENT S
     WHERE EXISTS (SELECT * FROM EXAM_MARKS
       WHERE S.STUDENT_ID=STUDENT_ID AND MARK=5);
17. SELECT SURNAME, NAME FROM STUDENT S
     WHERE NOT EXISTS (SELECT * FROM EXAM_MARKS
       WHERE S.STUDENT_ID=STUDENT_ID AND MARK=5);
18. SELECT COUNT(*) FROM STUDENT S
     WHERE NOT EXISTS (SELECT * FROM EXAM_MARKS
           WHERE S.STUDENT_ID=STUDENT_ID);
19. SELECT COUNT(DISTINCT STUDENT_ID)
     FROM EXAM_MARKS S
     WHERE NOT EXISTS (SELECT * FROM EXAM_MARKS
       WHERE STUDENT_ID=S.STUDENT_ID AND MARK<5);
```

```
20. SELECT COUNT(*) FROM STUDENT S
      WHERE EXISTS (SELECT * FROM EXAM_MARKS
        WHERE STUDENT_ID=S.STUDENT_ID AND MARK<3)
        AND CITY<>(SELECT CITY FROM UNIVERSITY
                   WHERE UNIV_ID=S.UNIV_ID);
```

Раздел 2.12

```
1. SELECT DISTINCT SUBJ_NAME FROM SUBJECT S
     WHERE SUBJ_NAME IN (SELECT SUBJ_NAME
                         FROM SUBJECT
                         WHERE SUBJ_ID<>S.SUBJ_ID);
2. SELECT SURNAME, NAME FROM STUDENT
     WHERE STUDENT_ID IN (SELECT STUDENT_ID
                          FROM EXAM_MARKS
                          WHERE MARK>2);
3. SELECT DISTINCT STUDENT_ID FROM EXAM_MARKS
     WHERE MARK IN (SELECT MARK FROM EXAM_MARKS
                    WHERE STUDENT_ID=12);
4. SELECT COUNT(*) FROM STUDENT
     WHERE STUDENT_ID NOT IN (SELECT STUDENT_ID
                              FROM EXAM_MARKS);
5. SELECT * FROM SUBJECT
     WHERE SUBJ_ID IN (SELECT SUBJ_ID
               FROM SUBJ_LECT SL, LECTURER L
               WHERE SL.LECTURER_ID=L.LECTURER_ID
               AND SURNAME='Колесников');
6. SELECT SURNAME, NAME FROM LECTURER
     WHERE LECTURER_ID
     IN (SELECT SL.LECTURER_ID
         FROM SUBJECT S, SUBJ_LECT SL
         WHERE SL.SUBJ_ID=S.SUBJ_ID AND SEMESTER<3);
7. SELECT DISTINCT SURNAME
     FROM LECTURER L, SUBJ_LECT S
     WHERE L.LECTURER_ID=S.LECTURER_ID
       AND SUBJ_ID IN (SELECT SUBJ_ID
         FROM LECTURER L1,SUBJ_LECT S1
         WHERE L1.LECTURER_ID=S1.LECTURER_ID
         AND SURNAME='Сорокин');
8. SELECT SURNAME FROM STUDENT S, UNIVERSITY U
     WHERE S.UNIV_ID=U.UNIV_ID
       AND U.CITY <= ALL (SELECT CITY
         FROM UNIVERSITY WHERE CITY IS NOT NULL);
9. SELECT DISTINCT SURNAME FROM STUDENT S
     WHERE 5 = ALL (SELECT MARK FROM EXAM_MARKS
                    WHERE STUDENT_ID=S.STUDENT_ID)
```

```
    AND CITY<>(SELECT CITY FROM UNIVERSITY
               WHERE UNIV_ID=S.UNIV_ID);
```

То же с использованием связанных подзапросов:

```
SELECT DISTINCT SURNAME FROM STUDENT S
  WHERE 5=(SELECT MIN(MARK) FROM EXAM_MARKS
           WHERE STUDENT_ID=S.STUDENT_ID)
           AND CITY<>(SELECT CITY FROMUNIVERSITY
                      WHERE UNIV_ID=S.UNIV_ID);
```

Раздел 2.13

1. В результат первого запроса будут включены студенты, у которых не проставлены оценки по дисциплинам (**NULL** в поле MARK вместо оценки), тогда как второй запрос студентов с **NULL**-значениями в поле MARK подсчитывать не будет.

Раздел 2.14

1.
```
SELECT * FROM UNIVERSITY
  WHERE RATING >= ANY (SELECT RATING
    FROM UNIVERSITY WHERE UNIV_NAME = 'ВГУ');
```

2.
```
SELECT * FROM STUDENT
  WHERE CITY <> ALL (SELECT CITY
                     FROM UNIVERSITY);

SELECT * FROM STUDENT
  WHERE NOT CITY = ANY (SELECT CITY
                        FROM UNIVERSITY);
```

3.
```
SELECT SUBJ_ID FROM EXAM_MARKS
  WHERE MARK > ALL (SELECT MARK FROM EXAM_MARKS
                    WHERE SUBJ_ID = 105);
```

4.
```
SELECT SUBJ_ID FROM EXAM_MARKS
  WHERE MARK > (SELECT MAX(MARK) FROM EXAM_MARKS
                WHERE SUBJ_ID = 105);
```

Раздел 2.15

1.
```
SELECT SUBJ_NAME, SURNAME, NAME, KURS, SEMESTER
  FROM SUBJECT, STUDENT
  WHERE SEMESTER=KURS*2-1 OR SEMESTER=KURS*2;
```

2.
```
SELECT DISTINCT SURNAME, NAME
  FROM STUDENT S, EXAM_MARKS E
  WHERE S.STUDENT_ID=E.STUDENT_ID AND MARK>2;
```

То же с использованием несвязанного вложенного подзапроса с **IN**:

```
SELECT SURNAME, NAME FROM STUDENT
  WHERE STUDENT_ID IN (SELECT STUDENT_ID
                       FROM EXAM_MARKS
                       WHERE MARK>2);
```

Раздел 2.15.1

```
1. SELECT UNIV_NAME, KURS, COUNT (STUDENT_ID)
     FROM STUDENT S, UNIVERSITY U
     WHERE S.UNIV_ID=U.UNIV_ID
     GROUP BY UNIV_NAME, KURS;
2. SELECT SUBJ_NAME, SURNAME, NAME
     FROM LECTURER L, SUBJ_LECT SL, SUBJECT SB
     WHERE L.LECTURER_ID=SL.LECTURER_ID
       AND SL.SUBJ_ID=SB.SUBJ_ID;
3. SELECT SURNAME, NAME, SEMESTER, SUM(HOUR)
     FROM LECTURER L, SUBJ_LECT SL, SUBJECT SB
     WHERE L.LECTURER_ID=SL.LECTURER_ID
       AND SL.SUBJ_ID=SB.SUBJ_ID
     GROUP BY SURNAME, NAME, SEMESTER;
4. SELECT UNIV_NAME, SUBJ_NAME
     FROM UNIVERSITY U, LECTURER L, SUBJ_LECT SL,
          SUBJECT SB
     WHERE U.UNIV_ID=L.UNIV_ID
       AND L.LECTURER_ID=SL.LECTURER_ID
       AND SL.SUBJ_ID=SB.SUBJ_ID;
5. SELECT UNIV_NAME, SEMESTER, SUM(HOUR)
     FROM UNIVERSITY U, LECTURER L, SUBJ_LECT SL,
          SUBJECT SB
     WHERE U.UNIV_ID=L.UNIV_ID
       AND L.LECTURER_ID=SL.LECTURER_ID
       AND SL.SUBJ_ID=SB.SUBJ_ID
     GROUP BY UNIV_NAME, SEMESTER;
6. SELECT UNIV_NAME, SUBJ_NAME, SUM(HOUR)
     FROM UNIVERSITY U, LECTURER L, SUBJ_LECT SL,
          SUBJECT SB
     WHERE U.UNIV_ID=L.UNIV_ID
       AND L.LECTURER_ID=SL.LECTURER_ID
       AND SL.SUBJ_ID=SB.SUBJ_ID
     GROUP BY UNIV_NAME, SUBJ_NAME;
7. SELECT SURNAME, NAME, SUBJ_NAME, SUM(HOUR)
     FROM LECTURER L, SUBJ_LECT SL, SUBJECT SB
     WHERE L.LECTURER_ID=SL.LECTURER_ID
       AND SL.SUBJ_ID=SB.SUBJ_ID
     GROUP BY SURNAME, NAME, SUBJ_NAME;
8. SELECT UNIV_NAME, MAX(STIPEND)
     FROM STUDENT S, UNIVERSITY U
     WHERE S.UNIV_ID=U.UNIV_ID
     GROUP BY UNIV_NAME
     ORDER BY 2;
```

```
 9. SELECT UNIV_NAME, SURNAME, NAME, BIRTHDAY
      FROM STUDENT S, UNIVERSITY U
      WHERE S.UNIV_ID=U.UNIV_ID
        AND BIRTHDAY=(SELECT MAX(BIRTHDAY)
                      FROM STUDENT
                      WHERE UNIV_ID=U.UNIV_ID);
10. SELECT UNIV_NAME, SURNAME, NAME, STIPEND
      FROM STUDENT S, UNIVERSITY U
      WHERE S.UNIV_ID=U.UNIV_ID
       AND STIPEND=(SELECT MAX(STIPEND) FROM STUDENT
                    WHERE UNIV_ID=U.UNIV_ID);
11. SELECT SURNAME, NAME, SUBJ_NAME, MARK
      FROM STUDENT ST, EXAM_MARKS E, SUBJECT SB
      WHERE ST.STUDENT_ID=E.STUDENT_ID
        AND E.SUBJ_ID=SB.SUBJ_ID;
12. SELECT SUBJ_NAME, SURNAME, NAME, MARK
      FROM STUDENT ST, EXAM_MARKS E, SUBJECT SB
      WHERE ST.STUDENT_ID=E.STUDENT_ID
        AND E.SUBJ_ID=SB.SUBJ_ID
        AND MARK=(SELECT MAX(MARK) FROM EXAM_MARKS
                  WHERE SUBJ_ID=SB.SUBJ_ID);
13. SELECT SUBJ_NAME, SURNAME, NAME, EXAM_DATE
      FROM STUDENT ST, EXAM_MARKS E, SUBJECT SB
      WHERE ST.STUDENT_ID=E.STUDENT_ID
        AND E.SUBJ_ID=SB.SUBJ_ID
        AND EXAM_DATE=(SELECT MAX(EXAM_DATE)
                       FROM EXAM_MARKS
                       WHERE SUBJ_ID=SB.SUBJ_ID);
14. SELECT SUBJ_NAME, SURNAME, NAME, EXAM_DATE, MARK
      FROM STUDENT ST, EXAM_MARKS E, SUBJECT SB
      WHERE ST.STUDENT_ID=E.STUDENT_ID
        AND E.SUBJ_ID=SB.SUBJ_ID
        AND MARK>2
        AND EXAM_DATE=(SELECT MIN(EXAM_DATE)
                       FROM EXAM_MARKS
                       WHERE SUBJ_ID=SB.SUBJ_ID);
15. SELECT * FROM LECTURER L, SUBJ_LECT S
      WHERE L.LECTURER_ID=S.LECTURER_ID
        AND EXISTS (SELECT * FROM SUBJ_LECT
                    WHERE LECTURER_ID=S.LECTURER_ID
                      AND SUBJ_ID<>S.SUBJ_ID);
16. SELECT * FROM LECTURER L, SUBJ_LECT S
      WHERE L.LECTURER_ID=S.LECTURER_ID
        AND NOT EXISTS (SELECT * FROM SUBJ_LECT
                    WHERE LECTURER_ID=S.LECTURER_ID
                      AND SUBJ_ID<>S.SUBJ_ID);
```

```
17. SELECT SURNAME, NAME, S.STUDENT_ID
      FROM STUDENT S, EXAM_MARKS E
      WHERE S.STUDENT_ID=E.STUDENT_ID
        AND EXISTS (SELECT * FROM EXAM_MARKS
                    WHERE S.STUDENT_ID=E.STUDENT_ID
                       AND SUBJ_ID=E.SUBJ_ID
                       AND EXAM_ID<>E.EXAM_ID);
18. SELECT UNIV_NAME, SURNAME, NAME
      FROM UNIVERSITY U, STUDENT S
      WHERE U.UNIV_ID=S.UNIV_ID
        AND EXISTS (SELECT * FROM EXAM_MARKS
                    WHERE STUDENT_ID=S.STUDENT_ID
                       AND MARK<3);
19. SELECT DISTINCT SURNAME, NAME
      FROM STUDENT S, EXAM_MARKS E
      WHERE S.STUDENT_ID=E.STUDENT_ID
      AND MARK=5;
```

Тот же результат с использованием связанного подзапроса с **EXISTS**:

```
    SELECT SURNAME, NAME FROM STUDENT S
      WHERE EXISTS
      (SELECT * FROM EXAM_MARKS
       WHERE S.STUDENT_ID=STUDENT_ID
       AND MARK=5);
20. SELECT * FROM SUBJECT S, SUBJ_LECT SL, LECTURER L
      WHERE S.SUBJ_ID=SL.SUBJ_ID
        AND SL.LECTURER_ID=L.LECTURER_ID
        AND SURNAME='Колесников';
```

Тот же результат с использованием несвязанного подзапроса с **IN**:

```
    SELECT * FROM SUBJECT
      WHERE SUBJ_ID IN (SELECT SUBJ_ID
                  FROM SUBJ_LECT SL, LECTURER L
                  WHERE SL.LECTURER_ID=L.LECTURER_ID
                  AND SURNAME='Колесников');
21. SELECT DISTINCT SURNAME, NAME
      FROM LECTURER L, SUBJ_LECT SL, SUBJECT S
      WHERE L.LECTURER_ID=SL.LECTURER_ID
        AND SL.SUBJ_ID=S.SUBJ_ID
        AND SEMESTER IN (1, 2);
```

Тот же результат с использованием несвязанного подзапроса с **IN**:

```
    SELECT SURNAME, NAME FROM LECTURER
      WHERE LECTURER_ID
        IN (SELECT SL.LECTURER_ID
            FROM SUBJECT S, SUBJ_LECT SL
            WHERE SL.SUBJ_ID=S.SUBJ_ID
               AND SEMESTER<3);
```

```
22. SELECT DISTINCT SURNAME, NAME
      FROM LECTURER L, SUBJ_LECT SL, SUBJECT S
      WHERE L.LECTURER_ID=SL.LECTURER_ID
        AND SL.SUBJ_ID=S.SUBJ_ID
        AND EXISTS (SELECT *
                    FROM SUBJ_LECT SL1, SUBJECT S1
                       WHERE SL1.SUBJ_ID=S1.SUBJ_ID
                       AND LECTURER_ID=L.LECTURER_ID
                       AND SEMESTER<>S.SEMESTER);
23. SELECT SUBJ_NAME FROM SUBJECT S, SUBJ_LECT L
      WHERE L.SUBJ_ID=S.SUBJ_ID
        AND EXISTS (SELECT * FROM SUBJ_LECT
                    WHERE SUBJ_ID=L.SUBJ_ID
                    AND LECTURER_ID<>L.LECTURER_ID);
24. SELECT SUM(HOUR)
      FROM LECTURER L, SUBJ_LECT SL, SUBJECT SB
      WHERE L.LECTURER_ID=SL.LECTURER_ID
        AND SL.SUBJ_ID=SB.SUBJ_ID
        AND SURNAME='Лагутин';
25. SELECT DISTINCT SURNAME
      FROM LECTURER L, SUBJ_LECT SL, SUBJECT S
      WHERE L.LECTURER_ID=SL.LECTURER_ID
        AND SL.SUBJ_ID=S.SUBJ_ID
    GROUP BY SURNAME
    HAVING SUM(HOUR)>(SELECT SUM(HOUR)
      FROM LECTURER L1, SUBJ_LECT SL1, SUBJECT S1
      WHERE L1.LECTURER_ID=SL1.LECTURER_ID
        AND SL1.SUBJ_ID=S1.SUBJ_ID
        AND SURNAME='Николаев');
26. SELECT SURNAME FROM LECTURER L, UNIVERSITY U
      WHERE L.UNIV_ID=U.UNIV_ID AND RATING<200;
27. SELECT SUM(HOUR)
      FROM LECTURER L, SUBJ_LECT SL, SUBJECT S,
           UNIVERSITY U
      WHERE L.LECTURER_ID=SL.LECTURER_ID
        AND SL.SUBJ_ID=S.SUBJ_ID
        AND L.UNIV_ID=U.UNIV_ID
        AND SEMESTER<3 AND UNIV_NAME='ВГУ';
28. SELECT AVG(HOUR)
      FROM LECTURER L, SUBJ_LECT SL,
           SUBJECT S, UNIVERSITY U
      WHERE L.LECTURER_ID=SL.LECTURER_ID
        AND SL.SUBJ_ID=S.SUBJ_ID
        AND L.UNIV_ID=U.UNIV_ID
        AND SEMESTER IN (3, 4) AND UNIV_NAME='ВГУ';
```

29.
```
SELECT S.STUDENT_ID, SURNAME, KURS
  FROM STUDENT S, (SELECT STUDENT_ID, SEMESTER
                   FROM EXAM_MARKS E, SUBJECT S
                   WHERE E.SUBJ_ID=S.SUBJ_ID
                     AND MARK=5
GROUP BY STUDENT_ID, SEMESTER
HAVING COUNT(MARK)>1) SEM
  WHERE S.STUDENT_ID=SEM.STUDENT_ID
GROUP BY S.STUDENT_ID, SURNAME, KURS
HAVING COUNT(DISTINCT SEMESTER)=2*KURS
  AND MAX(SEMESTER)=2*KURS;
```

30. Примеры возможных вариантов запросов:

```
SELECT DISTINCT SURNAME
  FROM STUDENT S, EXAM_MARKS EM, SUBJECT SB
  WHERE S.STUDENT_ID=EM.STUDENT_ID
    AND EM.SUBJ_ID=SB.SUBJ_ID
    AND SUBJ_NAME='Информатика';
```

```
SELECT DISTINCT SURNAME
  FROM STUDENT S JOIN EXAM_MARKS EM
  ON S.STUDENT_ID=EM.STUDENT_ID
  JOIN SUBJECT SB ON EM.SUBJ_ID=SB.SUBJ_ID
  WHERE SUBJ_NAME='Информатика';
```

```
SELECT SURNAME FROM STUDENT S
  WHERE EXISTS (SELECT *
                FROM EXAM_MARKS EM, SUBJECT SB
                WHERE S.STUDENT_ID=STUDENT_ID
                  AND EM.SUBJ_ID=SB.SUBJ_ID
                  AND SUBJ_NAME='Информатика');
```

```
SELECT SURNAME FROM STUDENT
  WHERE STUDENT_ID IN (SELECT STUDENT_ID
                FROM EXAM_MARKS EM, SUBJECT SB
                WHERE EM.SUBJ_ID=SB.SUBJ_ID
                  AND SUBJ_NAME='Информатика');
```

```
SELECT DISTINCT SURNAME
  FROM STUDENT S, EXAM_MARKS EM
  WHERE S.STUDENT_ID=EM.STUDENT_ID
    AND SUBJ_ID=(SELECT SUBJ_ID FROM SUBJECT
                 WHERE SUBJ_NAME='Информатика');
```

31. Примеры возможных вариантов запросов:

```
SELECT DISTINCT SURNAME
  FROM LECTURER L, SUBJ_LECT SL, SUBJECT SB
  WHERE L.LECTURER_ID=SL.LECTURER_ID
    AND SL.SUBJ_ID=SB.SUBJ_ID
    AND SUBJ_NAME='Информатика';
```

```
SELECT DISTINCT SURNAME
  FROM LECTURER L JOIN SUBJ_LECT SL
  ON L.LECTURER_ID=SL.LECTURER_ID
  JOIN SUBJECT SB ON SL.SUBJ_ID=SB.SUBJ_ID
  WHERE SUBJ_NAME='Информатика';

SELECT SURNAME FROM LECTURER L
  WHERE EXISTS (SELECT *
                FROM SUBJ_LECT SL, SUBJECT SB
                WHERE L.LECTURER_ID=LECTURER_ID
                  AND SL.SUBJ_ID=SB.SUBJ_ID
                  AND SUBJ_NAME='Информатика');

SELECT SURNAME FROM LECTURER
  WHERE LECTURER_ID IN (SELECT LECTURER_ID
                 FROM SUBJ_LECT SL, SUBJECT SB
                 WHERE SL.SUBJ_ID=SB.SUBJ_ID
                   AND SUBJ_NAME='Информатика');

SELECT DISTINCT SURNAME
  FROM LECTURER L, SUBJ_LECT SL
  WHERE L.LECTURER_ID=SL.LECTURER_ID
    AND SUBJ_ID=(SELECT SUBJ_ID FROM SUBJECT
                 WHERE SUBJ_NAME='Информатика');
```

Раздел 2.15.2

```
1. SELECT SURNAME, SUBJ_ID
     FROM STUDENT, EXAM_MARKS
     WHERE STUDENT.STUDENT_ID=EXAM_MARKS.STUDENT_ID;

   SELECT SURNAME, SUBJ_ID
     FROM STUDENT JOIN EXAM_MARKS
     ON STUDENT.STUDENT_ID = EXAM_MARKS.STUDENT_ID;
2. SELECT SURNAME, SUBJ_ID
     FROM STUDENT LEFT OUTER JOIN EXAM_MARKS
     ON STUDENT.STUDENT_ID = EXAM_MARKS.STUDENT_ID;

   SELECT SURNAME, SUBJ_ID
     FROM STUDENT, EXAM_MARKS
     WHERE STUDENT.STUDENT_ID =
           EXAM_MARKS.STUDENT_ID(+);
3. SELECT SURNAME, SUBJ_NAME
     FROM STUDENT, SUBJECT, EXAM_MARKS
     WHERE STUDENT.STUDENT_ID =
           EXAM_MARKS.STUDENT_ID
       AND SUBJECT.SUBJ_ID = EXAM_MARKS.SUBJ_ID;
4. SELECT SURNAME, SUBJ_NAME
     FROM STUDENT LEFT OUTER JOIN EXAM_MARKS
```

```
    ON STUDENT.STUDENT_ID=EXAM_MARKS.STUDENT_ID
    LEFT OUTER JOIN SUBJECT
    ON SUBJECT.SUBJ_ID=EXAM_MARKS.SUBJ_ID;

SELECT SURNAME, SUBJ_NAME
  FROM (SELECT * FROM STUDENT, EXAM_MARKS
  WHERE STUDENT.STUDENT_ID=
        EXAM_MARKS.STUDENT_ID(+)) T, SUBJECT
  WHERE T.SUBJ_ID=SUBJECT.SUBJ_ID(+);
```

5.
```
SELECT SUBJ_NAME, SURNAME, MARK
  FROM SUBJECT, STUDENT, EXAM_MARKS
  WHERE EXAM_MARKS.STUDENT_ID=STUDENT.STUDENT_ID
    AND SUBJECT.SUBJ_ID = EXAM_MARKS.SUBJ_ID
    AND MARK >= 4;

SELECT SUBJ_NAME, SURNAME, MARK
  FROM SUBJECT JOIN EXAM_MARKS
  ON SUBJECT.SUBJ_ID=EXAM_MARKS.SUBJ_ID
  JOIN STUDENT
  ON EXAM_MARKS.STUDENT_ID=STUDENT.STUDENT_ID
  WHERE MARK>=4;
```

6.
```
SELECT UNIV_NAME, MAX(STIPEND)
  FROM STUDENT, UNIVERSITY
  WHERE UNIVERSITY.UNIV_ID = STUDENT.UNIV_ID
    AND RATING > 300
  GROUP BY UNIV_NAME;

SELECT UNIV_NAME, MAX(STIPEND)
  FROM STUDENT JOIN UNIVERSITY
  ON UNIVERSITY.UNIV_ID = STUDENT.UNIV_ID
    AND RATING > 300
  GROUP BY UNIV_NAME;
```

7.
```
SELECT SURNAME, RATING
  FROM STUDENT LEFT OUTER JOIN UNIVERSITY
  ON STUDENT.UNIV_ID = UNIVERSITY.UNIV_ID
ORDER BY SURNAME;

SELECT SURNAME, RATING
  FROM STUDENT, UNIVERSITY
  WHERE STUDENT.UNIV_ID = UNIVERSITY.UNIV_ID(+)
ORDER BY SURNAME;
```

8.
```
SELECT * FROM STUDENT S
  LEFT OUTER JOIN UNIVERSITY U
  ON S.UNIV_ID=U.UNIV_ID;

SELECT * FROM STUDENT WHERE UNIV_ID IS NULL;
```

9.
```
SELECT * FROM UNIVERSITY U
  LEFT OUTER JOIN STUDENT S
  ON S.UNIV_ID=U.UNIV_ID;
```

```
   SELECT * FROM UNIVERSITY U WHERE NOT EXISTS
     (SELECT * FROM STUDENT
      WHERE UNIV_ID=U.UNIV_ID);
10. SELECT * FROM UNIVERSITY U
      LEFT OUTER JOIN LECTURER L
      ON L.UNIV_ID=U.UNIV_ID;

   SELECT * FROM UNIVERSITY U WHERE NOT EXISTS
     (SELECT * FROM LECTURER
      WHERE UNIV_ID=U.UNIV_ID);
11. SELECT * FROM LECTURER L
      LEFT OUTER JOIN UNIVERSITY U
      ON L.UNIV_ID=U.UNIV_ID;

   SELECT * FROM LECTURER WHERE UNIV_ID IS NULL;
12. SELECT * FROM STUDENT S
      LEFT OUTER JOIN EXAM_MARKS E
      ON S.STUDENT_ID=E.STUDENT_ID;

   SELECT * FROM STUDENT S WHERE NOT EXISTS
     (SELECT * FROM EXAM_MARKS
      WHERE STUDENT_ID=S.STUDENT_ID);
13. SELECT * FROM EXAM_MARKS E
      LEFT OUTER JOIN STUDENT S
      ON S.STUDENT_ID=E.STUDENT_ID;

   SELECT * FROM EXAM_MARKS
     WHERE STUDENT_ID IS NULL;
14. SELECT * FROM SUBJECT S
      LEFT OUTER JOIN EXAM_MARKS E
      ON S.SUBJ_ID=E.SUBJ_ID;

   SELECT * FROM SUBJECT S WHERE NOT EXISTS
     (SELECT * FROM EXAM_MARKS
      WHERE SUBJ_ID=S.SUBJ_ID);
15. SELECT * FROM EXAM_MARKS E
      LEFT OUTER JOIN SUBJECT S
      ON S.SUBJ_ID=E.SUBJ_ID;

   SELECT * FROM EXAM_MARKS WHERE SUBJ_ID IS NULL;
```

Раздел 2.15.3

```
1. SELECT A.SURNAME, B.SURNAME
     FROM STUDENT A, STUDENT B
     WHERE A.CITY = B.CITY
       AND A.STUDENT_ID < B.STUDENT_ID;
2. SELECT A.UNIV_NAME, B.UNIV_NAME
     FROM UNIVERSITY A, UNIVERSITY B
```

```
      WHERE A.CITY = B.CITY
        AND A.UNIV_NAME < B.UNIV_NAME;
```

3.
```
SELECT A.UNIV_NAME, A.CITY
  FROM UNIVERSITY A, UNIVERSITY B
  WHERE A.RATING >= B.RATING
    AND B.UNIV_NAME = 'ВГУ';
```

4.
```
SELECT DISTINCT E1.STUDENT_ID
  FROM EXAM_MARKS E1, EXAM_MARKS E2
  WHERE E1.MARK=E2.MARK
    AND E2.STUDENT_ID=12;
```

То же с использованием несвязанного подзапроса с **IN**:

```
SELECT DISTINCT STUDENT_ID FROM EXAM_MARKS
  WHERE MARK IN (SELECT MARK FROM EXAM_MARKS
                 WHERE STUDENT_ID=12);
```

5.
```
SELECT DISTINCT S1.LECTURER_ID, S2.LECTURER_ID
  FROM SUBJ_LECT S1, SUBJ_LECT S2
  WHERE S1.SUBJ_ID=S2.SUBJ_ID
    AND S1.LECTURER_ID<S2.LECTURER_ID;
```

Раздел 2.16

1.
```
SELECT UNIV_NAME, CITY, RATING, 'высокий'
  FROM UNIVERSITY
  WHERE RATING >= 300
UNION
  SELECT UNIV_NAME, CITY, RATING, 'низкий'
  FROM UNIVERSITY
  WHERE RATING < 300;
```

2.
```
SELECT SURNAME, 'успевает'
  FROM STUDENT A
  WHERE 3 <= ALL
    (SELECT MARK
     FROM EXAM_MARKS B
     WHERE A.STUDENT_ID = B.STUDENT_ID)
UNION
  SELECT SURNAME, 'не успевает'
  FROM STUDENT A
  WHERE 2 = ANY
    (SELECT MARK
     FROM EXAM_MARKS B
     WHERE A.STUDENT_ID = B.STUDENT_ID)
UNION
  SELECT SURNAME, 'не сдавал'
  FROM STUDENT A
  WHERE NOT EXIST
    (SELECT MARK
```

```
        FROM EXAM_MARKS B
        WHERE A.STUDENT_ID = B.STUDENT_ID)
  ORDER BY 1;
3. SELECT NAME, SURNAME, 'студент'
     FROM STUDENT
     WHERE CITY = 'Москва'
   UNION
     SELECT NAME, SURNAME, 'преподаватель'
     FROM LECTURER
     WHERE CITY = 'Москва';
4. SELECT NAME, SURNAME, 'студент'
     FROM STUDENT A, UNIVERSITY B
     WHERE A.UNIV_ID = B.UNIV_ID
       AND UNIV_NAME = 'ВГУ'
   UNION
     SELECT NAME, SURNAME, 'преподаватель'
     FROM STUDENT A, UNIVERSITY B
     WHERE A.UNIV_ID = B.UNIV_ID
       AND UNIV_NAME = 'ВГУ';
5. SELECT CITY, UNIV_NAME, RATING, 'max'
     FROM UNIVERSITY U
     WHERE RATING=(SELECT MAX(RATING)
                   FROM UNIVERSITY
                   WHERE CITY=U.CITY)
   UNION
     SELECT CITY, UNIV_NAME, RATING, 'min'
     FROM UNIVERSITY U
     WHERE RATING=(SELECT MIN(RATING)
                   FROM UNIVERSITY
                   WHERE CITY=U.CITY);
6. SELECT KURS, SURNAME, NAME, STIPEND, 'max'
     FROM STUDENT S
     WHERE STIPEND=(SELECT MAX(STIPEND)
                    FROM STUDENT
                    WHERE KURS=S.KURS)
   UNION
     SELECT KURS, SURNAME, NAME, STIPEND, 'min'
     FROM STUDENT S
     WHERE STIPEND=(SELECT MIN(STIPEND)
                    FROM STUDENT
                    WHERE KURS=S.KURS);
7. SELECT KURS, SURNAME, NAME, BIRTHDAY, 'младший'
     FROM STUDENT S
     WHERE BIRTHDAY=(SELECT MAX(BIRTHDAY)
                     FROM STUDENT
                     WHERE KURS=S.KURS)
```

```
   UNION
   SELECT KURS, SURNAME, NAME, BIRTHDAY, 'старший'
     FROM STUDENT S
     WHERE BIRTHDAY=(SELECT MIN(BIRTHDAY)
                     FROM STUDENT
                     WHERE KURS=S.KURS);
```

8.
```
   SELECT UNIV_NAME, SURNAME, NAME
     FROM UNIVERSITY U JOIN STUDENT S
     ON S.UNIV_ID=U.UNIV_ID
   UNION ALL
   SELECT UNIV_NAME, 'Студентов нет', NULL
     FROM UNIVERSITY U WHERE NOT EXISTS
     (SELECT * FROM STUDENT
      WHERE UNIV_ID=U.UNIV_ID);
```

Если тип атрибута SURNAME — строка фиксированной длины, то необходимо фразу 'Студентов нет' соответственно дополнить пробелами.

9.
```
   SELECT UNIV_NAME, SURNAME, NAME
     FROM UNIVERSITY U JOIN LECTURER L
     ON L.UNIV_ID=U.UNIV_ID
   UNION ALL
   SELECT UNIV_NAME, 'Преподавателей нет', NULL
     FROM UNIVERSITY U WHERE NOT EXISTS
     (SELECT * FROM LECTURER
      WHERE UNIV_ID=U.UNIV_ID);
```

Если тип атрибута SURNAME — строка фиксированной длины, то необходимо фразу 'Преподавателей нет' соответственно дополнить пробелами.

10.
```
   SELECT SURNAME, NAME, MARK
     FROM STUDENT S JOIN EXAM_MARKS E
     ON S.STUDENT_ID=E.STUDENT_ID
   UNION ALL
   SELECT SURNAME, NAME, 0
     FROM STUDENT S WHERE NOT EXISTS
     (SELECT * FROM EXAM_MARKS
      WHERE STUDENT_ID=S.STUDENT_ID);
```

Раздел 3.1

1.
```
   INSERT INTO
     SUBJECT (SEMESTER, SUBJ_NAME, HOUR, SUBJ_ID)
     VALUES (4, 'Алгебра', 72, 201);
```

2.
```
   INSERT INTO STUDENT (STUDENT_ID, SURNAME, NAME,
                        KURS, CITY, UNIV_ID)
     VALUES (101, 'Орлов', 'Николай',
             1, 'Воронеж', 10);
```

```
3. DELETE FROM EXAM_MARKS WHERE STUDENT_ID = 100;
4. UPDATE UNIVERSITY
     SET RATING = RATING + 5
     WHERE CITY = 'Санкт-Петербург';
5. UPDATE STUDENT
     SET CITY = 'Воронеж'
     WHERE SURNAME = 'Иванов';
```

Раздел 3.2.3

```
1. INSERT INTO STUDENT1
     SELECT *
     FROM STUDENT
     WHERE 5 <
        (SELECT COUNT (SUBJ_ID)
         FROM EXAM_MARKS
         WHERE EXAM_MARKS.STUDENT_ID =
               STUDENT.STUDENT_ID
           AND MARK > 2);
2. DELETE FROM SUBJECT1
     WHERE 0 =
     (SELECT COUNT(MARK)
      FROM EXAM_MARKS
      WHERE EXAM_MARKS.SUBJ_ID = SUBJECT.SUBJ_ID);
3. UPDATE STUDENT
     SET STIPEND = STIPEND * 1.2
     WHERE MARK IS NOT NULL AND 50 <
     (SELECT SUM (MARK)
      FROM EXAM_MARKS
      WHERE EXAM_MARKS.STUDENT_ID =
            STUDENT.STUDENT_ID);
```

Раздел 4.4

```
1. CREATE TABLE LECTURER1
     (LECTURER_ID INTEGER NOT NULL UNIQUE,
      SURNAME CHAR(25) NOT NULL,
      NAME CHAR(10) NOT NULL,
      CITY CHAR(15) DEFAULT 'Воронеж',
      UNIV_ID INTEGER);
2. CREATE TABLE SUBJECT1
     (SUBJECT_ID INTEGER NOT NULL UNIQUE,
      SUBJ_NAME CHAR(20),
      HOUR DECIMAL (4),
      SEMESTER SMALLINT);
3. CREATE TABLE UNIVERSITY1
     (UNIV_ID INTEGER NOT NULL UNIQUE,
```

```
      UNIV_NAME CHAR(30) NOT NULL UNIQUE,
      RATING DECIMAL (4),
      CITY CHAR(15));
4. CREATE TABLE EXAM_MARKS1
     (EXAM_ID INTEGER NOT NULL,
      STUDENT_ID INTEGER NOT NULL,
      SUBJ_ID INTEGER NOT NULL,
      MARK INTEGER,
      EXAM_DATE DATE);
5. CREATE TABLE SUBJ_LECT1
     (LECTURER_ID INTEGER NOT NULL,
      SUBJ_ID INTEGER NOT NULL);
6. CREATE INDEX STUD ON STUDENT(KURS);
7. CREATE INDEX MARK1 ON EXAM_MARKS(EXAM_DATE);
```

Раздел 4.5.9

```
1. CREATE TABLE EXAM_MARKS
     (EXAM_ID INTEGER NOT NULL,
      SUBJ_ID INTEGER NOT NULL,
      STUDENT_ID INTEGER NOT NULL,
      MARK INTEGER,
      EXAM_DATE DATE NOT NULL,
      CONSTRAINT EXAM_MARKS_CONSTR_1
        UNIQUE (EXAM_ID, SUBJ_ID, STUDENT_ID),
      CONSTRAINT EXAM_MARKS_CONSTR_2
        UNIQUE (EXAM_ID, EXAM_DATE);
2. CREATE TABLE SUBJECT
     (SUBJ_ID INTEGER PRIMARY KEY,
      SUBJ_NAME CHAR(25),
      HOUR INTEGER DEFAULT 36 NOT NULL,
      SEMESTER INTEGER CHECK (SEMESTER
                              BETWEEN 1 AND 12));
3. CREATE TABLE EXAM_MARKS
     (EXAM_ID INTEGER NOT NULL,
      SUBJ_ID INTEGER NOT NULL,
      STUDENT_ID INTEGER NOT NULL,
   CONSTRAINT EXAM_MARKS_CHECK
      CHECK (((EXAM_ID > SUBJ_ID) AND
              (SUBJ_ID > STUDENT_ID)));
```

Раздел 4.6.10

```
1. CREATE TABLE SUBJECT_1
     (SUBJECT_ID  INTEGER PRIMARY KEY,
      SUBJ_NAME   CHAR(20),
```

```
      HOUR          DECIMAL (4),
      SEMESTER      SMALLINT);
2. CREATE TABLE SUBJ_LECT_1
     (LECTURER_ID INTEGER NOT NULL
       REFERENCES LECTURER_1,
      SUBJ_ID INTEGER NOT NULL
       REFERENCES SUBJECT_1,
   CONSTRAINT SUBJ_LECT
     PRIMARY KEY (LECTURER_ID, SUBJ_ID));
3. CREATE TABLE SUBJ_LECT_1
     (LECTURER_ID INTEGER NOT NULL,
      SUBJ_ID INTEGER NOT NULL,
   CONSTRAINT SUBJ_LECT
     PRIMARY KEY (LECTURER_ID, SUBJ_ID),
   CONSTRAINT LECT_ID_FOR_KEY
     FOREIGN KEY (LECTURER_ID)
     REFERENCES LECTURER_1 ON UPDATE NO ACTION
                           ON DELETE NO ACTION,
   CONSTRAINT SUBJ_ID_FOR_KEY FOREIGN KEY (SUBJ_ID)
     REFERENCES SUBJECT_1 ON UPDATE NO ACTION
                          ON DELETE NO ACTION);
4. CREATE TABLE LECTURER_1
     (LECTURER_ID  INTEGER PRIMARY KEY,
      SURNAME      CHAR(25),
      NAME         CHAR(10),
      CITY         CHAR(15),
      UNIV_ID      INTEGER,
   CONSTRAINT UNIV_ID _FOR_KEY FOREIGN KEY (UNIV_ID)
     REFERENCES UNIVERSITY_1 ON UPDATE CASCADE
                             ON DELETE CASCADE);
5. CREATE TABLE UNIVERSITY_1
     (UNIV_ID    INTEGER PRIMARY KEY,
      UNIV_NAME  CHAR(30),
      RATING     DECIMAL (4),
      CITY       CHAR(15));
6. CREATE TABLE EXAM_MARKS_1
     (EXAM_ID     INTEGER,
      SUBJ_ID     INTEGER,
      STUDENT_ID  INTEGER,
      MARK        INTEGER,
      SUBJ_NAME   CHAR(15),
      SURNAME     CHAR(25),
   CONSTRAINT EXAM_MARKS_PR_KEY
        PRIMARY KEY (EXAM_ID, SUBJ_ID, STUDENT_ID ),
   CONSTRAINT EX_M_FOR_KEY_1 FOREIGN KEY (SUBJ_ID)
```

```
      REFERENCES SUBJECT_1 ON UPDATE CASCADE
                           ON DELETE NO ACTION,
  CONSTRAINT EX_M_FOR_KEY_2
      FOREIGN KEY (STUDENT_ID)
      REFERENCES STUDENT_1 ON UPDATE CASCADE
                           ON DELETE NO ACTION);
```

7.
```
CREATE TABLE STUDENT_1
   (STUDENT_ID      INTEGER PRIMARY KEY,
    SURNAME         CHAR(25),
    NAME            CHAR(10),
    STIPEND         INTEGER,
    KURS            INTEGER,
    CITY            CHAR(15),
    BIRTHDAY        DATE,
    UNIV_ID         INTEGER
       REFERENCES UNIVERSITY(UNIV_ID),
    SENIOR_STUDENT  INTEGER
       REFERENCES STUDENT(STUDENT_ID));
```

8.
```
CREATE TABLE STUDENT_1
   (STUDENT_ID  INTEGER PRIMARY KEY,
    SURNAME     CHAR(25),
    NAME        CHAR(10),
    STIPEND     INTEGER,
    KURS        INTEGER,
    CITY        CHAR(15),
    BIRTHDAY    DATE,
    UNIV_ID     INTEGER
      REFERENCES UNIVERSITY ON DELETE SET NULL);
```

9.
```
CREATE TABLE STUDENT
   (STUDENT_ID  INTEGER,
    SURNAME     CHAR(25),
    NAME        CHAR(10),
    STIPEND     INTEGER,
    KURS        INTEGER,
    CITY        CHAR(15),
    BIRTHDAY    DATE,
    UNIV_ID     INTEGER,
  CONSTRAINT PK_STUDENT PRIMARY KEY(STUDENT_ID));

  CREATE TABLE LECTURER
     (LECTURER_ID  INTEGER,
      SURNAME      CHAR(25),
      NAME         CHAR(10),
      CITY         CHAR(15),
      UNIV_ID      INTEGER,
  CONSTRAINT PK_LECTURER PRIMARY KEY(LECTURER_ID));
```

```
CREATE TABLE SUBJECT
   (SUBJ_ID     INTEGER,
    SUBJ_NAME   CHAR(10),
    HOUR        INTEGER,
    SEMESTER    INTEGER,
CONSTRAINT PK_SUBJECT PRIMARY KEY(SUBJ_ID));

CREATE TABLE UNIVERSITY
   (UNIV_ID     INTEGER,
    UNIV_NAME   CHAR(10),
    RATING      INTEGER,
    CITY        CHAR(15),
CONSTRAINT PK_UNIVERSITY PRIMARY KEY(UNIV_ID));

CREATE TABLE EXAM_MARKS
   (EXAM_ID      INTEGER NOT NULL,
    STUDENT_ID   INTEGER NOT NULL,
    SUBJ_ID      INTEGER NOT NULL,
    MARK         INTEGER,
    EXAM_DATE    DATE NOT NULL,
CONSTRAINT PK_EXAM_MARKS
  PRIMARY KEY (EXAM_ID, STUDENT_ID, SUBJ_ID));

CREATE TABLE SUBJ_LECT
   (LECTURER_ID  INTEGER NOT NULL,
    SUBJ_ID      INTEGER NOT NULL,
CONSTRAINT PK_SUBJ_LECT
  PRIMARY KEY (LECTURER_ID, SUBJ_ID));

ALTER TABLE STUDENT
  ADD CONSTRAINT FK_STUDENT FOREIGN KEY(UNIV_ID)
    REFERENCES UNIVERSITY (UNIV_ID);

ALTER TABLE LECTURER
  ADD CONSTRAINT FK_LECTURER
    FOREIGN KEY (UNIV_ID)
    REFERENCES UNIVERSITY (UNIV_ID);

ALTER TABLE EXAM_MARKS
  ADD CONSTRAINT FK_EXAM_MARKS_1
    FOREIGN KEY (STUDENT_ID)
    REFERENCES STUDENT (STUDENT_ID);

ALTER TABLE EXAM_MARKS
  ADD CONSTRAINT FK_EXAM_MARKS_2
    FOREIGN KEY (SUBJ_ID)
    REFERENCES SUBJECT (SUBJ_ID);
```

```
ALTER TABLE SUBJ_LECT
  ADD CONSTRAINT FK_SUBJ_LECT_1
    FOREIGN KEY (SUBJ_ID)
    REFERENCES SUBJECT (SUBJ_ID);

ALTER TABLE SUBJ_LECT
  ADD CONSTRAINT FK_SUBJ_LECT_2
    FOREIGN KEY (LECTURER_ID)
    REFERENCES LECTURER (LECTURER_ID);
```

Раздел 5.7

```
1. CREATE VIEW FIVE
     AS SELECT * FROM STUDENT A
     WHERE 5 = (SELECT MIN (MARK)
       FROM EXAM_MARKS B
       WHERE A.STUDENT_ID = B.STUDENT_ID);
2. CREATE VIEW KOL
     AS SELECT CITY, COUNT (STUDENT_ID) AS KOL_STUD
     FROM STUDENT
     GROUP BY CITY;
3. CREATE VIEW NOVSTUDENT
     AS SELECT EXAM_MARKS.STUDENT_ID,
               SURNAME, NAME,
       AVG(MARK) AS AVG_MARK, SUM(MARK) AS SUM_MARK
     FROM EXAM_MARKS, STUDENT
     WHERE STUDENT.STUDENT_ID =
           EXAM_MARKS.STUDENT_ID
     GROUP BY SURNAME, NAME, EXAM_MARKS.STUDENT_ID;
4. CREATE VIEW KOLEXAM
     AS SELECT EXAM_MARKS.STUDENT_ID,
               SURNAME, NAME,
       COUNT(EXAM_ID) AS KOL_EXAM
     FROM EXAM_MARKS, STUDENT
     WHERE STUDENT.STUDENT_ID =
           EXAM_MARKS.STUDENT_ID
     GROUP BY SURNAME, NAME, EXAM_MARKS.STUDENT_ID;
```

Раздел 5.9

1. б), в).

```
2. CREATE VIEW STIP
     AS SELECT STUDENT_ID, STIPEND
     FROM STUDENT
     WHERE STIPEND BETWEEN 100 AND 200
     WITH CHECK OPTION;
```

Раздел 6.9

```
1. GRANT UPDATE ON EXAM_MARKS TO PETROV;
2. GRANT SELECT ON EXAM_MARKS TO SIDOROV
     WITH GRANT OPTION;
3. REVOKE INSERT ON STUDENT FROM IVANOV;
4. CREATE VIEW FOR_SIDOROV
     AS SELECT *
        FROM UNIVERSITY
        WHERE RATING BETWEEN 300 AND 400
        WITH CHECK OPTION;

   GRANT INSERT, UPDATE ON FOR_SIDOROV TO SIDOROV;
5. CREATE VIEW FOR_PETROV
     AS SELECT *
        FROM EXAM_MARKS
        WHERE STUD_ID IN
        (SELECT STUD_ID
         FROM EXAM_MARKS
         WHERE MARK=2);

   GRANT SELECT ON EXAM_MARKS TO PETROV;

   REVOKE UPDATE ON FOR_PETROV TO PETROV;
```

Раздел 6.12

```
1. CREATE SYNONYM EXAM_MARKS FOR IVANOV.EXAM_MARKS;
2. CREATE PUBLIC SYNONYM EXAM_MARKS FOR EXAM_MARKS;
```

Приложение

ЗАДАЧИ ПО ПРОЕКТИРОВАНИЮ БД

В приложении приводятся тексты задач по проектированию баз данных, относящихся к различным предметным областям. Требуется в соответствии с условиями задач:

- сформировать структуру таблиц баз данных;
- подобрать подходящие имена таблицам и их полям;
- обеспечить требования нормализации таблиц баз данных (т. е. приведение к пятой нормальной форме);
- сформировать SQL-запросы для создания таблиц баз данных с указанием первичных и внешних ключей и необходимых ограничений, SQL-запросы для добавления, изменения и выборки необходимых данных.

При решении задач предполагается использование средств, позволяющих разрабатывать схемы баз данных, и приложений, работающих с базами данных (Power Designer, Oracle Developer, ERWin, Power Builder, Borland Delphi, C++ Builder и др.)

Задача 1. *Летопись острова Санта Белинда.*

Где-то в великом океане лежит воображаемый остров Санта Белинда. Вот уже триста лет ведется подробная летопись острова. В летопись заносятся и данные обо всех людях, хоть какое-то время проживавших на острове. Для каждого из островитян записываются его имя, пол, даты рождения и смерти. Хранятся там и имена их родителей, если известно, кто они. У некоторых отсутствуют сведения об отце, у некоторых — о матери, а часть людей, судя по записям, — круглые сироты. Из летописи можно узнать, когда был построен каждый дом, стоящий на острове (а если сейчас его уже нет, то когда он был снесен), точный адрес и подробный план этого дома, кто и когда в нем жил.

Точно так же, как и столетия назад, на острове действуют предприниматели, занимающиеся, в частности, ловлей рыбы, заготовкой сахарного тростника и табака. Большинство из них все делают сами, а некоторые нанимают работников, заключая с ними контракты разной

продолжительности. Имеются записи и о том, кто кого нанимал, на какую работу, когда начался и закончился контракт. Собственно, круг занятий жителей острова крайне узок и не меняется веками. Неудивительно поэтому, что в летописи подробно описывается каждое дело, будь то рыбная ловля или выпечка хлеба. Все предприниматели — уроженцы острова. Некоторые объединяются в кооперативы, и по записям можно установить, кто участвовал в деле, когда вступил и когда вышел из него, каким паем владел. Имеются краткие описания деятельности каждого предпринимателя или кооператива, сообщающие, в частности, когда было начато дело, когда и почему прекращено.

Предлагается сформировать систему нормализованных таблиц, в которых можно было бы хранить всю эту многообразную информацию. Подыщите выразительные имена для таблиц и полей, снабдив их при необходимости соответствующими пояснениями.

Задача 2. *База данных "Скачки".*

В информационной системе клуба любителей скачек должна быть представлена информация об участвующих в скачках лошадях (кличка, пол, дата рождения), их владельцах (имя, адрес, телефон) и жокеях (имя, адрес, дата рождения, рейтинг). Необходимо сформировать таблицы для хранения информации по каждому состязанию: дата, время и место проведения скачек (ипподром), название состязаний (если таковое имеется), номера заездов, клички участвующих в заездах лошадей и имена жокеев с указанием занятого места и показанного в заезде времени.

Задача 3. *База данных "Хроники восхождений" в альпинистском клубе.*

В базе данных должны записываться даты начала и завершения каждого восхождения, имена и адреса участвовавших в нем альпинистов, название и высота горы, страна и район, где эта гора расположена. Присвойте выразительные имена таблицам и полям для хранения указанной информации. Написать запросы, осуществляющие следующие операции.

1) Для введенного пользователем интервала дат показать список гор с указанием даты последнего восхождения. Для каждой горы сформировать в хронологическом порядке список групп, осуществлявших восхождение.

2) Предоставить возможность добавления новой вершины с указанием ее названия, высоты и страны местоположения.

3) Предоставить возможность изменения данных о вершине, если на нее не было восхождения.

4) Показать список альпинистов, осуществлявших восхождение в указанный интервал дат. Для каждого альпиниста вывести список гор, на которые он осуществлял восхождения в этот период, с указанием названия группы и даты восхождения.

5) Предоставить возможность добавления нового альпиниста в состав указанной группы.

6) Показать информацию о количестве восхождений каждого альпиниста на каждую гору. При выводе список отсортировать по количеству восхождений.

7) Показать список восхождений (групп), которые осуществлялись в указанный пользователем период времени. Для каждой группы показать ее состав.

8) Предоставить возможность добавления новой группы, указав ее название, вершину, время начала восхождения.

9) Предоставить информацию о том, сколько альпинистов побывало на каждой горе. Список отсортировать в алфавитном порядке по названию вершин.

Задача 4. *База данных медицинского кооператива.*

Базу данных использует для работы коллектив врачей. В таблицы должны быть занесены имя, пол, дата рождения и домашний адрес каждого их пациента. Всякий раз, когда врач осматривает больного (пришедшего на прием или на дому), фиксируется дата и место проведения осмотра, симптомы, диагноз и предписания больному, проставляется имя пациента и имя врача. Если врач прописывает больному какое-либо лекарство, в таблицу заносится название лекарства, способ его приема, словесное описание предполагаемого действия и возможных побочных эффектов.

Задача 5. *База данных "Городская Дума".*

В базе хранятся имена, адреса, домашние и служебные телефоны всех членов Думы. В Думе работает около сорока комиссий, все участники которых являются членами Думы. Каждая комиссия имеет свой профиль, например, вопросы образования, проблемы, связанные с жильем, и т. п. Данные по каждой из комиссий включают: председатель и состав, прежние (за 10 предыдущих лет) председатели и члены этой комиссии, даты включения и выхода из состава комиссии, избрания ее председателей. Члены Думы могут заседать в нескольких комиссиях. В базу заносятся время и место проведения каждого заседания комиссии с указанием депутатов и служащих Думы, которые участвуют в его организации.

1) Показать список комиссий, для каждой — ее состав с указанием председателя.

2) Предоставить возможность добавления нового члена комиссии.

3) Для введенного пользователем интервала дат и названия комиссии показать в хронологическом порядке всех ее председателей.

4) Показать список членов Думы, для каждого из них — список комиссий, в которых он участвовал и/или был председателем.

5) Предоставить возможность добавления новой комиссии, с указанием председателя.

6) Для указанного интервала дат и комиссии выдать список членов с указанием количества пропущенных заседаний.

7) Вывести список заседаний в указанный интервал дат в хронологическом порядке, для каждого заседания — список присутствующих.

8) Предоставить возможность добавления нового заседания, с указанием присутствующих.

9) По каждой комиссии показать количество проведенных заседаний в указанный период времени.

Задача 6. *База данных рыболовной фирмы.*

Фирме принадлежит небольшая флотилия рыболовных катеров. Каждый катер имеет паспорт, куда занесены его название, тип, водоизмещение и дата постройки. Фирма регистрирует каждый выход на лов, записывая название катера, имена и адреса членов команды с указанием их должностей (капитан, боцман и т. п.), даты выхода и возвращения, а также улов (массу пойманной рыбы) отдельно по сортам (например, трески). За время одного рейса катер может посетить несколько рыболовных мест (банок). Фиксируется дата прихода на каждую банку и дата отплытия, качество выловленной рыбы (отличное, хорошее, плохое). На борту улов не взвешивается. Написать запросы, осуществляющие следующие операции.

1) По указанному типу и интервалу дат вывести все катера, осуществлявшие выход в море, указав для каждого в хронологическом порядке записи о выходе в море и значениях улова.

2) Предоставить возможность добавления выхода катера в море с указанием команды.

3) Для указанного интервала дат вывести для каждого сорта рыбы список катеров с наибольшим уловом.

4) Для указанного интервала дат вывести список банок, с указанием среднего улова за этот период. Для каждой банки вывести список катеров, осуществлявших лов.

5) Предоставить возможность добавления новой банки с указанием данных о ней.

6) Для заданной банки вывести список катеров, которые получили улов выше среднего.

7) Вывести список сортов рыбы и для каждого сорта — список рейсов с указанием даты выхода и возвращения, величины улова. При этом список показанных рейсов должен быть ограничен интервалом дат.

8) Для выбранного пользователем рейса и банки добавить данные о сорте и количестве пойманной рыбы.

9) Предоставить возможность пользователю изменять характеристики выбранного катера.

10) Для указанного интервала дат вывести в хронологическом порядке список рейсов за этот период времени, с указанием для каждого рейса пойманного количества каждого сорта рыбы.

11) Предоставить возможность добавления нового катера.

12) Для указанных сорта рыбы и банки вывести список рейсов с указанием количества пойманной рыбы. Список должен быть отсортирован в порядке уменьшения количества пойманной рыбы.

Задача 7. *База данных фирмы, проводящей аукционы.*

Фирма занимается продажей с аукциона антикварных изделий и произведений искусства. Владельцы вещей, выставляемых на проводимых фирмой аукционах, юридически являются продавцами. Лица, приобретающие эти вещи, именуются покупателями. Получив от продавцов партию предметов, фирма решает, на котором из аукционов выгоднее представить конкретный предмет. Перед проведением очередного аукциона каждой из выставляемых на нем вещей присваивается отдельный номер лота. Две вещи, продаваемые на различных аукционах, могут иметь одинаковые номера лотов.

В книгах фирмы делается запись о каждом аукционе. Там отмечаются дата, место и время его проведения, а также специфика (например, выставляются картины, написанные маслом и не ранее 1900 г.). Заносятся также сведения о каждом продаваемом предмете: аукцион, на который он заявлен, номер лота, продавец, отправная цена и краткое словесное описание. Продавцу разрешается выставлять любое количество вещей, а покупатель имеет право приобретать любое количество вещей. Одно и то же лицо или фирма может выступать и как продавец, и как покупатель. После аукциона служащие фирмы, проводящей аукционы, записывают фактическую цену, уплаченную за проданный предмет, и фиксируют данные покупателя.

Написать запросы, осуществляющие следующие операции.

1) Для указанного интервала дат вывести список аукционов в хронологическом порядке с указанием наименования, даты и места проведения. Для каждого из них показать список выставленных вещей.

2) Добавить для продажи на указанный пользователем аукцион предмет искусства с указанием начальной цены.

3) Вывести список аукционов с указанием отсортированных по величине суммарных доходов от продажи.

4) Для указанного интервала дат вывести список проданных на аукционах предметов. Для каждого из предметов дать список аукционов, где он выставлялся.

5) Предоставить возможность добавления факта продажи заданного предмета на указанном аукционе.

6) Для указанного интервала дат вывести список продавцов в порядке убывания общей суммы, полученной ими от продажи предметов в этот промежуток времени.

7) Вывести список покупателей и для каждого из них — список аукционов, где были сделаны приобретения в указанный интервал дат.

8) Предоставить возможность добавления записи о проводимом аукционе (место, время).

9) Для указанного места вывести список аукционов, отсортированных по количеству выставленных вещей.

10) Для указанного интервала дат вывести список продавцов, которые принимали участие в аукционах, с указанием для каждого из них списка выставленных предметов.

11) Предоставить возможность добавления и изменения информации о продавцах и покупателях.

12) Вывести список покупателей с указанием количества приобретенных предметов в указанный период времени.

Задача 8. *База данных музыкального магазина.*

Таблицы базы данных содержат информацию о музыкантах, музыкальных произведениях и обстоятельствах их исполнения. Несколько музыкантов, образующих единый коллектив, называются ансамблем. Это может быть классический оркестр, джазовая группа, квартет, квинтет и т. п. К музыкантам причисляют исполнителей (играющих на одном или нескольких инструментах), композиторов, дирижеров и руководителей ансамблей.

Кроме того, в базе данных хранится информация о компакт-дисках, которыми торгует магазин. Каждый компакт-диск, а точнее, его наклейка, идентифицируется отдельным номером, так что всем его копиям, созданным в разное время, присвоены одинаковые номера. На компакт-диске может быть записано несколько вариантов исполнения одного и того же произведения — для каждого из них в базе заведена отдельная запись. Когда выходит новый компакт-диск, регистрируется название выпустившей его компании (например, EMI), а также адрес оптовой фирмы, у которой магазин может приобрести этот компакт-диск. Не исключено, что компания-производитель занимается и оптовой продажей компакт-дисков. Магазин фиксирует текущие оптовые и розничные цены на каждый компакт-диск, дату его выпуска, количество экземпляров, проданных за прошлый год и в нынешнем году, а также число еще не проданных компакт-дисков.

Задача 9. *База данных кегельной лиги.*

Ставится задача спроектировать базу данных для секретаря кегельной лиги небольшого городка, расположенного на Среднем Западе США. В ней секретарь будет хранить всю информацию, относящуюся к кегельной лиге, а средствами СУБД — формировать еженедельные отчеты о состоянии лиги. Специальный отчет предполагается формировать в конце сезона.

Секретарю понадобятся фамилии и имена членов лиги, их телефонные номера и адреса. Так как в лигу могут входить только жители городка, нет необходимости хранения для каждого игрока названия города и почтового индекса. Интерес представляют число очков, набранных каждым игроком в еженедельной серии из трех встреч, в которых он принял участие, и его текущая результативность (среднее

число набираемых очков в одной встрече). Секретарю необходимо знать для каждого игрока название команды, за которую он выступает, и фамилию (и имя) капитана каждой команды. Помимо названия, секретарь планирует назначить каждой команде уникальный номер.

Исходные значения результативности каждого игрока необходимы как в конце сезона, при определении игрока, достигшего наибольшего прогресса в лиге, так и при вычислении гандикапа для каждого игрока на первую неделю нового сезона. Лучшая игра каждого игрока и лучшие серии потребуются при распределении призов в конце сезона.

На каждую неделю каждой команде требуется назначать площадку, на которой она будет выступать. Эту информацию хранить в БД не нужно (соперники выступают на смежных площадках).

Наконец, в БД должна содержаться вся информация, необходимая для расчета положения команд. Команде засчитывается одна победа за каждую игру, в которой ей удалось набрать больше очков (выбить больше кеглей), чем команде соперников. Точно так же команде засчитывается одно поражение за каждую встречу, в которой эта команда выбила меньшее количество кеглей, чем команда соперников. Команде также засчитывается одна победа (поражение) в случае, если по сравнению с командой соперников ею набрано больше (меньше) очков за три встречи, состоявшиеся на неделе. Таким образом, на каждой неделе разыгрывается 4 командных очка (побед или поражений). В случае ничейного результата каждая команда получает 1/2 победы и 1/2 поражения. В случае неявки более чем двух членов команде автоматически засчитывается 4 поражения, а команде соперников — 4 победы. В общий результат команде, которой засчитана неявка, очки не прибавляются, даже если явившиеся игроки в этой встрече выступили, однако в индивидуальные показатели — число набранных очков и проведенных встреч — будут внесены соответствующие изменения. Написать запросы, осуществляющие следующие операции.

1) Для указанного интервала дат показать список выступающих команд. Для каждой из них вывести состав и капитана команды.

2) Предоставить возможность добавления новой команды.

3) Вывести список игровых площадок с указанием количества проведенных игр на каждой из них.

4) Для указанного интервала дат вывести список игровых площадок, с указанием списка игравших на них команд.

5) Предоставить возможность заполнения результатов игры двух команд на указанной площадке.

6) Вывести список площадок с указанием суммарной результативности игроков на каждой из них.

Задача 10. *База данных библиотеки.*

Разработать информационную систему обслуживания библиотеки, которая содержит следующую информацию: название книги, ФИО авторов, наименование издательства, год издания, количество страниц,

количество иллюстраций, цена, название филиала библиотеки или книгохранилища, в которых находится книга, количество имеющихся в библиотеке экземпляров конкретной книги, количество студентов, которым выдавалась конкретная книга, названия факультетов, в учебном процессе которых используется указанная книга.

Задача 11. *База данных по учету успеваемости студентов.*
База данных должна содержать данные:

- о контингенте студентов — фамилия, имя, отчество, год поступления, форма обучения (дневная/вечерняя/заочная), номер или название группы;
- об учебном плане — название специальности, дисциплина, семестр, количество отводимых на дисциплину часов, форма отчетности (экзамен/зачет);
- о журнале успеваемости студентов — год/семестр, студент, дисциплина, оценка.

Задача 12. *База данных для учета аудиторного фонда университета.*

База данных должна содержать следующую информацию об аудиторном фонде университета: наименование корпуса, в котором расположено помещение, номер комнаты, расположение комнаты в корпусе, ширина и длина комнаты в метрах, назначение и вид помещения, подразделение университета, за которым закреплено помещение. В базе данных также должна быть информация о высоте потолков в помещениях (в зависимости от места расположения помещений в корпусе). Следует также учитывать, что структура подразделений университета имеет иерархический вид, когда одни подразделения входят в состав других (факультет, кафедра, лаборатория, ...).

Помимо SQL-запросов для создания таблиц базы данных, составьте запрос на создание представления (**`VIEW`**), в котором помимо приведенной выше информации присутствовали бы данные о площадях и объемах каждого помещения.

Задача 13. *База данных регистрации происшествий.*
Необходимо создать базу данных регистрации происшествий. База должна содержать:

- данные для регистрации сообщений о происшествиях (регистрационный номер сообщения, дата регистрации, краткая фабула (тип происшествия));
- информацию о принятом по происшествию решении (отказано в возбуждении дела, удовлетворено ходатайство о возбуждении уголовного дела с указанием регистрационного номера заведенного дела, отправлено по территориальной принадлежности);
- информацию о лицах, причастных к происшествию (регистрационный номер лица, фамилия, имя, отчество, адрес, количество

судимостей, отношение конкретного лица к данному происшествию: виновник, потерпевший, подозреваемый, свидетель, ...).

Задача 14. *База данных для подготовки работы конференции.*

База данных должна содержать справочник персоналий участников конференции (фамилия, имя, отчество, ученая степень, ученое звание, научное направление, место работы, кафедра (отдел), должность, страна, город, почтовый индекс, адрес, рабочий телефон, домашний телефон, электронный адрес), и информацию, связанную с участием в конференции (докладчик или участник, дата рассылки первого приглашения, дата поступления заявки, тема доклада, отметка о поступлении тезисов, дата рассылки второго приглашения, дата поступления оргвзноса, размер поступившего оргвзноса, дата приезда, дата отъезда, потребность в гостинице).

Задача 15. *База данных для обслуживания склада.*

База данных должна обеспечить автоматизацию складского учета. В ней должны содержаться следующие данные:

- информация о "единицах хранения" — номер ордера, дата, код поставщика, балансный счет, код сопроводительного документа по справочнику документов, номер сопроводительного документа, код материала по справочнику материалов, счет материала, код единицы измерения, количество пришедшего материала, цена единицы измерения;
- информация о хранящихся на складе материалах (справочник материалов) — код класса материала, код группы материала, наименование материала;
- информация о единицах измерения конкретных видов материалов — код материала, единица измерения (метры, килограммы, литры и т. п.).
- информация о поставщиках материалов — код поставщика, его наименование, ИНН, юридический адрес (индекс, город, улица, дом), адрес банка (индекс, город, улица, дом), номер банковского счета.

Задача 16. *База данных фирмы.*

Фирма отказалась от приобретения некоторых товаров у своих поставщиков, решив самостоятельно наладить их производство. С этой целью она организовала сеть специализированных цехов, каждый из которых принимает определенное участие в технологическом процессе.

Каждому виду выпускаемой продукции присваивается, как обычно, свой шифр товара, под которым он значится в файле товарных запасов. Этот же номер служит и шифром продукта. В записи с этим шифром указывается, когда была изготовлена последняя партия этого продукта, какова ее стоимость, сколько операций потребовалось.

Операцией считается законченная часть процесса производства, которая целиком выполняется силами одного цеха в соответствии с техническими требованиями, перечисленными на отдельном чертеже. Для каждого продукта и для каждой операции в базе данных фирмы заведена запись, содержащая описание операции, ее среднюю продолжительность и номер требуемого чертежа, по которому его можно отыскать. Кроме того, указывается номер цеха, обычно производящего данную операцию.

В запись, связанную с конкретной операцией, заносятся потребные количества расходных материалов, а также присвоенные им шифры товара. Расходными называют такие материалы, как, например, электрический кабель, который нельзя использовать повторно. При выдаче расходного материала со склада в процессе подготовки к выполнению операции, регистрируется фактически выданное количество, соответствующий шифр товара, номер служащего, ответственного за выдачу, дата и время выдачи, номер операции и номер наряда на проведение работ (о котором несколько ниже). Реально затраченное количество материала может не совпадать с расчетным (например, из-за брака).

Каждый из цехов располагает требуемым инструментарием и оборудованием. При выполнении некоторых операций их иногда недостаточно, и цех вынужден обращаться в центральную инструментальную за недостающими инструментами. Каждый тип инструмента снабжен отдельным номером, и на него заведена запись со словесным описанием. Кроме того, отмечается, какое количество инструментов этого типа выделено цехам и какое осталось в инструментальной. Экземпляры инструмента конкретного типа, например, гаечные ключи одного размера, различаются по своим индивидуальным номерам. На фирме для каждого типа инструмента имеется запись, содержащая перечень всех индивидуальных номеров. Кроме того, указаны даты их поступления на склад.

По каждой операции в фирме отмечают типы и соответствующие количества инструментов, которые должны использоваться при ее выполнении. Когда инструменты действительно берутся со склада, фиксируется индивидуальный номер каждого экземпляра, указываются номер заказавшего их цеха и номер наряда на проведение работ. В этом случае реально полученное количество также не всегда совпадает с заказанным.

Наряд на проведение работ по форме напоминает заказ на приобретение товаров, но, в отличие от последнего, направляется не поставщику, а в один из цехов. Оформляется наряд после того, как руководство фирмы сочтет необходимым выпустить партию некоторого продукта. В наряд заносятся шифр продукта, дата оформления наряда, срок, к которому должен быть выполнен заказ, а также требуемое количество продукта.

Разработайте структуру таблиц базы данных, подберите имена таблиц и полей, в которых могла бы разместиться вся эта информация.

Напишите SQL-запросы, осуществляющие следующие операции.

1) Для выбранного цеха выдать список выполняемых им операций. Для каждой операции показать список расходных материалов с указанием количества.

2) Показать список инструментов и предоставить возможность добавления нового.

3) Выдать список используемых инструментов, отсортированных по количеству их использования в различных нарядах.

4) Для указанного интервала дат вывести в хронологическом порядке список нарядов, для каждого из которых показать список используемых инструментов.

5) Показать список операций и предоставить возможность добавления новой операции.

6) Выдать список расходуемых материалов, отсортированных по количеству их использования в различных нарядах.

7) Выдать список товаров, с указанием используемых при их изготовлении инструментов.

8) Показать список нарядов в хронологическом порядке и предоставить возможность добавления нового.

9) Выдать отчет о производстве товаров различными цехами, указав наименование цеха, название товара и его количество.

Предметный указатель