Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Калмыцкий государственный университет имени Б.Б.Городовикова»

В. А. Сидоренко

Язык SQL

Учебное пособие

Элиста 2021

Аннотация

Учебное пособие «Язык SQL» предназначено для студентов третьего курса направления «Фундаментальная информатика и информационные технологии». Пособие поможет в изучении языка SQL, применяемого при разработке баз данных. При этом подразумевается, что основные знания по технологиям баз данных студент уже получил.

Изучение дисциплины на третьем курсе занимает один семестр, объём теоретической части – 36 часов.

Введение

В настоящее время информационные системы, применяющие базы данных, представляют собой одну из важнейших областей современных компьютерных технологий. С этой сферой связана большая часть современного рынка программных продуктов. Среди общих тенденций в развитии таких систем выделяются процессы интеграции и стандартизации, затрагивающие структуры данных и способы их обработки и интерпретации, системное и прикладное программное обеспечение, средства взаимодействия компонентов баз данных и многое другое. Современные системы управления базами данных (СУБД) основаны на реляционной модели представления данных – в большой степени благодаря простоте и четкости ее концептуальных понятий и строгого математического обоснования.

Неотъемлемая и важнейшая часть любой системы, применяющей базы данных, – языковые средства, обеспечивающие возможность доступа и действий над данными, определения их структур, способов использования и интерпретации. Язык SQL (Structured Query Language) появился в 1970-е годы 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.

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

Стандарт SQL определяется **ANSI** (*Американским Национальным Институтом Стандартов*) и в данное время также принимается **ISO** (*международной организацией по стандартизации*). В 1992 году ANSI принял стандарт, который был назван SQL-92 (или SQL2).

Этот стандарт поддерживается всеми ведущими мировыми фирмами, действующими в сфере технологий баз данных. Использование выразительного и эффективного стандартного языка позволило обеспечить высокую степень независимости разрабатываемых прикладных программных систем от конкретного типа используемой СУБД, существенно поднять уровень и унификацию инструментальных средств разработки приложений, работающих с реляционными базами данных.

Говоря о стандарте языка SQL, следует заметить, что большинство его коммерческих реализаций имеют некоторые, большие или меньшие, отличия от стандарта. Это, конечно, ухудшает совместимость систем, использующих

различные «диалекты» SQL. Но, с другой стороны, полезные расширения реализаций языка обеспечивают его развитие и со временем включаются в новые редакции стандарта. Учитывая место, занимаемое SQL в современных информационных технологиях, его знание необходимо любому специалисту, работающему в этой области.

Глава 1

ВВЕДЕНИЕ В РЕЛЯЦИОННУЮ БАЗУ ДАННЫХ

Теория проектирования баз данных

Прежде, чем изучать язык SQL, вспомним некоторые основные понятия баз данных.

Программирование баз данных – очень большой и серьезный раздел самого что ни на есть практического программирования. Многие программисты большую часть своего времени тратят именно на проектирование баз данных и разработку приложений, работающих с ними. Это неудивительно – в настоящее время каждая государственная организация, каждая фирма или крупная корпорация имеют рабочие места с компьютерами. Имеется масса данных, которые нужно не только сохранить, но и обработать, получить комплексные отчеты. Без баз данных сегодня не обойтись.

Недостаточно просто написать программу, взаимодействующую с БД. Нужно уметь правильно спроектировать эту базу данных. Проектирование баз данных, в общем, является первым шагом разработки приложения. Только когда база данных спроектирована, программист приступает непосредственно к проекту приложения. На этой лекции мы коротко определимся с терминологией БД (вспомним, если кто забыл), затем изучим вопросы проектирования баз данных.

На предыдущем курсе мы упоминали, что существуют такие типы баз данных: локальные, файл-серверные, клиент-серверные и распределенные БД. Нам с вами доводилось работать с **локальными** БД, однако многое осталось «за кадром» – в рамках одного курса просто невозможно дать материал по разнообразным темам, для каждой из которых написано немало книг. Здесь мы продолжим знакомство с БД. Мы познакомимся с различными механизмами доступа к базам данных. Подробно изучим архитектуру **клиент-сервер**, которая является наиболее востребованной сегодня архитектурой программирования БД. Также рассмотрим механизмы создания **распределенных** (или многоуровневых) баз данных. Файл-серверные БД имеют очень ограниченные возможности, и в настоящий момент практически не используются, поэтому мы не будем касаться этой темы. Вместо этого гораздо удобней использовать распределенную архитектуру совместно с применением локальных технологий. Обо всем этом и о многом другом мы поговорим на этом курсе.

Ранее мы пользовались **BDE** – встроенным механизмом доступа к базам данных. Больше к этим темам мы возвращаться не будем. Тем не менее, в рамках изучения новых возможностей при работе с базами данных, мы кратко коснемся и BDE, наряду с другими технологиями доступа.

Терминология

Базой данных (БД) называется электронное хранилище информации, доступ к которому имеет один или несколько компьютеров.

В былые времена под базой данных понимали файл, где данные хранились в табличном виде. Сейчас под базой данных обычно подразумевают папку, в которой хранится один или несколько файлов с таблицами. Эти таблицы, вместе или по отдельности, взаимодействуют с пользовательским приложением. Существуют базы данных, в которых таблицы, индексы и другие служебные данные хранятся в одном файле. К таким БД можно отнести, например, MS SQL Server, MS Access, InterBase. В этом случае базой данных будет созданный файл.

Таблицы, имеющие связи между собой, называют **реляционными**, и базы данных, в которых имеются взаимосвязанные таблицы, также называются реляционными. Реляционные базы данных в настоящее время наиболее распространены.

Часто пользовательские приложения не работают с базами данных напрямую. Имеются специальные программы, называемые **Системы Управления Базами Данных (СУБД)**, которые служат посредниками между базой данных и пользовательским приложением. Такой подход называют архитектурой клиент-сервер, а такие СУБД часто называют серверами баз данных. Иногда еще добавляют букву **Р** (**РСУБД** – Реляционная СУБД).

Однако не все СУБД предназначены для архитектуры клиент-сервер. Например, программа Access из пакета MS Office – это СУБД, предназначенная для локального или файл-серверного использования.

Основой любой БД является таблица. **Таблица** – это файл определенного формата с данными, представленными в табличном виде, например:

№	Фамилия	Имя	Отчество
1	Иванов	Иван	Иванович
2	Петров	Петр	Петрович
3	Сидоров	Сидор	Сидорович

Рис. 1.1. Представление данных в табличном виде.

Такая таблица состоит из полей и записей.

Поле – столбец таблицы, имеющий название, тип данных и размер. Поле предназначено для описания отдельного атрибута записи. Например, поле «№» имеет целочисленный тип данных, а поле «Фамилия» – строковый.

Запись – строка таблицы, описывающая какой-то объект, или иначе, набор атрибутов какого-то объекта. Например, строка под номером 1 описывает человека – Иванова Ивана Ивановича.

Первичный ключ – это поле или набор полей, однозначно идентифицирующих запись. В ключевом поле не может быть двух записей с одинаковым значением. Например, поле «Фамилия» нельзя делать ключевым, ведь в таблице могут оказаться однофамильцы. Поле «№» больше подходит для того, чтобы сделать его ключевым. Первичные ключи помогают упорядочить записи и облегчают установку связей между таблицами. Каждая таблица может содержать только один первичный ключ.

Индекс – это поле или набор полей, которые часто используются для сортировки или поиска данных. Индексные поля еще называют вторичными ключами. В отличие от первичных ключей, поля для индексов могут содержать как уникальные, так и повторяющиеся значения. Например, поле «Фамилия» можно сделать индексным – ведь поиск и сортировка записей часто может производиться по этому полю. Индексы могут быть уникальными, то есть, не допускающими совпадений в записях, как первичные ключи, и не уникальными, допускающими такие совпадения. Индексы могут быть как в восходящем порядке (А, Б, …, Я), так и в нисходящем (Я, Ю, …, А). Таблица может иметь множество индексов. Можно все поля сделать индексными, причем даже на каждое поле по два индекса – в восходящем и нисходящем порядке. Однако при этом следует иметь в виду, что база данных в этом случае будет непомерно раздута, и работа с ней значительно замедлится. Другими словами, нужно соблюдать меру, и делать индексными только те поля, по которым действительно часто придется сортировать или фильтровать данные.

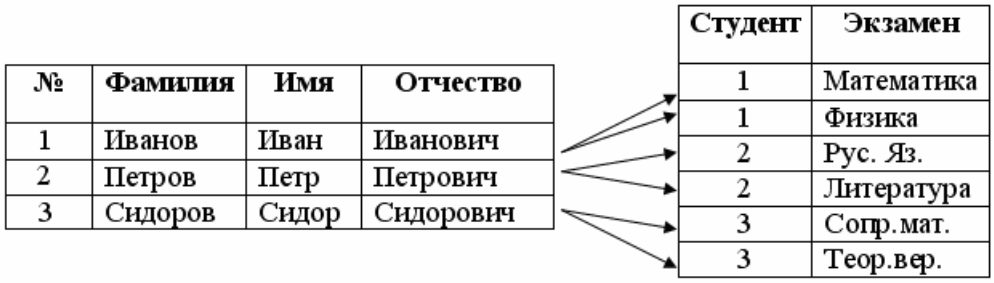
Связи (отношения)

Реляционные связи (отношения) между таблицами предназначены для разбивки таблиц на самодостаточные части. Рассмотрим пример. Допустим, люди из предыдущей таблицы – студенты. Таблица предназначена для того, чтобы указать, какие экзамены были сданы конкретным студентом. Следовательно, в таблицу требуется добавить поле «Экзамен»:

№	Фамилия	Имя	Отчество	Экзамен
1	Иванов	Иван	Иванович	Математика
2	Иванов	Иван	Иванович	Физика
3	Петров	Петр	Петрович	Рус. Яз.
4	Петров	Петр	Петрович	Литература
5	Сидоров	Сидор	Сидорович	Сопр.мат.
6	Сидоров	Сидор	Сидорович	Теор.вер.

Рис. 1.2. Исправленная таблица

Сразу бросается в глаза недостаток такого проектирования: данные из полей «Фамилия», «Имя» и «Отчество» многократно повторяются. Пользователю придется вводить большое количество дублирующих данных, а таблица получается «распухшей», переполненной этими данными. Исправить положение несложно, нужно лишь разбить эту таблицу на две разных таблицы, имеющие релятивную связь:



№	Фамилия	Имя	Отчество
1	Иванов	Иван	Иванович
2	Петров	Петр	Петрович
3	Сидоров	Сидор	Сидорович

Студент	Экзамен
1	Математика
1	Физика
2	Рус. Яз.
2	Литература
3	Сопр.мат.
3	Теор.вер.

Рис. 1.3. Реляционная связь между таблицами

Как вы можете заметить, избыточность данных исчезла – в одной таблице представлены только данные по студентам, в другой – данные по экзаменам. Связь между таблицами организована по ключевому полю «№» таблицы со студентами. В таблице с экзаменами, вместо полных данных о студенте вписывается только его номер. Студент может сдать сколько угодно много экзаменов, пользователь же просто выберет его из списка, и в таблицу попадет его номер. Такую таблицу легче заполнять, и размер ее будет тоже меньше.

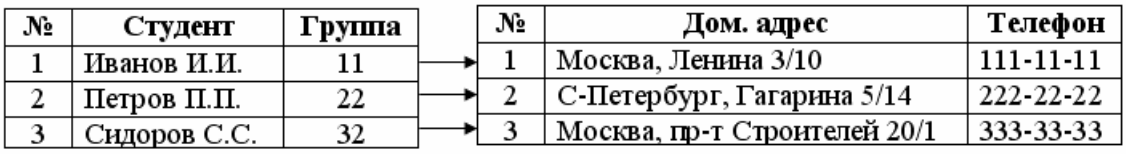
При создании связей, как правило, одна таблица называется **главной (master)**, другая – **подчиненной (details)**. В нашем случае главной является таблица со студентами. Таблица со списком сданных экзаменов – подчиненная.

Связь, представленная в рисунке 1.3 называется отношением **один-ко-многим**. То есть, одна запись из одной таблицы может иметь связь с множеством записей из другой таблицы. Однако имеется возможность и того, что запись из первой таблицы не будет иметь никаких связей с другой таблицей – студент может еще не сдать ни одного экзамена. Отношение *один-ко-многим* встречается наиболее часто.

Хотя в данной ситуации, если развивать тему, то можно легко прийти к выводу, что связи между этими таблицами удобнее считать *многие-ко-многим* – когда экзамены повторяются, но для разных студентов: та же «Математика» может стоять не только у Иванова, но и у Петрова, Сидорова.

Связь один-ко-многим в устойчивом состоянии можно представить в примере с писателями – где в одной таблице список писателей, а во второй – список их произведений.

Отношение **один-к-одному** подразумевает, что одной записи в главной таблице соответствует одна запись в подчиненной таблице. Взгляните на рисунок:



№	Студент	Группа
1	Иванов И.И.	11
2	Петров П.П.	22
3	Сидоров С.С.	32

№	Дом. адрес	Телефон
1	Москва, Ленина 3/10	111-11-11
2	С-Петербург, Гагарина 5/14	222-22-22
3	Москва, пр-т Строителей 20/1	333-33-33

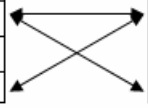
Рис. 1.4. Отношение один-к-одному

Данные о студентах, такие как фамилия, группа, могут часто использоваться для самых разных отчетов. Однако домашний адрес и телефон студентов нужны далеко не всегда, поэтому они вынесены в другую таблицу. Если бы мы объединили эти таблицы в одну, то получили бы таблицу с

переизбытком данных. Связь *один-к-одному* используют для того, чтобы отделить главную информацию от второстепенных данных.

Отношение **многие-ко-многим** встречается реже. Такое отношение подразумевает, что одна запись из главной таблицы может иметь связь со многими записями из подчиненной таблицы. А одна запись из подчиненной таблицы может иметь связь со многими записями главной таблицы. Рассмотрим такой типичный пример:

№	Покупатель
1	Иванов И.И.
2	Петров П.П.
3	Сидоров С.С.



Код	Товар	Ед.изм	Цена р.
А-01	Макароны	кг.	100
А-02	Сахар	кг.	150
С-34	Хлеб	шт.	50

Рис. 1.5. Отношение многие-ко-многим

Как видно из рисунка, один покупатель может купить несколько товаров, в то же время как один товар может быть куплен несколькими покупателями. Считается, что базу данных можно спроектировать так, чтобы любая связь *многие-ко-многим* была бы заменена одной или более связями *один-ко-многим*.

Ссылочная целостность

Ссылочной целостностью называют особый механизм, осуществляемый средствами СУБД или программистом, ответственный за поддержание непротиворечивых данных в связанных релятивными отношениями таблицах. Ссылочная целостность подразумевает, что в таблицах, имеющих релятивные связи, нет ссылок на несуществующие записи. Взгляните на рис. 1.3. Если мы удалим из списка студента Иванова И.И., и при этом не изменим таблицу со сданными экзаменами, ссылочная целостность будет нарушена, в таблице с экзаменами появится «мусор» – данные, на которые не ссылается ни одна запись из таблицы студентов. Ссылочная целостность будет нарушена. Таким образом, если мы удаляем из списка студента Иванова И.И., следует позаботиться о том, чтобы из таблицы со сданными экзаменами также были удалены все записи, на которые ранее ссылалась удаленная запись главной таблицы. Существует несколько видов изменений данных, которые могут привести к нарушению ссылочной целостности:

1. Удаляется запись в родительской таблице, но не удаляются соответствующие связанные записи в дочерней таблице.
2. Изменяется запись в родительской таблице, но не изменяются соответствующие ключи в дочерней таблице.
3. Изменяется ключ в дочерней таблице, но не изменяется значение связанного поля родительской таблицы.

Многие СУБД блокируют действия пользователя, которые могут привести к нарушению связей. Нарушение хотя бы одной такой связи делает информацию в БД недостоверной. Если мы, например, удалили Иванова И.И., то теперь номер 1 принадлежит Петрову П.П.. Имеющиеся связи указывают, что он сдал экзамены по математике и физике, но не сдавал экзаменов по

русскому языку и литературе. Достоверность данных нарушена. Конечно, в таких случаях в качестве ключа обычно используют счетчик – поле автоинкрементного типа. Если удалить запись со значением 1, то другие записи не изменят своего значения, значение 1 просто невозможно будет присвоить какой-то другой записи, оно будет отсутствовать в таблице. Путаницы в связях не случится, однако все равно подчиненная таблица будет иметь «потерянные» записи, не связанные ни с какой записью главной таблицы. Механизм ссылочной целостности должен запрещать удаление записи в главной таблице до того, как будут удалены все связанные с ней записи в дочерней таблице.

Нормализация базы данных

Каждый программист обычно по-своему проектирует базу данных для программы, над которой работает. У одних это получается лучше, у других – хуже. Качество спроектированной БД в немалой степени зависит от опыта и интуиции программиста, однако существуют некоторые правила, помогающие улучшить проектируемую БД. Такие правила носят *рекомендательный* характер, и называются **нормализацией** базы данных.

Процесс нормализации данных заключается в устранении избыточности данных в таблицах.

Существует несколько *нормальных форм*, но для практических целей интерес представляют только первые три *нормальные формы*.

Первая нормальная форма (1НФ) требует, чтобы каждое поле таблицы БД было неделимым (атомарным) и не содержало повторяющихся групп.

Неделимость означает, что в таблице не должно быть полей, которые можно разбить на более мелкие поля. Например, если в одном поле мы объединим фамилию студента и группу, в которой он учится, требование неделимости соблюдаться не будет. Первая нормальная форма требует, чтобы мы разбили эти данные по двум полям.

Под понятием *повторяющиеся группы* подразумевают поля, содержащие одинаковые по смыслу значения. Взгляните на рисунок:

№	Студент 1	Студент 2	Студент 3
1	Иванов И.И.	Петров П.П.	Сидоров С.С.

Рис. 1.6. Повторяющиеся группы

Верно, такую таблицу можно сделать, однако она нарушает правило *первой нормальной формы*. Поля «Студент 1», «Студент 2» и «Студент 3» содержат одинаковые по смыслу объекты, их требуется поместить в одно поле «Студент», как в рисунке 1.4. Ведь в группе не бывает по три студента, правда? Их гораздо больше. И представляете, как будет выглядеть таблица, содержащая данные на тридцать студентов? Это тридцать одинаковых полей! В приведенном выше рисунке поля описывают студентов в формате «Фамилия И.О.». Однако если оператор будет вводить эти описания в формате «Фамилия Имя Отчество», то нарушается также правило неделимости. В этом случае

каждое такое поле следует разбить на три отдельных поля, так как поиск может вестись не только по фамилии, но и по имени или по отчеству.

Вторая нормальная форма (2НФ) требует, чтобы таблица удовлетворяла всем требованиям первой нормальной формы, и чтобы любое не ключевое поле однозначно идентифицировалось полным набором ключевых полей. Рассмотрим пример: некоторые студенты посещают спортивные платные секции, и вуз взял на себя оплату этих секций. Взгляните на рисунок:

№ студента	Секция	Плата
100	Плавание	100
110	Скейтборд	150
112	Теннис	175
254	Плавание	100
260	Теннис	175

Рис. 1.7. Нарушение второй нормальной формы

В чем здесь нарушение? Ключом этой таблицы служат поля «№ студента» – «Секция». Однако данная таблица также содержит отношение «Секция» – «Плата». Если мы удалим запись студента № 110, то потеряем данные о стоимости секции по скейтборду. А после этого мы не сможем ввести информацию об этой секции, пока в нее не запишется хотя бы один студент. Говорят, что такое отношение подвержено как аномалии удаления, так и аномалии вставки. В соответствие с требованиями *второй нормальной формы*, каждое не ключевое поле должно однозначно зависеть от ключа. Поле «Плата» в приведенном примере содержит сведения о стоимости данной секции, и ни коим образом не зависит от ключа – номера студента. Таким образом, чтобы удовлетворить требованию *второй нормальной формы*, данную таблицу следует разбить на две таблицы, каждая из которых зависит от своего ключа:

№ студента	Секция
100	Плавание
110	Скейтборд
112	Теннис
254	Плавание
260	Теннис

Ключ: № студента

Секция	Плата
Плавание	100
Скейтборд	150
Теннис	175

Ключ: Секция

Рис. 1.8. Правильная вторая нормальная форма

Мы получили две таблицы, в каждой из которых не ключевые данные однозначно зависят от своего ключа.

Третья нормальная форма (3НФ) требует, чтобы в таблице не имелось транзитивных зависимостей между неключевыми полями, то есть, чтобы значение любого поля, не входящего в первичный ключ, не зависело от другого поля, также не входящего в первичный ключ. Допустим, в нашей студенческой базе данных есть таблица с расходами на спортивные секции:

Секция	Плата	Кол-во студентов	Общая стоимость
Плавание	100	2	200
Скейтборд	150	1	150
Теннис	175	2	350

Рис. 1.9. Нарушение третьей нормальной формы

Как нетрудно заметить, ключевым полем здесь является поле «Секция». Поля «Плата» и «Кол-во студентов» зависят от ключевого поля и не зависят друг от друга. Однако поле «Общая стоимость» зависит от полей «Плата» и «Кол-во студентов», которые не являются ключевыми, следовательно, нарушается правило *третьей нормальной формы*.

Поле «Общая стоимость» в данном примере можно спокойно убрать из таблицы, ведь если потребуется вывести такие данные, нетрудно будет перемножить значения полей «Плата» и «Кол-во студентов», и создать для вывода вычисляемое поле.

Таким образом, нормализация данных подразумевает, что вы вначале проектируете свою базу данных: планируете, какие таблицы у вас будут, какие в них будут поля, какого типа и размера. Затем вы приводите каждую таблицу к первой нормальной форме. После этого приводите полученные таблицы ко второй, затем к третьей нормальной форме, после чего можете утверждать, что ваша база данных нормализована.

Однако такой подход имеет и недостатки: если вам требуется разработать программный комплекс для крупного предприятия, база данных будет довольно большой. При нормализации данных, вы можете получить сотни взаимосвязанных между собой таблиц. С увеличением числа нормализованных таблиц уменьшается восприятие программистом базы данных в целом, то есть вы можете потерять общее представление вашей базы данных, запутаетесь в связях. Кроме того, поиск в чересчур нормализованных данных может быть замедлен. Отсюда вывод: при работе с данными большого объема ищите компромисс между требованиями нормализации и собственным общим восприятием базы данных.

Глава 2

SQL: ОБЗОР

Отличие SQL от процедурных языков программирования

SQL относится к классу непроцедурных языков программирования. В отличие от универсальных процедурных языков, которые также могут быть использованы для работы с базами данных, 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 записей (кортежей)* одной или нескольких таблиц (отношений). В

результате выполнения запроса также образуется *множество записей* результирующей таблицы. Другими словами, в SQL результатом любой операции над таблицами также является таблица. Запрос SQL задаёт не процедуру, то есть последовательность действий, необходимых для получения результата, а условия, которым должны удовлетворять записи результирующей таблицы, сформулированные в терминах входной (или входных) таблицы.

Интерактивный и встроенный SQL

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

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

Встроенный SQL состоит из команд SQL, встроенных внутрь программ, обычно написанных на каком-то другом языке (Паскаль, C, C++ и др.). Это делает программы, использующие такие языки, более мощными, гибкими и эффективными, обеспечивая их применение для работы с данными, хранящимися в реляционных базах. При этом, однако, требуются дополнительные средства интерфейса SQL с языком, в который он встраивается.

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.д.).

В число дополнительных функций DDL могут быть включены и средства ограничения целостности данных, определения порядка структур их хранения, описания элементов физического уровня хранения данных.

Язык обработки данных – DML (Data Manipulation Language) – предоставляет возможность выборки информации из базы данных и ее преобразования.

Язык Управления Данными – DCD – состоит из средств, которые определяют, разрешить ли пользователю выполнять определенные действия или нет.

Они являются составными частями DDL в ANSI. Это не различные языки, а разделы команд SQL сгруппированных по их функциям.

Итак, основным направлением в разработке автоматизированных информационных систем в настоящее время является ориентация на использование СУБД, базирующихся на SQL-серверах. В чём же состоят преимущества разработки информационных систем на их основе?

1. SQL-серверы прямо ориентированы на создание интегрированных, многопользовательских систем, имея в своем распоряжении развитые словари данных.

2. Средства разработки для этих СУБД оптимизированы в отношении коллективной разработки сложных систем в рамках единой стратегической линии.

3. Развитый механизм обработки транзакций позволяет обеспечить целостность данных при одновременной работе многих пользователей.

4. Использование единого языка доступа к данным (SQL) позволяет упростить переход от одной СУБД к другой.

5. Обеспечивается масштабируемость разрабатываемых систем.

6. Поддерживается возможность работы как в локальной, так и в глобальной сетях.

Синтаксис SQL

Начнем с определений.

Комментарий. Всякий *комментарий* – это необязательный текст, который вы печатаете на отдельной строке программы, чтобы объяснить эту программу. Комментарий должен начинаться с двух дефисов. Когда СУБД обнаруживает их, она игнорирует то, что стоит за ними, то есть сам комментарий. Комментарии занимают целую строку.

Команда SQL. Любая *команда SQL* – это допустимая комбинация лексем, которую предваряет какое-нибудь ключевое слово. Здесь под лексемой понимается основная неделимая частица языка SQL в том значении, что грамматически никакую лексему нельзя разделить на более мелкие составные элементы. Лексемами являются ключевые слова, идентификаторы, операторы, литералы и другие символы (все они будут рассмотрены позднее).

Предложения. Любая команда SQL включает не менее одного предложения. В самом общем случае всякое *предложение SQL* – это фрагмент команды SQL, который начинается с какого-нибудь ключевого слова, является обязательным или необязательным и должен быть записан в определенном порядке. В рассматриваемом примере мы имеем четыре предложения, а именно: SELECT, FROM, WHERE, ORDER.

Ключевые слова. Произвольное *ключевое слово*, иногда называемое *зарезервированным*, – это такое слово, которое в языке SQL имеет определенное значение и применение которого в SQL строго регламентировано. Следует иметь в виду, что использование любого ключевого слова вне контекста (например, в качестве идентификатора) будет считаться ошибкой. В табл. 2.1 перечислены ключевые слова SQL, а в табл. 2.2 – потенциальные слова SQL (которые пока не являются официально зарезервированными, но однажды могут ими стать).

Идентификаторы. Произвольный *идентификатор* – это такое слово, которое вы (или разработчик базы данных) применяете для того, чтобы именовать объекты произвольной базы данных, в том числе таблицы, столбцы, псевдоимена (псевдонимы) и представления. Идентификатор не может быть ключевым словом, и его длина не может превышать 128 знаков. Знаком в SQL может быть любой символ алфавита, включая символы латинского алфавита и латинские идеограммы (однако обратите внимание на советы, приведенные в конце настоящего раздела). В нашем примере именами, в частности, являются au_fname, au_lname, authors и state.

Завершающая точка с запятой. Запись каждой команды SQL должна заканчиваться точкой с запятой.

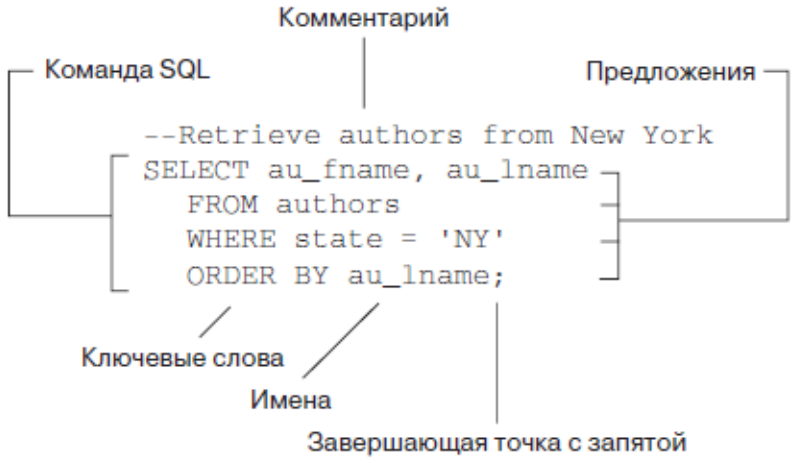


Рис. 2.1. Пример команды SQL с комментарием

Таблица 2.1. Ключевые слова SQL

ABSOLUTE	COMMIT	ELSE	INSERT	Null
ACTION	CONNECT	END	INT	ONLY
ADD	CONNECTION	END-EXEC	INTEGER	OPEN
ALL	CONSTRAINT	ESCAPE	INTERSECT	OPTION
ALLOCATE	CONSTRAINTS	EXCEPT	INTERVAL	OR
ALTER	CONTINUE	EXCEPTION	INTO	ORDER
AND	CONVERT	EXEC	IS	OUTER
ANY	CORRESPONDING	EXECUTE	ISOLATION	OUTPUT
ARE	COUNT	EXISTS	JOIN	OVERLAPS
AS	CREATE	EXTERNAL	KEY	PAD
ASC	CROSS	EXTRACT	LANGUAGE	PARTIAL
ASSERTION	CURRENT	FALSE	LAST	POSITION
AT	CURRENT_DATE	FETCH	LEADING	PRECISION
AUTHORIZATION	CURRENT_TIME	FIRST	LEFT	PREPARE
AVG	CURRENT_TIMESTAMP	FLOAT	LEVEL	PRESERVE
BEGIN	CURRENT_USER	FOR	LIKE	PRIMARY
BETWEEN	CURSOR	FOREIGN	LOCAL	PRIOR
BIT	DATE	FOUND	LOWER	PRIVILEGES
BIT_LENGTH	DAY	FROM	MATCH	PROCEDURE
BOTH	DEALLOCATE	FULL	MAX	PUBLIC
BY	DEC	GET	MIN	READ
CASCADE	DECIMAL	GLOBAL	MINUTE	REAL
CASCADED	DECLARE	GO	MODULE	REFERENCES
CASE	DEFAULT	GOTO	MONTH	RELATIVE
CAST	DEFERRABLE	GRANT	NAMES	RESTRICT
CATALOG	DEFERRED	GROUP	NATIONAL	REVOKE
CHAR	DELETE	HAVING	NATURAL	RIGHT
CHARACTER	DESC	HOUR	NCHAR	ROLLBACK
CHAR_LENGTH	DESCRIBE	IDENTITY	NEXT	ROWS
CHARACTER_LENGTH	DESCRIPTOR	IMMEDIATE	NO	SCHEMA
CHECK	DIAGNOSTICS	IN	NOT	SCROLL
CLOSE	DISCONNECT	INDICATOR	NULL	SECOND
COALESCE	DISTINCT	INITIALLY	NULLIF	SECTION
COLLATE	DOMAIN	INNER	NUMERIC	SELECT
COLLATION	DOUBLE	INPUT	OCTET_LENGTH	SESSION
COLUMN	DROP	INSENSITIVE	OF	SESSION_USER
SET	SUM	TRAILING	UPPER	WHENEVER
SIZE	SYSTEM_USER	TRANSACTION	USAGE	WHERE
SMALLINT	TABLE	TRANSLATE	USER	WITH
SOME	TEMPORARY	TRANSLATION	USING	WORK
SPACE	THEN	TRIM	VALUE	WRITE
SQL	TIME	TRUE	VALUES	YEAR
SQLCODE	TIMESTAMP	UNION	VARCHAR	ZONE
SQLERROR	TIMEZONE_HOUR	UNIQUE	VARYING	
SQLSTATE	TIMEZONE_MINUTE	UNKNOWN	VIEW	
SUBSTRING	TO	UPDATE	WHEN	

Таблица 3.2. Потенциальные ключевые слова SQL

AFTER	EQUALS	OLD	RETURN	TEST
ALIAS	GENERAL	OPERATION	RETURNS	THERE
ASYNC	IF	OPERATORS	ROLE	TRIGGER
BEFORE	IGNORE	OTHERS	ROUTINE	TYPE
BOOLEAN	LEAVE	PARAMETERS	ROW	UNDER
BREADTH	LESS	PENDANT	SAVEPOINT	VARIABLE
COMPLETION	LIMIT	PREORDER	SEARCH	VIRTUAL
CALL	LOOP	PRIVATE	SENSITIVE	VISIBLE
CYCLE	MODIFY	PROTECTED	SEQUENCE	WAIT
DATA	NEW	RECURSIVE	SIGNAL	WHILE
DEPTH	NONE	REF	SIMILAR	WITHOUT
DICTIONARY	OBJECT	REFERENCING	SQLEXCEPTION	
EACH	OFF	REPLACE	SQLWARNING	
ELSEIF	OID	RESIGNAL	STRUCTURE	

SQL – это язык со свободным форматом предложений. Любая его команда может:

- быть напечатана как в верхнем, так и в нижнем регистре (например, ключевые слова SELECT и select считаются идентичными);
- продолжаться на следующей строке сколь угодно долго при условии, что вы не будете разбивать на две части слова, лексемы и строки в кавычках (то есть строковые константы);
- быть напечатана на одной строке с любыми другими командами;
- начинаться на любой позиции горизонтальной разметки экрана/листа.

Однако вам следует придерживаться определенного стиля написания команд SQL, например использовать верхний регистр для ключевых слов и нижний регистр для идентификаторов. Кроме того, можно начинать каждое предложение с новой строки с соответствующим отступом. Вопрос стиля нельзя игнорировать, потому что его отсутствие приводит к следующим ошибкам:

- неправильная орфография при написании какого-нибудь идентификатора или команды;
- отсутствие завершающей точки с запятой;
- неправильный порядок расположения предложений в теле команды;
- отсутствие кавычек у строковых констант (литералов) и констант даты и времени;
- наличие кавычек у цифровых констант;
- неправильное совместное применение имен таблиц и имен столбцов.

Применять пробелы в именах базы данных не рекомендуется. Но если очень хочется, можно вставить пробелы в любой идентификатор, тогда его надо будет заключить в одинарные кавычки, вот так: ‘last name’. Тем не менее подчеркнем еще раз: лучше применять не пробелы, а знак подчеркивания (last_name) или идентификатор, набранный заглавными и прописными буквами (LastName).

Выражение – это любая разрешенная комбинация символов, которая служит для вычисления единого агрегированного значения данных. Составляя любое выражение, вы можете комбинировать математические или логические операторы, идентификаторы, константы, функции, имена столбцов и т.д. В табл. 2.3 перечислены часто используемые выражения вместе с примерами. Всякий раз, когда то или иное распространенное выражение возникнет по ходу изложения материала, мы будем разбирать его достаточно подробно.

Коммерческие СУБД налагают собственные ограничения на длину идентификатора и на алфавит. Поэтому имеет смысл проанализировать документацию по вашей СУБД, используя ключи поиска «идентификаторы» и «имена». Более того, различные СУБД имеют свои собственные дополнительные ключевые слова, которые, очевидно, не могут быть идентификаторами в «родной» СУБД (но вполне могут быть идентификаторами по канонам SQL и, возможно, в других СУБД). Вот почему неплохо было бы организовать поиск в документации по вашей СУБД с использованием ключей поиска «ключевые слова» и «зарезервированные слова». Имейте в виду, что Microsoft SQL Server, Oracle, MySQL и PostgreSQL разрешают встроенные сопроводительные и много строчные комментарии, которые необходимо помещать между символами **/*** и ***/**. Организуйте поиск в документации по ключам «комментарии». Имейте в виду, что комментарии MySQL могут начинаться со знака **#**.

Таблица 2.3. Типы выражений

Выражение	Пример
`Case` (Выбор)	`CASE WHEN n <> 0 THEN x/n ELSE 0 END`
`Cast` (Преобразование типов)	`CAST(pubdate AS CHARACTER)`
`Datatime` (Дата или время)	`start_time + '01:30'`
`Interval` (Интервал между датами или между отметками времени суток)	`INTERVAL '7' DAY*2`
`Numeric` (Числовой)	`(sales*price)/12`
`String` (Строковый)	`'Dear ' || au_fname || ','`

Типы данных SQL

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

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 обеспечить совместимость SQL с другими языками программирования.

ТИПЫ ДАННЫХ, РАСПОЗНАВАЕМЫЕ С ПОМОЩЬЮ ANSI, состоят из символов и различных типов чисел, которые могут классифицироваться как *точные* числа и *приблизительные* числа.

Точные числовые типы – это номера, с десятичной точкой или без десятичной точки.

Приблизительные числовые типы – это номера в показательной (экспоненциальной по основанию 10) записи.

Для всех прочих типов, отличия слишком малы, чтобы их как-то классифицировать.

Иногда типы данных используют аргумент, который я называю *размером аргумента*, чей точный формат и значение меняется в зависимости от конкретного типа.

Значения по умолчанию обеспечены для всех типов, если размер аргумента отсутствует.

ТИПЫ ANSI

Ниже представлены типы данных ANSI (имена в круглых скобках – это синонимы):

TEXT
ТЕКСТ

CHAR (или CHARACTER)

Строка текста в реализационно-определенном формате. Размер аргумента здесь это единственное неотрицательное целое число, которое ссылается к максимальной длине строки. Значения этого типа, должны быть заключены в одиночные кавычки, например 'text'. Две рядом стоящие одиночные кавычки ('') внутри строки будет пониматься как одна одиночная кавычка (').

ПРИМЕЧАНИЕ: фраза *Реализационно-определенный* или Реализационно-зависимый, указывает, что этот аргумент или формат зависит от конкретной программы, в которой реализуются данные.

EXACT NUMERIC
ТОЧНОЕ ЧИСЛО

DEC (или DECIMAL)

Десятичное число; то есть, число которое может иметь десятичную точку. Здесь аргумент размера имеет две части: точность и масштаб. Масштаб не может превышать точность.

Сначала указывается точность, разделительная запятая и далее аргумент масштаба. Точность указывает, сколько значащих цифр имеет число. Максимальное десятичное число, составляющее номер – реализационно-определенное значение, равное или большее чем этот номер. Масштаб указывает максимальное число цифр справа от десятичной точки. Масштаб = 0 делает поле эквивалентом целого числа.

NUMERIC

Такое же, как DECIMAL за исключением того, что максимальное десятичное не может превышать аргумента точности.

INT (или INTEGER)

Число без десятичной точки. Эквивалентно DECIMAL, но без цифр справа от десятичной точки, то есть с масштабом равным 0. Аргумент размера не используется (он автоматически устанавливается в реализационно-зависимое значение).

SMALLINT

Такое же, как INTEGER, за исключением того, что, в зависимости от реализации, размер по умолчанию может (или не может) быть меньшее, чем INTEGER.

APPROXIMATE NUMERIC
ПРИБЛИЗИТЕЛЬНОЕ ЧИСЛО

FLOAT

Число с плавающей запятой на основе 10 показательной функции. Аргумент размера состоит из одного числа определяющего минимальную точность.

REAL

Такое же, как FLOAT, за исключением того, что никакого аргумента размера не используется. Точность установлена реализационно-зависимую по умолчанию.

DOUBLE PRECISION

Такое же, как REAL, за исключением того, что (или DOUBLE) реализационно-определяемая точность для DOUBLE PRECISION должна превышать реализационно-определяемую точность REAL.

Сложность числовых типов ANSI можно, по крайней мере частично, объяснить усилием сделать вложенный SQL, совместимым с рядом других языков.

Большинство коммерческих программ используют ещё и другие специальные типы. Такие как, **DATE** (ДАТА) и **TIME** (ВРЕМЯ) – фактически почти стандартные типы (хотя точный формат их меняется). Некоторые пакеты

также поддерживают такие типы, как например **MONEY** (ДЕНЬГИ) и **BINARY** (ДВОИЧНЫЕ). (MONEY – это специальная система исчисления, используемая компьютерами. Вся информация в компьютере передается двоичными числами и затем преобразовываются в другие системы, чтобы мы могли легко использовать их и понимать.)

Кроме этого, больше всего реализаций также имеют нестандартный тип **VARCHAR** (n). 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, не производится ее дополнение пробелами до заданного максимального значения.

Строковыми типами данных в рассматриваемых коммерческих СУБД являются:

- Microsoft Access – text, memo;
- Microsoft SQL Server – varchar, text, nchar, nvarchar, ntext;
- Oracle – char, varchar, varchar2, nchar, nvarchar, nvarchar2;
- MySQL – char, varchar, nchar, nvarchar, text, tinytext, mediumtext, longtext;
- PostgreSQL – char, varchar, text.

Глава 3

ИСПОЛЬЗОВАНИЕ SQL ДЛЯ ИЗВЛЕЧЕНИЯ ИНФОРМАЦИИ ИЗ ТАБЛИЦ

СОЗДАНИЕ ЗАПРОСА

Как мы подчеркивали ранее, SQL символизирует собой Структурированный Язык Запросов. Запросы – вероятно, наиболее часто используемый аспект SQL. Фактически, для категории SQL пользователей, маловероятно чтобы кто-либо использовал этот язык для чего-то другого. По этой причине, мы будем начинать наше обсуждение SQL с обсуждения запроса и как он выполняется на этом языке.

ЧТО ТАКОЕ ЗАПРОС?

Запрос – команда, которую вы даете вашей программе базы данных, и которая сообщает ей, чтобы она вывела определенную информацию из таблиц в память. Эта информация обычно посылается непосредственно на экран компьютера или терминала, которым вы пользуетесь, хотя, в большинстве случаев, ее можно также послать принтеру, сохранить в файле (как объект в памяти компьютера), или представить как вводную информацию для другой команды или процесса.

ГДЕ ПРИМЕНЯЮТСЯ ЗАПРОСЫ?

Запросы обычно рассматриваются как часть языка DML. Однако, так как запрос не меняет информацию в таблицах, а просто показывает ее пользователю, мы будем рассматривать запросы как самостоятельную категорию среди команд DML которые производят действие, а не просто показывают содержание базы данных.

Все запросы в SQL состоят из одиночной команды. Структура этой команды обманчиво проста, потому что вы должны расширять ее так, чтобы выполнить высоко сложные оценки и обработки данных. Эта команда называется – SELECT (ВЫБОР).

КОМАНДА SELECT

В самой простой форме, команда SELECT просто инструктирует базу данных, чтобы извлечь информацию из таблицы. Например, вы могли бы вывести таблицу Продавцов, напечатав следующее:

```
SELECT snum, sname, sity, comm
FROM Salespeople;
```

Вывод для этого запроса показывается в Рисунке 3.1.

```
=============== SQL Execution Log ============
  SELECT snum, sname, sity, comm
  FROM Salespeople;
| ==================================|
| snum   sname      city       comm |
| ----------------------------------|
| 1001   Peel       London      0.12 |
| 1002   Serres     San Jose    0.13 |
| 1004   Motika     London      0.11 |
| 1007   Rifkin     Barcelona   0.15 |
| 1003   Axelrod    New York    0.10 |
====================================
```

Рис. 3.1: Команда SELECT

Другими словами, эта команда просто выводит все данные из таблицы. Большинство программ будут также давать заголовки столбца как выше, а некоторые позволяют детальное форматирование вывода, но это уже вне стандартной спецификации.

Имеется объяснение каждой части этой команды:

SELECT – ключевое слово, которое сообщает базе данных, что эта команда – запрос. Все запросы начинаются этим словом, сопровождаемым пробелом.

snum, sname – это список столбцов из таблицы, которые выбираются запросом. Любые столбцы, не перечисленные здесь, не будут включены в вывод команды. Это, конечно, не значит, что они будут удалены или их информация будет стерта из таблиц, потому что запрос не воздействует на информацию в таблицах, он только показывает данные.

```
FROM
Salespeople
```

– ключевое слово, подобно SELECT, которое должно быть представлено в каждом запросе. Оно сопровождается пробелом и затем именем таблицы используемой в качестве источника информации. В данном случае – это таблица Продавцов (Salespeople).

; – точка с запятой используется во всех интерактивных командах SQL, чтобы сообщать базе данных, что команда заполнена и готова выполниться. В некоторых системах наклонная черта влево (\) в строке, является индикатором конца команды. Естественно, запрос такого характера не обязательно будет упорядочивать вывод любым указанным способом. Та же самая команда, выполненная с теми же самыми данными, но в разное время не сможет вывести тот же самый порядок. Обычно, строки обнаруживаются в том порядке, в котором они найдены в таблице – этот порядок произволен. Это не обязательно будет тот порядок, в котором данные вводились или сохранялись. Вы можете упорядочивать вывод командами SQL непосредственно, с помощью специального предложения. Позже, мы покажем, как это делается. А сейчас, просто усвойте, что в отсутствии явного упорядочения, нет никакого определенного порядка в вашем выводе.

Наше использование возврата (Клавиша ENTER) является произвольным. Мы должны точно установить, как удобнее составить запрос, в несколько строк или в одну троку, следующим образом:

```
SELECT snum, sname, city, comm FROM Salespeople;
```

С тех пор как SQL использует точку с запятой, чтобы указывать конец команды, большинство программ SQL обрабатывают возврат (через клавишу ENTER) как пробел. Это хорошая идея, чтобы использовать возвраты и выравнивание, чтобы сделать ваши команды более легкими для чтения и более правильными.

ВЫБИРАЙТЕ ВСЕГДА САМЫЙ ПРОСТОЙ СПОСОБ

Если вы хотите видеть каждый столбец таблицы, имеется необязательное сокращение, которое вы можете использовать. Звездочка (*) может применяться для вывода полного списка столбцов следующим образом:

```
SELECT *
FROM Salespeople;
```

Это приведет к тому же результату, что и наша предыдущая команда.

ОПИСАНИЕ SELECT

В общем случае, команда SELECT начинается с ключевого слова SELECT, сопровождаемого пробелом. После этого должен следовать список имен столбцов, которые вы хотите видеть, отделяемые запятыми. Если вы хотите видеть все столбцы таблицы, вы можете заменить этот список звездочкой (*). Ключевое слово FROM следующее далее, сопровождается пробелом и именем таблицы, запрос к которой делается. В заключение, точка с запятой (**;**) должна

использоваться, чтобы закончить запрос и указать что команда готова к выполнению.

ПРОСМОТР ТОЛЬКО ОПРЕДЕЛЕННОГО СТОЛБЦА ТАБЛИЦЫ

Команда SELECT способна извлечь строго определенную информацию из таблицы. Сначала, мы можем предоставить возможность увидеть только определенные столбцы таблицы. Это выполняется легко, простым исключением столбцов, которые вы не хотите видеть, из части команды SELECT. Например, запрос

```
SELECT sname, comm
FROM Salespeople;
```

будет производить вывод, показанный на Рисунке 3.2.

```
=============== SQL Execution Log ============
  SELECT snum, comm
  FROM Salespeople;
| ================|
| sname      comm |
| ----------------|
| Peel       0.12 |
| Serres     0.13 |
| Motika     0.11 |
| Rifkin     0.15 |
| Axelrod    0.10 |
===================
```

Рис. 3.2: Выбор определенных столбцов

Могут иметься таблицы, которые имеют большое количество столбцов, содержащих данные, не все из которых являются относящимися к поставленной задаче. Следовательно, вы можете найти способ подбора и выбора только полезных для Вас столбцов.

ПЕРЕУПОРЯДОЧЕНИЕ СТОЛБЦА

Даже если столбцы таблицы, по определению, упорядочены, это не означает, что вы будете восстанавливать их в том же порядке. Конечно, звездочка (*) покажет все столбцы в их естественном порядке, но если вы укажете столбцы отдельно, вы можете получить их в том порядке, в котором хотите. Давайте рассмотрим таблицу Порядков, содержащую дату приобретения (odate), номер продавца (snum), номер порядка (onum), и суммы приобретения (amt):

```
SELECT odate, snum, onum, amt
FROM Orders;
```

Вывод этого запроса показан на Рисунке 3.3.

```
============== SQL Execution Log ==============
  SELECT odate, snum, onum, amt
  FROM Orders;
| ---------------------------------|
| odate       snum    onum    amt    |
```

```
| ----------- ------- ------ --------|
| 10/03/1990  1007    3001     18.69 |
| 10/03/1990  1001    3003    767.19 |
| 10/03/1990  1004    3002   1900.10 |
| 10/03/1990  1002    3005   5160.45 |
| 10/03/1990  1007    3006   1098.16 |
| 10/04/1990  1003    3009   1713.23 |
| 10/04/1990  1002    3007     75.75 |
| 10/05/1990  1001    3008   4723.00 |
| 10/06/1990  1002    3010   1309.95 |
| 10/06/1990  1001    3011   9891.88 |
=====================================
```

Рис. 3.3: Реконструкция столбцов

Как вы можете видеть, структура информации в таблицах – это просто основа для активной перестройки структуры в SQL.

УДАЛЕНИЕ ИЗБЫТОЧНЫХ ДАННЫХ

DISTINCT (ОТЛИЧИЕ) – аргумент, который обеспечивает Вас способом устранять двойные значения из вашего предложения SELECT. Предположим, что вы хотите знать, какие продавцы в настоящее время имеют свои порядки в таблице Порядков. Под порядком (здесь и далее) будет пониматься запись в таблицу Порядков, регистрирующую приобретения, сделанные в определенный день определенным заказчиком у определенного продавца на определенную сумму). Вам не нужно знать, сколько порядков имеет каждый; вам нужен только список номеров продавцов (snum). Поэтому Вы можете ввести:

```
SELECT snum
FROM Orders;
```

для получения вывода, показанного в Рисунке 3.4.

```
=============== SQL Execution Log ============
  SELECT snum
  FROM Orders;
| ==== |
| snum |
| -----|
| 1007 |
| 1001 |
| 1004 |
| 1002 |
| 1007 |
| 1003 |
| 1002 |
| 1001 |
| 1002 |
| 1001 |
========
```

Рис. 3.4: SELECT с дублированием номеров продавцов

Для получения списка без дубликатов, для удобочитаемости, вы можете ввести следующее:

```
SELECT DISTINCT snum
FROM Orders;
```

Вывод для этого запроса показан в Рисунке 3.5.

Другими словами, DISTINCT следит за тем, какие значения были ранее, так чтобы они не были продублированы в списке. Это полезный способ избежать избыточности данных, но важно, чтобы при этом вы понимали, что вы делаете. Если вы не хотите потерять некоторые данные, вы не должны безоглядно использовать DISTINCT, потому что это может скрыть какую-то проблему или какие-то важные данные. Например, вы могли бы предположить, что имена всех ваших заказчиков различны. Если кто-то помещает второго Clemens в таблицу Заказчиков, а вы используете SELECT DISTINCT cname, вы не будете даже знать о существовании двойника. Вы можете получить не того Clemens и даже не знать об этом. Так как вы не ожидаете избыточности, в этом случае вы не должны использовать DISTINCT.

ПАРАМЕТРЫ DISTINCT

DISTINCT может указываться только один раз в данном предложении SELECT. Если предложение выбирает многочисленные поля, DISTINCT опускает строки, где все выбранные поля идентичны. Строки, в которых некоторые значения одинаковы, а некоторые различны – будут сохранены. DISTINCT, фактически, приводит к показу всей строки вывода, не указывая полей (за исключением, когда он используется внутри агрегатных функций, как описано в Главе 6), так что нет никакого смысла чтобы его повторять.

```
=============== SQL Execution Log ============
  SELECT DISTINCT snum
  FROM Orders;
| =====|
| snum |
| ---- |
| 1001 |
| 1002 |
| 1003 |
| 1004 |
| 1007 |
========
```

Рис. 3.5: SELECT без дублирования

DISTINCT ВМЕСТО ALL

Вместо DISTINCT, вы можете указать ALL. Это будет иметь противоположный эффект, дублирование строк вывода сохранится. Так как это тот же самый случай, когда вы не указываете ни DISTINCT ни ALL, то ALL по существу скорее пояснительный, а не действующий аргумент.

КВАЛИФИЦИРОВАННЫЙ ВЫБОР ПРИ ИСПОЛЬЗОВАНИИ ПРЕДЛОЖЕНИЙ

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

WHERE – предложение команды SELECT, которое позволяет вам устанавливать предикаты, условие которых может быть или верным или неверным для любой строки таблицы. Команда извлекает только те строки из таблицы, для которых такое утверждение верно. Например, предположим вы хотите видеть имена и комиссионные всех продавцов в Лондоне. Вы можете ввести такую команду:

```
SELECT sname, city
FROM Salespeople
WHERE city = "LONDON";
```

Когда предложение WHERE представлено, программа базы данных просматривает всю таблицу по одной строке и исследует каждую строку, чтобы определить верно ли утверждение. Следовательно, для записи Peel, программа рассмотрит текущее значение столбца city, определит что оно равно "London", и включит эту строку в вывод. Запись для Serres не будет включена, и так далее. Вывод для вышеупомянутого запроса показан на Рисунке 3.6.

```
=============== SQL Execution Log ============
| SELECT sname, city |
| FROM Salespeople |
| WHERE city = 'London' |
| ========================================== |
| sname  city |
| ------- ---------- |
| Peel   London |
| Motika London |
==============================================
```

Рис. 3.6: SELECT с предложением WHERE

Давайте попробуем пример с числовым полем в предложении WHERE. Поле rating таблицы Заказчиков предназначено, чтобы разделять заказчиков на группы, основанные на некоторых критериях, которые могут быть получены в итоге через этот номер. Возможно это – форма оценки кредита или оценки, основанные на опыте предыдущих приобретений. Такие числовые коды могут быть полезны в реляционных базах данных как способ подведения итогов сложной информации. Мы можем выбрать всех заказчиков с рейтингом 100, следующим образом:

```
SELECT *
FROM Customers
WHERE rating = 100;
```

Одиночные кавычки не используются здесь потому, что оценка – это числовое поле. Результаты запроса показаны в Рисунке 3.7.

Предложение WHERE совместимо с предыдущим материалом в этой главе.

Другими словами, вы можете использовать номера столбцов, устранять дубликаты, или переупорядочивать столбцы в команде SELECT которая использует WHERE. Однако, вы можете изменять порядок столбцов для имен только в предложении SELECT, но не в предложении WHERE.

```
=============== SQL Execution Log ============
  SELECT *
  FROM Customers
  WHERE rating = 100;
| ================================== |
| cnum   cname   city   rating   snum |
| ---------------------------------- |
| 2001  Hoffman  London  100     1001 |
| 2006  Clemens  London  100     1001 |
| 2007  Pereira  Rome    100     1001 |
=======================================
```

Рис. 3.7: SELECT с числовым полем в предикате

РЕЗЮМЕ

Теперь вы знаете несколько способов заставить таблицу давать вам ту информацию, какую вы хотите, а не просто выбрасывать наружу все ее содержание. Вы можете переупорядочивать столбцы таблицы или устранять любую из них. Вы можете решать, хотите вы видеть дублированные значения или нет.

Наиболее важно то, что вы можете устанавливать условие, называемое *предикатом*, которое определяет или не определяет указанную строку таблицы из тысяч таких же строк, будет ли она выбрана для вывода.

Предикаты могут становиться очень сложными, предоставляя вам высокую точность в решении, какие строки вам выбирать с помощью запроса. Именно эта способность решать точно, что вы хотите видеть, делает запросы SQL такими мощными.

На следующих занятиях будут рассмотрены, в большей мере, особенности, которые расширяют мощность предикатов. Вам будут представлены операторы иные, чем те, которые используются в условиях предиката, а также способы объединения многочисленных условий в единый предикат.

Глава 4

ИСПОЛЬЗОВАНИЕ РЕЛЯЦИОННЫХ И БУЛЕВЫХ ОПЕРАТОРОВ ДЛЯ СОЗДАНИЯ БОЛЕЕ ИЗОЩРЕННЫХ ПРЕДИКАТОВ

Ранее вы узнали, что предикаты могут оценивать равенство оператора как верного или неверного. Они могут также оценивать другие виды связей кроме равенств. Эта глава будет исследовать другие реляционные операторы,

используемые в SQL. Вы также узнаете, как использовать операторы Буля, чтобы изменять и объединять значения предиката. С помощью операторов Буля (или, проще говоря, логических операторов), одиночный предикат может содержать любое число условий. Это позволяет вам создавать очень сложные предикаты. Использование круглых скобок в структуре этих сложных предикатов будет также объясняться.

РЕЛЯЦИОННЫЕ ОПЕРАТОРЫ

Реляционный оператор – математический символ, который указывает на определенный тип сравнения между двумя значениями. Вы уже видели, как используются равенства, такие как 2 + 3 = 5 или city = "London". Но также имеются другие реляционные операторы. Предположим, что вы хотите видеть всех Продавцов с их комиссионными выше определенного значения. Вы можете использовать тип сравнения "больше чем" – >).

Реляционные операторы которыми располагает SQL :

= *Равный*
> *Больше чем*
< *Меньше чем*
>= *Больше чем или равно*
<= *Меньше чем или равно*
<> *Не равно*

Эти операторы имеют стандартные значения для числовых значений. Для значения символа, их определение зависит от формата преобразования, **ASCII** или **EBCDIC**, который вы используете. SQL сравнивает символьные значения в терминах основных номеров как определено в формате преобразования. Даже значение символа, такого как "1", который представляет номер, не обязательно равняется номеру, который он представляет. Вы можете использовать реляционные операторы, чтобы установить алфавитный порядок, например, "a" < "n", где средство ***a*** первое в алфавитном порядке, но все это ограничивается с помощью параметра преобразования формата.

И в ASCII и в EBCDIC, символы – по значению: меньше чем все другие символы, которым они предшествуют в алфавитном порядке и имеют один вариант (верхний или нижний). В ASCII, все символы верхнего регистра – меньше, чем все символы нижнего регистра, поэтому "Z" < "a", а все номера – меньше чем все символы, поэтому "1" < "Z". То же относится и к EBCDIC. Чтобы сохранить обсуждение более простым, мы допустим, что вы будете использовать текстовый формат ASCII. Проконсультируйтесь с вашей документацией системы, если вы не уверены, какой формат вы используете или как он работает.

Значения, сравниваемые здесь, называются – *скалярными значениями*. Скалярные значения производятся скалярными выражениями; 1 + 2 – это скалярное выражение, которое производит скалярное значение 3. Скалярное значение может быть символом или числом, хотя очевидно, что только номера используются с арифметическими операторами, такими как + (плюс) или * (звезда).

Предикаты обычно сравнивают значения скалярных величин, используя или реляционные операторы или специальные операторы SQL чтобы увидеть, верно ли это сравнение. Некоторые операторы SQL описаны далее.

Предположим, что вы хотите увидеть всех заказчиков с оценкой (rating) выше 200. Так как 200 – это скалярное значение, как и значение в столбце оценки, для их сравнения вы можете использовать реляционный оператор.

```
SELECT *
FROM Customers
WHERE rating > 200;
```

Вывод для этого запроса показывается в Рисунке 4.1.

```
  SELECT *
  FROM Customers
  WHERE rating > 200;
| ======================================== |
| snum   cname      city      rating  snum |
| ---------------------------------------- |
| 2004   Crass      Berlin    300     1002 |
| 2008   Cirneros   San Jose  300     1007 |
============================================
```

Рис. 4.1: Использование больше чем (>)

Конечно, если бы мы захотели увидеть еще и заказчиков с оценкой, равной 200, мы стали бы использовать предикат

```
rating >= 200
```

БУЛЕВЫ ОПЕРАТОРЫ

Основные Булевы операторы также распознаются в SQL. Выражения Буля являются или верными или неверными, подобно предикатам. Булевы операторы связывают одно или более верных / неверных значений и производят единственное верное или неверное значение. Стандартными операторами Буля, распознаваемыми в SQL, являются: **AND**, **OR**, и **NOT**.

Существуют другие, более сложные, операторы Буля (типа "исключенный или"), но они могут быть сформированы из этих трех простых операторов – AND, OR, NOT.

Как вы можете понять, Булева верная / неверная логика – основана на цифровой компьютерной операции; и фактически, весь SQL (или любой другой язык) может быть сведен до уровня Булевой логики.

Операторы Буля и как они работают

AND берет два Буля (в форме A AND B) как аргументы и оценивает их по отношению к истине, верны ли они оба.

OR берет два Буля (в форме A OR B) как аргументы и оценивает на правильность, верен ли один из них.

NOT берет одиночный Булев (в форме NOT A) как аргументы и заменяет его значение с неверного на верное или верное на неверное.

Связывая предикаты с операторами Буля, вы можете значительно увеличить их возможности. Предположим, вы хотите видеть всех заказчиков в San Jose, которые имеют оценку (рейтинг) выше 200:

```
SELECT *
FROM Customers
WHERE city = “San Jose”
AND rating > 200;
```

Вывод для этого запроса показан на Рисунке 4.2. Имеется только один заказчик, который удовлетворяет этому условию.

```
=============== SQL Execution Log ============
  SELECT *
  FROM Customers
  WHERE city = 'San Jose'
  AND rating > 200;
| ===================================== |
| cnum  cname     city      rating  snum |
| ------------------------------------- |
| 2008  Cirneros  San Jose  300     1007 |
=========================================
```

Рис. `4.2: SELECT` использующий `AND`

Если вы же используете OR, вы получите всех заказчиков, которые находились в San Jose **или** (OR) которые имели оценку выше 200.

```
SELECT *
FROM Customers
WHERE city = “San Jose” OR rating > 200;
```

Вывод для этого запроса показывается в Рисунке 4.3.

```
=============== SQL Execution Log ============
  SELECT *
  FROM Customers
  WHERE city = 'San Jose'
  OR rating > 200;
| ===================================== |
| cnum  cname     city      rating snum |
| ------------------------------------- |
| 2003  Liu       San Jose  200    1002 |
| 2004  Grass     Berlin    300    1002 |
| 2008  Cirneros  San Jose  300    1007 |
=========================================
```

Рис. `4.3: SELECT` использующий `OR`

NOT может использоваться для инвертирования значений Буля. Имеется пример запроса с NOT:

```
SELECT *
FROM Customers
WHERE city = "San Jose" OR NOT rating > 200;
```

Вывод этого запроса показывается в Рисунке 4.4.

```
=============== SQL Execution Log ============
  SELECT *
  FROM Customers
  WHERE city = 'San Jose'
  OR NOT rating > 200;
| ================================== |
| cnum  cname     city      rating snum |
| ---------------------------------- |
| 2001  Hoffman   London    100    1001 |
| 2002  Giovanni  Rome      200    1003 |
| 2003  Liu       San Jose  200    1002 |
| 2006  Clemens   London    100    1001 |
| 2008  Cirneros  San Jose  300    1007 |
| 2007  Pereira   Rome      100    1004 |
=======================================
```

Рис. 4.4: SELECT использующий NOT

Обратите внимание, что оператор NOT должен предшествовать Булеву оператору, чье значение должно измениться, и не должен помещаться перед реляционным оператором. Например, неправильным вводом оценки предиката будет:

```
rating NOT > 200
```

А как SQL оценит следующее?

```
SELECT *
FROM Customers
WHERE NOT city = “San Jose” OR rating > 200;
```

NOT применяется здесь только к выражению **city = 'SanJose'**, или к выражению **rating > 200** тоже? Как и написано, правильный ответ будет прежним. SQL может применять NOT с выражением Буля только сразу после него. Вы можете получить другой результат при команде:

```
SELECT *
FROM Customers
WHERE NOT(city = “San Jose” OR rating > 200);
```

Здесь SQL понимает круглые скобки как означающие, что все внутри них будет оцениваться первым и обрабатываться как единое выражение с помощью всего, что снаружи них (это является стандартной интерпретацией в математике). Другими словами, SQL берет каждую строку и определяет, соответствует ли истине равенство **city = "San Jose'** или равенство **rating > 200**. Если любое условие верно, выражение Буля внутри круглых скобок верно. Однако, если выражение Буля внутри круглых скобок верно, предикат как единое целое неверен, потому что NOT преобразует верно в неверно и наоборот.

Вывод для этого запроса показывается в Рисунке 4.5.

```
=============== SQL Execution Log ============
  SELECT *
  FROM Customers
```

```
  WHERE NOT (city = 'San Jose'
  OR rating > 200);
| ====================================|
| cnum  cname       city  rating snum |
| ------------------------------------ |
| 2001  Hoffman    London  100   1001 |
| 2002  Giovanni   Rome    200   1003 |
| 2006  Clemens    London  100   1001 |
| 2007  Pereira    Rome    100   1004 |
=======================================
```

Рис. 4.5: SELECT использующий NOT и вводное предложение

Имеется намеренно сложный пример. Посмотрим, сможете ли вы проследить его логику (вывод показан на Рисунке 4.6):

```
SELECT *
FROM Orders
WHERE NOT ((odate = 10/03/1990 AND snum >1002) OR amt > 2000.00);


============== SQL Execution Log ==============
  SELECT *
  FROM Orders
  WHERE NOT ((odate = 10/03/1990 AND snum > 1002)
  OR amt > 2000.00);
========================================== |
| onum     amt      odate       cnum   snum |
| ------------------------------------- |
| 3003    767.19   10/03/1990   2001   1001 |
| 3009   1713.23   10/04/1990   2002   1003 |
| 3007     75.75   10/04/1990   2004   1002 |
| 3010   1309.95   10/06/1990   2004   1002 |
==========================================
```

Рис. 4.6: Полный (комплексный) запрос

Комбинации булевых операторов в сложных выражениях не столь просты, как каждый из в отдельности. Способ оценки сложного булева выражения следующий: оценить булево(ы) выражение(ия), имеющее(ие) наибольшую глубину вхождения в круглые скобки, скомбинировать результаты в одно булево выражение, а затем связать его значение со значениями выражений, имеющих меньшую глубину вхождения в круглые скобки.

Дадим детальное объяснение оценки рассмотренного выше примера. Наибольшую глубину вхождения в булево выражение имеет предикат: odate = 10/03/1990 and snum > 1002, со связкой AND, образующий булево выражение, которое оценивается как истинное для всех тех строк, которые удовлетворяют каждому из этих условий. Это составное булево выражение (которое мы назовем булево выражение номер 1 или, для краткости, Bl) соединено с amt > 2000.00 (выражение B2) с помощью OR и образует третье выражение (B3), которое является истинным для данной строки в том случае, если либо Bl либо B2 истинны для этой строки. B3 полностью содержится в

круглых скобках, которым предшествует NOT, и образует заключительное булево выражение (B4), которое является условием предиката. Следовательно, B4 – предикат запроса – истинен, если B3 ложен и наоборот. B3 ложен, если ложен каждый из Bl и B2. Bl ложен для строк, в которых либо odate не совпадает с заданным значением 10/03/1990, либо значение snum не превышает 1002. B2 ложен для всех строк, в которых значение поля amount не превосходит 2000.00. Любая строка с суммой, превышающей 2000.00, делает B2 истинным, отсюда B3 тоже истинно, а B4 – ложно. Следовательно, все такие строки исключаются из числа выходных данных. Остающиеся строки от 3 октября 1990 года с snum, превышающим 1002 (такой, например, является строка с onum 3001 за октябрь, 3, 1990 с snum 1007), делают B1 истинным, следовательно, и B3 истинно, а значит предикат ложен. Эти записи также исключаются из рассмотрения. Оставшиеся строки входят в состав выходных данных (см. рис. 4.6).

РЕЗЮМЕ

В этой главе более полно представлены сведения из области предикатов. Показано, как можно найти значения, которые связаны с данным значением любым количеством способов, заданных с помощью различных реляционных операторов, как применяются булевы операторы **AND** и **OR** для комбинации сложных условий, каждое из которых может рассматриваться как единственный предикат. Булев оператор **NOT** изменяет на противоположное значение условия или группы условий. Все булевы выражения и операторы отношения управляются с помощью круглых скобок, которые определяют порядок выполнения операции. Эти операции могут иметь любой уровень сложности. Было рассмотрено, как можно разложить записанное сложное выражение на составные части, каждая из которых является простой.

В главе 5 будут представлены особые операторы языка SQL.

Глава 5

ИСПОЛЬЗОВАНИЕ СПЕЦИАЛЬНЫХ ОПЕРАТОРОВ В УСЛОВИЯХ

В дополнении к реляционным и булевским операторам SQL использует специальные операторы **IN, BETWEEN, LIKE,** и **IS NULL.** Здесь рассмотрим, как их использовать и как реляционные операторы позволяют создавать более сложные и мощные предикаты. Обсуждение оператора IS NULL будет включать отсутствие данных и значение NULL, которое указывает на то, что данные отсутствуют. Вы также узнаете о разновидностях использования оператора NOT применяющегося с этими операторами.

ОПЕРАТОР IN

Оператор IN определяет набор значений, в которое данное значение может или не может быть включено. В соответствии с нашей учебной базой данных, на которой вы обучаетесь по настоящее время, если вы хотите найти всех продавцов, которые размещены в Barcelona или в London, вы должны использовать следующий запрос (вывод показывается в Рисунке 5.1):

```
SELECT *
FROM Salespeople
WHERE city = 'Barcelona' OR city = 'London';
```

```
================ SQL Execution Log ============
  SELECT *
  FROM Salespeople
  WHERE city = 'Barcelona'
  OR city = 'London';
| ================================|
| snum  sname     city       comm |
| ------------------------------ |
| 1001  Peel      London     0.12 |
| 1004  Motika    London     0.11 |
| 1007  Rifkin    Barcelona  0.15 |
==================================
```

Рис. 5.1: Нахождение продавцов в Барселоне и Лондоне

Имеется и более простой способ получить ту же информацию:

```
SELECT *
FROM Salespeople
WHERE city IN ( 'Barcelona', 'London' );
```

Вывод для этого запроса показывается в Рисунке 5.2.

```
================ SQL Execution Log ============
  SELECT *
  FROM Salespeople
  WHERE city IN ('Barcelona', 'London';
| ================================|
| snum  sname    city       comm |
| ------------------------------ |
| 1001  Peel     London     0.12 |
| 1004  Motika   London     0.11 |
| 1007  Rifkin   Barcelona  0.15 |
==================================
```

Рис. 5.2: SELECT использует IN

Как видно из примера, IN определяет множество, элементы которого точно перечисляются в круглых скобках и разделяются запятыми. Если в поле, имя которого указано слева от IN, есть одно из перечисленных в списке значений (требуется точное совпадение), то предикат считается истинным. Если элементы множества имеют числовой, а не символьный тип, то одиночные кавычки непосредственно слева и справа от значения необходимо опустить.

Можно найти всех покупателей, обслуживаемых продавцами 1001, 1007, 1004. Выходные данные для следующего запроса представлены на рис. 5.3:

```
SELECT *
FROM Customers
WHERE cnum IN ( 1001, 1007, 1004 );
```

```
=============== SQL Execution Log ============
  SELECT *
  FROM Customers
  WHERE snum IN (1001, 1007, 1004);
| ================================== |
| snum  cname      city    rating  snum |
| ---------------------------------- |
| 2001  Hoffman   London     100   1001 |
| 2006  Clemens   London     100   1001 |
| 2008  Cisneros  San Jose   300   1007 |
| 2007  Pereira   Rome       100   1004 |
=======================================
```

Рис. 5.3: SELECT использует IN с номерами

ОПЕРАТОР BETWEEN

Оператор **BETWEEN** похож на оператор IN. В отличие от определения по номерам из набора, как это делает IN, BETWEEN определяет диапазон, в который должны значения умещаться, что делает предикат верным. Вы должны ввести ключевое слово BETWEEN с начальным значением, ключевое **AND** и конечное значение. В отличие от IN, BETWEEN чувствителен к порядку, и первое значение в предложении должно быть первым по алфавитному или числовому порядку. Следующий пример будет извлекать из таблицы Продавцов всех продавцов с комиссионными между 0.10 и 0.12 (вывод показывается в Рисунке 5.4):

```
SELECT *
FROM Salespeople
WHERE comm BETWEEN .10 AND .12;
```

```
=============== SQL Execution Log ============
  SELECT *
  FROM Salespeople
  WHERE comm BETWEEN .10 AND .12;
| ================================|
| snum  sname     city      comm |
| ------------------------------ |
| 1001  Peel      London    0.12 |
| 1004  Motika    London    0.11 |
| 1003  Axelrod   New York  0.10 |
==================================
```

Рис. 5.4: SELECT использует BETWEEN

SQL не делает непосредственной поддержки невключения BETWEEN. Вы должны или определить ваши граничные значения так, чтобы включающая интерпретация была приемлема, или сделать что-нибудь типа этого:

```
SELECT *
FROM Salespeople
WHERE (comm BETWEEN 0.10, AND 0.12) AND NOT comm IN (0.10, 0.12);
```

Вывод для этого запроса показывается в Рисунке 5.5.

```
============== SQL Execution Log ============
  SELECT *
  FROM Salespeople
  WHERE (comm BETWEEN 0.10 AND 0.12)
  AND NOT comm IN (0.10, 0.12);
| ============================|
| snum  sname     city    comm |
| --------------------------- |
| 1004  Motika   London   0.11 |
================================
```

Рис. 5.5: Сделать BETWEEN – невключенным

Пусть эта запись и неуклюжа, но она показывает, как новые операторы можно комбинировать с булевыми операторами для получения более сложных предикатов. Значит, IN и BETWEEN используются, как и операторы сравнения, для сопоставления значений, одно из которых является множеством (для IN) или диапазоном (для BETWEEN).

Аналогично всем операторам сравнения, BETWEEN действует на символьных полях, представленных в двоичном (ASCII) эквиваленте, т.е. для выборки можно воспользоваться алфавитным порядком. Следующий запрос выбирает всех покупателей имена которых попадают в заданный алфавитный диапазон:

```
SELECT *
FROM Customers
WHERE cname BETWEEN 'A' AND 'G';
```

Вывод для этого запроса показывается в Рисунке 5.6.

```
============== SQL Execution Log ============
  SELECT *
  FROM Customers
  WHERE cname BETWEEN 'A' AND 'G';
| ================================== |
| cnum  cname      city    rating snum |
| ---------------------------------- |
| 2006  Clemens   London    100   1001 |
| 2008  Cisneros  San Jose  300   1007 |
======================================
```

Рис. 5.6: Использование BETWEEN в алфавитных порядках

Grass и Giovanni отсутствуют, несмотря на то, что BETWEEN является включающим, так как он сравнивает строки неравной длины. Строка 'G' короче строки 'Giovanni', поэтому BETWEEN дополняет 'G' пробелами. Пробелы

предшествуют символам латинского алфавита (в большинстве реализаций), поэтому Giovanni оказался невыбранным. Аналогично и для Grass. Помните об этом при использовании BETWEEN с алфавитными диапазонами. Для включения в результат выполнения запроса сведений о покупателях, фамилии которых начинаются на 'G', нужно указать следующую букву алфавита ('H') или приписать символ 'z' (несколько символов 'z', если это необходимо) после второго граничного значения.

ОПЕРАТОР LIKE

LIKE применим только к полям типа CHAR или VARCHAR, поскольку он используется для поиска подстрок. Другими словами, он осуществляет просмотр строки для выяснения: входит ли заданная подстрока в указанное поле. С этой же целью используются *шаблоны* – специальные символы, которые могут соответствовать чему-нибудь. Термин «шаблоны» в программистской среде иногда заменяется термином «групповые символы» (англ. *wildkards*).

Существует два типа шаблонов, используемых с LIKE:

- символ подчеркивания (_) замещает любой одиночный символ. Например, '**b_t**' будет соответствовать словам '**bat**' или '**bit**', но не будет соответствовать 'brat'.
- Символ "процент" (%) заменяет последовательность символов произвольной длины, в том числе и нулевой.. Например '**%p%t**' будет соответствовать словам '**put', 'posit',** или '**opt',** но не '**spite**'.

Можно найти всех заказчиков, чьи имена начинаются с G (вывод показывается в Рисунке 5.7):

```
SELECT
FROM Customers
WHERE cname LIKE 'G%';
```

```
=============== SQL Execution Log ============
  SELECT *
  FROM Customers
  WHERE cname LIKE 'G%';
| ================================= |
| cnum   cname    city  rating  snum |
| --------------------------------- |
| 2002  Giovanni  Rome    200   1003 |
| 2004  Grass     Berlin  300   1002 |
=====================================
```

Рис. 5.7: `SELECT` использует `LIKE` с %

LIKE может оказаться полезным при осуществлении поиска имени или другого значения, полное написание которого неизвестно. Предположим, не совсем понятно, как правильно записывается фамилия одного из продавцов (salespeople): Peal или Peel. Можно использовать ту часть, которая известна, и

символы шаблона для нахождения всех возможных вариантов (выходные данные для этого запроса представлены на рис. 5.8):

```
SELECT *
FROM Salespeople
WHERE sname LIKE 'P__l%';
```

Каждый символ подчеркивания в шаблоне представляет единственный символ, поэтому, например, имя Prettel не вошло бы в состав выходных данных. Символ шаблона (%) в конце строки необходим в тех реализациях SQL, в которых длина поля sname превосходит количество букв в имени Peel (здесь это очевидно, потому что другие значения превышают четыре символа). В таком случае значение поля sname реально хранится как Peel, за ним следует ряд пробелов. Следовательно, символ 'l' не является последним в строке. Символ (%) в шаблоне заменяет все пробелы. Всё вышеперечисленное не относится к полю sname типа VARCHAR.

```
=============== SQL Execution Log ============
  SELECT *
  FROM Salespeople
  WHERE sname LIKE 'P__l%';
| ========================|
| snum  sname city    comm |
| ----------------------- |
| 1001  Peel  London  0.12 |
============================
```

Рис. 5.8: SELECT использует LIKE с подчеркиванием (_)

Чтобы найти в строке символ подчеркивания или процента, в предикате LIKE любой символ можно определить как Escape-символ. Он используется в предикате непосредственно перед символом процента или подчеркивания и означает, что следующий за ним символ интерпретируется именно как обычный символ, а не как символ шаблона.

Например, поиск символа подчеркивания в столбце sname можно задать следующим образом:

```
SELECT *
FROM Salespeople
WHERE sname LIKE '%/_%'ESCAPE'/';
```

Для тех данных, которые хранятся в таблице в текущий момент времени, выходных данных нет, поскольку в именах продавцов нет подчеркиваний. Предложение ESCAPE определяет '/' как Escape-символ, который используется в LIKE-строке, за ним следуют символ процента, символ подчеркивания или сам символ '/', т.е. тот символ, поиск которого будет осуществляться в столбце и который уже не интерпретируется как символ шаблона. Escape-символ может быть единственным символом и применяться только к единственному символу, который следует за ним. В приведенном примере начальный и конечный символы процента являются символами шаблона, только символ подчеркивания представляет собой символ как таковой.

Escape-символ может использоваться и в своем собственном значении. Другими словами, если нужно найти в столбце Escape-символ, то его необходимо ввести дважды. Первый раз он действует как обычный Escape-символ и означает: "*следующий символ надо понимать буквально так, как он указан*", а второй раз указывает на то, что речь идет непосредственно об Escape-символе. Далее представлен пример поиска строки '_/' в столбце sname:

```
SELECT *
FROM Salespeople
WHERE sname LIKE '%/_//%'ESCAPE'/';
```

В этом случае выходных данных нет. Просматриваемая строка состоит из любой последовательности символов (%), за которыми следуют символ подчеркивания (/_), Escape-символ (//) и любая последовательность заключительных символов (%).

РАБОТА С НУЛЕВЫМИ (NULL) ЗНАЧЕНИЯМИ

Часто будут иметься записи в таблице, которые не имеют никаких значений для каждого поля, например, потому что информация не завершена, или потому что это поле просто не заполнялось. SQL учитывает такой вариант, позволяя вам вводить значение **NULL** (ПУСТОЙ) в поле, вместо значения. Когда значение поля равно NULL, это означает, что программа базы данных специально промаркировала это поле как не имеющее никакого значения для этой строки (или записи). Это отличается от просто назначения полю значения нуля или пробела, которые база данных будет обрабатывать также как и любое другое значение. Точно так же, как NULL не является техническим значением, оно не имеет и типа данных. Оно может помещаться в любой тип поля. Тем не менее, NULL в SQL часто упоминается как *нуль*.

Предположим, что вы получили нового заказчика, который ещё не был назначен продавцу. Чем ждать продавца, к которому его нужно назначить, вы можете ввести заказчика в базу данных теперь же, так что он не потеряется при перестановке. Вы можете ввести строку для заказчика со значением NULL в поле snum и заполнить это поле значением позже, когда продавец будет назначен.

NULL ОПЕРАТОР

Так как NULL указывает на отсутствие значения, вы не можете знать, каков будет результат любого сравнения с использованием NULL. Когда NULL сравнивается с любым значением, даже с другим таким же NULL, результат будет ни верным ни неверным, он – неизвестен. Неизвестный Булев, вообще ведет себя так же, как неверная строка, которая произведя неизвестное значение в предикате не будет выбрана запросом – имейте ввиду, что в то время как NOT (неверное) – равняется верно, NOT (неизвестное) – равняется неизвестно. Следовательно, выражение типа 'city = NULL' или 'city IN (NULL)' будет неизвестно, независимо от значения city. Часто вы должны делать различия между неверно и неизвестно – между строками, содержащими

значения столбцов, которые не соответствуют условию предиката и которые содержат NULL в столбцах. По этой причине, SQL предоставляет специальный оператор **IS**, который используется с ключевым словом NULL, для размещения значения NULL. Найдём все записи в нашей таблице Заказчиков с NULL значениями в city столбце:

```
SELECT *
FROM Customers
WHERE city IS NULL;
```

Здесь не будет никакого вывода, потому что мы не имеем никаких значений NULL в наших типовых таблицах. Значения NULL – очень важны, и мы вернемся к ним позже.

ИСПОЛЬЗОВАНИЕ NOT СО СПЕЦИАЛЬНЫМИ ОПЕРАТОРАМИ

Специальные операторы, которые мы изучали в этой главе, могут немедленно предшествовать Булеву NOT.

Он противоположен реляционным операторам, которые должны иметь оператор NOT вводимым выражением. Например, если мы хотим устранить NULL из нашего вывода, мы будем использовать NOT, чтобы изменить на противоположное значение предиката:

```
SELECT *
FROM Customers
WHERE city IS NOT NULL;
```

При отсутствии значений NULL (как в нашем случае), будет выведена вся таблица Заказчиков. Аналогично можно ввести следующее:

```
SELECT *
FROM Customers
WHERE NOT city IS NULL;
```

что также приемлемо.

Мы можем также использовать NOT с **IN**:

```
SELECT *
FROM Salespeople
WHERE city NOT IN ( 'London', 'San Jose' );
```

А это – другой способ подобного же выражения:

```
SELECT *
FROM Salespeople
WHERE NOT city IN ( 'London', ' San Jose' );
```

Вывод для этого запроса показывается в Рисунке 5.9.

```
============== SQL Execution Log ============
  SELECT *
  FROM Salespeople
  WHERE city NOT IN ('London', 'San Jose';
| ============================|
| snum  sname    city       comm |
```

```
| ---------------------------- |
| 1003  Rifkin   Barcelona  0.15 |
| 1007  Axelrod  New York   0.10 |
================================
```

Рис. 5.9: Использование NOT с IN

Таким же способом вы можете использовать **NOT BETWEEN** и **NOT LIKE**.

РЕЗЮМЕ

Теперь вы можете создавать предикаты в терминах связей, специально определенных SQL. Вы можете искать значения в определенном диапазоне (**BETWEEN**) или в числовом наборе (**IN**), или вы можете искать символьные значения, которые соответствуют тексту внутри параметров (**LIKE**).

Вы также изучили некоторые вещи относительно того, как SQL поступает при отсутствии данных – что реальность мировой базы данных – используя NULL вместо конкретных значений. Вы можете извлекать или исключать значения NULL из вашего вывода, используя оператор **IS NULL**.

Теперь, когда вы имеете в вашем распоряжении весь набор стандартных математических и специальных операторов, вы можете переходить к специальным функциям SQL, которые работают на всех группах значений, а не просто на одиночном значении, что важно. Это уже тема Главы 6.

Глава 6

ОБОБЩЕНИЕ ДАННЫХ С ПОМОЩЬЮ АГРЕГАТНЫХ ФУНКЦИЙ

В этой главе, вы перейдете от простого использования запросов к извлечению значений из базы данных и определению, как вы можете использовать эти значения, чтобы получить из них информацию. Это делается с помощью *агрегатных* или общих функций, которые берут группы значений из поля и сводят их до одиночного значения. Вы узнаете, как использовать эти функции, как определить группы значений, к которым они будут применяться, и как определить какие группы выбираются для вывода. Вы будете также видеть, при каких условиях вы сможете объединить значения поля с этой полученной информацией в одиночном запросе.

ЧТО ТАКОЕ АГРЕГАТНЫЕ ФУНКЦИИ ?

Запросы могут производить обобщенное групповое значение полей точно также как и значение одного поля. Это делается с помощью агрегатных функций. Агрегатные функции производят одиночное значение для всей группы таблицы. Имеется список этих функций:

- **COUNT** вычисляет количество строк или не-NULL значения полей, которые выбрал запрос.

- **SUM** вычисляет арифметическую сумму всех выбранных значений данного поля.
- **AVG** вычисляет усреднение всех выбранных значений данного поля.
- **MAX** находит наибольшее из всех выбранных значений данного поля.
- **MIN** находит наименьшее из всех выбранных значений данного поля.

КАК ИСПОЛЬЗОВАТЬ АГРЕГАТНЫЕ ФУНКЦИИ ?

Агрегатные функции используются подобно именам полей в предложении SELECT запроса, но с одним исключением, они берут имена поля как аргументы. Только числовые поля могут использоваться с SUM и AVG.

С COUNT, MAX, и MIN могут использоваться и числовые или символьные поля. Когда они используются с символьными полями, MAX и MIN будут транслировать их в эквивалент ASCII, который должен сообщать, что MIN будет означать первое, а MAX последнее значение в алфавитном порядке (выдача алфавитного упорядочения обсуждается более подробно в Главе 4).

Чтобы найти SUM всех наших покупок в таблицы Порядков, мы можем ввести следующий запрос, с его выводом в Рисунке 6.1:

```
SELECT SUM (amt)
FROM Orders;
```

```
=============== SQL Execution Log ============
  SELECT SUM (amt)
  FROM Orders;
| ========|
|         |
| ------- |
| 26658.4 |
===========
```

Рис. 6.1: Вычисление суммы

Это, конечно, отличается от выбора поля, при котором возвращается одиночное значение, независимо от того, сколько строк находится в таблице. Из-за этого, агрегатные функции и поля не могут выбираться одновременно, пока предложение **GROUP BY** (описанное далее) не будет использовано. Нахождение усреднённой суммы – это похожая операция (вывод следующего запроса показывается в Рисунке 6.2):

```
SELECT AVG (amt)
FROM Orders;
```

```
=============== SQL Execution Log ============
  SELECT AVG (amt)
  FROM Orders;
| ========|
|         |
| ------- |
| 2665.84 |
===========
```

Рис. 6.2: Вычисление среднего

СПЕЦИАЛЬНЫЕ АТРИБУТЫ COUNT

Функция **COUNT** отличается от предыдущих тем, что подсчитывает количество значений в данном столбце или количество строк в таблице. Когда подсчитываются значения по столбцу, в команде используется DISTINCT для подсчета числа различных значений данного поля. Можно использовать его, например, для подсчета количества продавцов, имеющих в настоящее время заказы в таблице Orders (выходные данные представлены на рис. 6.3):

```
SELECT COUNT ( DISTINCT snum )
FROM Orders;
```

ИСПОЛЬЗОВАНИЕ DISTINCT

В данном примере DISTINCT вместе со следующим за ним именем поля, к которому он применяется, заключен в круглые скобки и не следует непосредственно за SELECT, как это было в примере главы 3. Такая форма применения DISTINCT с COUNT к отдельным столбцам предписывается стандартом ANSI, но многие программы не придерживаются этого требования. Можно использовать множество COUNT для DISTINCT полей в одном запросе; этот случай, рассмотренный в главе 3, отличается от случая применения DISTINCT к строкам.

Указанным способом DISTINCT можно применять с любой функцией агрегирования, но чаще всего он используется с COUNT. Применение его с MAX и MIN бесполезно; а, используя SUM и AVG, необходимо включение в выходные данные повторяющихся значений, так как они влияют на сумму и среднее для значений всех столбцов.

```
=============== SQL Execution Log ============
  SELECT COUNT (DISTINCT snum)
  FROM Orders;
| ==|
|   |
| --|
| 5 |
=====
```

Рис. 6.3: Подсчет значений поля

ИСПОЛЬЗОВАНИЕ COUNT СО СТРОКАМИ, А НЕ СО ЗНАЧЕНИЯМИ

Чтобы подсчитать общее число строк в таблице, используйте функцию COUNT со звездочкой вместо имени поля, как, например, в следующем примере, выходные данные для которого представлены на рис. 6.4:

```
SELECT COUNT (*)
FROM Customers
```

```
=============== SQL Execution Log ============
  SELECT COUNT (*)
  FROM Customers;
| ==|
|   |
| --|
```

```
| 7 |
=====
```

Рис. 6.4: Подсчет строк вместо значений

COUNT со звездочкой включает как NULL-значения, так и повторяющиеся значения, значит DISTINCT в этом случае не применим. По этой причине в результате получается число, превышающее COUNT для отдельного поля, который исключает из этого поля все избыточные строки или NULL-значения. DISTINCT исключен для COUNT(*), поскольку он не имеет смысла для хорошо спроектированной и управляемой базы данных. В такой базе данных не должно быть ни строк, содержащих в каждом поле только NULL-значения, ни полностью повторяющихся строк (поскольку первые не содержат никаких данных, а последние полностью избыточны).

С другой стороны, если имеются избыточные или содержащие одни NULL-значения строки, то нет необходимости применять COUNT для избавления от этой информации.

ИСПОЛЬЗОВАНИЕ ДУБЛИКАТОВ В АГРЕГАТНЫХ ФУНКЦИЯХ

Агрегатные функции могут также (во многих реализациях) иметь аргумент ALL, который размещается перед именем поля, как и DISTINCT, но обозначает противоположное: включить дубликаты. Требования ANSI не допускают подобного для COUNT, но многие реализации игнорируют это ограничение. Различие между ALL и * при использовании COUNT заключается в следующем:

- ALL использует имя поля в качестве аргумента;
- ALL не подсчитывает NULL-значения.

Поскольку * является единственным аргументом, который включает NULL-значения и используется только с COUNT, функции, отличные от COUNT, игнорируют NULL-значения в любом случае. Следующая команда осуществляет подсчет количества значений поля rating, отличных от NULL-значений, в таблице Customers (включая повторения):

```
SELECT COUNT ( ALL rating )
FROM Customers;
```

АГРЕГАТЫ, ПОСТРОЕННЫЕ НА СКАЛЯРНОМ ВЫРАЖЕНИИ

До сих пор были использованы агрегатные функции с одним полем в качестве аргумента. Можно использовать агрегатные функции с аргументами, которые состоят из скалярных выражений, включающих одно поле или большее количество полей. (При этом не разрешается применять DISTINCT.) Предположим, таблица Orders содержит дополнительный столбец с величиной предыдущего баланса (blnc) для каждого покупателя. Можно найти текущий баланс, добавив значение поля amount (amt) к значению поля blnc. Можно найти наибольшее значение текущего баланса:

```
SELECT MAX ( blnc + amt )
FROM Orders;
```

В процессе выполнения этого запроса для каждой строки таблицы выполняется сложение значений двух указанных полей записи и выбирается наибольшее из полученных значений. Конечно, поскольку покупатели могут иметь несколько заказов, их окончательный баланс в данном случае оценивается отдельно для каждого заказа.

Предполагается, что последняя заявка имеет наибольшее значение баланса данного покупателя. В противном случае в предыдущем примере мог быть выбран старый баланс. В SQL можно часто использовать скалярное выражение вместе с полями или вместо них.

ПРЕДЛОЖЕНИЕ GROUP BY

Предложение **GROUP BY** позволяет вам определять подмножество значений в особом поле в терминах другого поля, и применять функцию агрегата к подмножеству.

Это дает вам возможность объединять поля и агрегатные функции в едином предложении SELECT.

Например, предположим, что нужно найти наибольший заказ из тех, что получил каждый из продавцов. Можно сделать отдельный запрос на каждого продавца, выбрав MAX (amt) для таблицы Orders для каждого значения поля snum и используя GROUP BY, однако, возможно объединить все в одной команде:

```
SELECT snum, MAX (amt)
FROM Orders
GROUP BY snum;
```

Вывод для этого запроса показывается в Рисунке 6.5.

```
=============== SQL Execution Log ==============
  SELECT snum, MAX (amt)
  FROM Orders
  GROUP BY snum;
| ============= |
| snum          |
| ------------- |
| 1001   767.19 |
| 1002  1713.23 |
| 1003    75.75 |
| 1004  1309.95 |
| 1007  1098.16 |
=================
```

Рис. 6.5: Нахождение максимальной суммы продажи у каждого продавца

GROUP BY применяет агрегатные функции отдельно к каждой серии групп, которые определяются общим значением поля. В данном случае каждая группа состоит из всех тех строк, которые имеют одно и то же значение snum, а функция MAX применяется отдельно к каждой такой группе. Это означает, что поле, к которому применяется GROUP BY, по определению имеет на выходе только одно значение на каждую группу, что соответствует применению

агрегатных функций. Такая совместимость результатов и позволяет комбинировать агрегаты с полями указанным способом. Можно также применять GROUP BY с многозначными полями. Обращаясь к предыдущему примеру, можно предположить, что необходимо увидеть наибольший заказ, сделанный каждому продавцу на каждую дату. Для этого нужно сгруппировать данные таблицы Orders по дате внутри одного и того же поля Orders и применить функцию MAX к каждой группе. В результате будет получено:

```
SELECT snum, odate, MAX (amt)
FROM Orders
GROUP BY snum, odate;
```

Выходные данные для этого запроса представлены на рис. 6.6.

```
=============== SQL Execution Log ==============
  SELECT snum, odate, MAX (amt)
  FROM Orders
  GROUP BY snum, odate;
| ========================== |
| snum   odate               |
| -------------------------- |
| 1001  10/03/1990   767.19 |
| 1001  10/05/1990  4723.00 |
| 1001  10/06/1990  9891.88 |
| 1002  10/03/1990  5160.45 |
| 1002  10/04/1990    75.75 |
| 1002  10/06/1990  1309.95 |
| 1003  10/04/1990  1713.23 |
| 1004  10/03/1990  1900.10 |
| 1007  10/03/1990  1098.16 |
============================
```

Рис. 6.6: Нахождение наибольшей суммы приобретений на каждый день

Пустые группы, т.е. даты, когда данный продавец не получал заказов, в результате не представлены.

ПРЕДЛОЖЕНИЕ HAVING

Обращаясь к предыдущему примеру, можно предположить, что интересны только покупки, превышающие $3000.00. Однако использовать агрегатные функции в предложении WHERE нельзя (если только не применяется подзапрос, который будет объяснен позднее), поскольку предикаты оцениваются в терминах единственной строки, тогда как агрегатные функции оцениваются в терминах групп строк. Это значит, что нельзя формулировать запрос следующим образом:

```
SELECT snum, odate, MAX(amt)
FROM Orders
WHERE MAX(amt) > 3000.00
GROUP BY snum, odate;
```

Это неприемлемо с точки зрения точной интерпретации ANSI. Чтобы увидеть максимальную покупку, превышающую $3000.00, следует использовать предложение HAVING. Оно определяет критерий, согласно которому определенные группы исключаются из числа выходных данных, так же, как предложение WHERE делает это для отдельных строк. Правильная команда выглядит так:

```
SELECT snum, odate, MAX(amt)
FROM Orders
GROUP BY snum, odate
HAVING MAX(amt) > 3000.00;
```

Выходные данные для этого запроса представлены на рис. 6.7.

```
=============== SQL Execution Log ==============
  SELECT snum, odate, MAX (amt)
  FROM Orders
  GROUP BY snum, odate
  HAVING MAX (amt) > 3000.00;
| ========================== |
| snum   odate               |
| ------ ---------- -------- |
| 1001  10/05/1990   4723.00 |
| 1001  10/06/1990   9891.88 |
| 1002  10/03/1990   5160.45 |
==============================
```

Рис. 6.7: Удаление групп агрегатных значений

Аргументы HAVING подчиняются тем же правилам, что и аргументы SELECT в команде, использующей GROUP BY, и должны иметь единственное значение для каждой выходной группы. Следующая команда некорректна:

```
SELECT snum, MAX(amt)
FROM Orders
GROUP BY snum
HAVING odate = 10/03/1988;
```

В предложении HAVING нельзя указывать поле odate, поскольку оно может иметь (и действительно имеет) более одного значения для каждой выходной группы. HAVING должно относиться только к агрегатам и полям, выбранным по GROUP BY. Вот корректный способ формулировки приведенного запроса (выходные данные представлены на рис. 6.8):

```
SELECT snum, MAX(amt)
FROM Orders
WHERE odate = 10/03/1990
GROUP BY snum;
```

```
=============== SQL Execution Log ==============
  SELECT snum, odate, MAX (amt)
  FROM Orders
  GROUP BY snum, odate;
| ================|
| snum            |
```

```
| ------ -------- |
| 1001    767.19  |
| 1002    5160.45  |
| 1014    1900.10  |
| 1007    1098.16  |
====================
```

Рис. 6.8: Максимальное значение суммы приобретений у каждого продавца на 03.10.1990

Поскольку odate не является и не может быть выбранным полем, значимость полученных здесь данных, конечно, менее очевидна, чем в некоторых других примерах. Выходные данные должны были бы содержать нечто вроде следующего предложения: "Вот наибольшие заявки на 3 октября". В главе 7 будет объяснено, как вставить текст в выходные данные.

HAVING может иметь только такие аргументы, у которых единственное значение для группы выходных данных. На практике чаще всего применяются агрегатные функции, но можно осуществлять выбор полей и с помощью GROUP BY. Например, можно взглянуть на самые большие заказы для Serres и Rifkin:

```
SELECT snum, MAX(amt)
FROM Orders
GROUP BY snum
HAVING snum IN (1002, 1007)
```

Выходные данные для этого запроса представлены на рис. 6.9.

```
=============== SQL Execution Log ==============
  SELECT snum, MAX (amt)
  FROM Orders
  GROUP BY snum
  HAVING snum IN (1002, 1007);
| ===============|
| snum           |
| ------ ------- |
| 1002   5160.45 |
| 1007   1098.16 |
==================
```

Рис. 6.9: Использование HAVING с GROUP BY полями

Не используйте вложенные агрегаты

В версии языка SQL, определяемой ANSI, нельзя применять агрегатную функцию с агрегатом в качестве аргумента. Предположим, нужно определить, в какой день было сделано наибольшее число заявок. Если ввести команду:

```
SELECT odate, MAX ( SUM (amt) )
FROM Orders
GROUP BY odate;
```

то она, вероятно, будет отвергнута. (Существуют реализации, которые не учитывают такие ограничения, что дает определенные преимущества, поскольку вложенность агрегатов может быть полезной, даже если она и

вызывает сомнения.) Например, в данной команде SUM должна быть применена к каждой odate-группе, а MAX – ко всем группам, причем она выдает единственное значение для всех этих групп. В то же время предложение GROUP BY предполагает, что должна быть одна строка выходных данных на каждую группу odate.

РЕЗЮМЕ

Теперь вы научились использовать запросы иначе. Возможность выводить, а не просто локализовать, значения очень важна и означает, что не надо отслеживать путь получения информации, если можно сформулировать запрос на ее вывод. Запрос дает результаты на текущий момент, тогда как таблица итогов и средних величин полезна на тот момент, когда она в последний раз обновлялась. Это не значит, что агрегатные функции во всех случаях могут полностью снять потребность проследить путь информации.

Агрегатные функции применимы к группам значений, определяемым предложением GROUP BY. Эти группы имеют общее значение поля и могут использоваться внутри других групп, имеющих общее значение поля. Предикаты нужны для определения строк, к которым применяется функция агрегирования. Такое комбинирование позволяет получить агрегаты на основе полностью определенных подмножеств значений поля. Затем вы можете задать условие исключения определенных результирующих групп с помощью предложения HAVING.

Вы уже знаете, как запросы генерируют значения. В главе 7 будет показано, эти значения можно применять.

Глава 7

ФОРМАТИРОВАНИЕ РЕЗУЛЬТАТОВ ЗАПРОСОВ

Назначение этой главы – расширить возможности обработки результатов запросов. Вы узнаете, как вставить текст и константы в выбранные поля, как использовать последние в математических выражениях, результаты вычисления которых станут выходными данными и, наконец, как представить выходные данные в заданной последовательности. Последнее предполагает возможность упорядочить выходные данные по любому столбцу или по любым значениям, полученным на основе данных, содержащихся в столбце.

Строки и выражения

Во многих базах данных, использующих SQL, имеются специальные средства, позволяющие оформлять результаты запросов. Естественно, в разных программных продуктах они кардинально отличаются, но эти различия здесь не обсуждаются. Однако и стандартная версия SQL имеет ряд характерных свойств, позволяющих сделать нечто большее, чем просто вывести значения полей и функций агрегирования. О них и пойдет речь в данной главе.

Скалярные выражения с выбранными полями. Предположим, необходимо выполнить простые числовые операции с данными для представления их в более удобном виде. SQL позволяет вносить скалярные выражения и константы в выбранные поля. Эти выражения могут дополнять или заменять поля в предложениях SELECT и могут содержать множество выбранных полей. Например, если вам удобнее представить комиссионные продавцов в виде процентов, а не десятичных чисел, достаточно указать:

```
SELECT snum, sname, city, comm * 100
FROM Salespeople;
```

Выходные данные для этого запроса представлены на рис. 7.1.

```
=============== SQL Execution Log ============
  SELECT snum, sname, city, comm * 100
  FROM Salespeople;
| ====================================|
| snum   sname     city                |
| ------ --------- ----------- --------|
| 1001  Peel      London     12.000000 |
| 1002  Serres    San Jose   13.000000 |
| 1004  Motika    London     11.000000 |
| 1007  Rifkin    Barcelona  15.000000 |
| 1003  Axelrod   New York   10.000000 |
=======================================
```

Рис. 7.1: Использование выражения в запросе

Выходные столбцы. Последний столбец в предыдущем примере не имеет имени, поскольку является выходным столбцом. Выходные столбцы – это столбцы, которые создаются с помощью запроса (в тех случаях, когда в предложении запроса SELECT используются агрегатные функции, константы или выражения), а не извлекаются непосредственно из таблицы. Поскольку имена столбцов являются атрибутами таблицы, столбцы, не переходящие из таблицы в выходные данные, не имеют имен. Почти во всех ситуациях выходные столбцы отличаются от столбцов, извлекаемых из таблицы тем, что они не поименованы.

Внесение текста в выходные данные запроса. Буква 'A', не обозначающая ничего кроме самой себя, является константой, как и число 1. Константы, а также текст, можно включать в предложение запроса SELECT. Однако, буквенные константы, в отличие от числовых, нельзя использовать в выражениях. В SELECT предложение можно включить 1+2, но не 'A'+ 'B', поскольку 'A' и 'B' здесь просто буквы, а не переменные или символы, используемые для обозначения чего-либо отличного от них самих. Тем не менее, возможность вставить текст в выходные данные запроса вполне реальна.

Можно изменить предыдущий пример, пометив комиссионные, выраженные в процентах, символом "процент" (%), что позволяет представить их в выходных данных в виде символов и комментариев, например:

```
SELECT snum, sname, city, '%', comm * 100
FROM Salespeople;
```

```
=============== SQL Execution Log ============
  SELECT snum, sname, city, '%', comm * 100
  FROM Salespeople;
| ====================================|
| snum   sname    city                |
| ----------------------------------- |
| 1001  Peel     London    % 12.000000 |
| 1002  Serres   San Jose  % 13.000000 |
| 1004  Motika   London    % 11.000000 |
| 1007  Rifkin   Barcelona % 15.000000 |
| 1003  Axelrod  New York  % 10.000000 |
=======================================
```

Рис. 7.2: Включение символов в выходные данные

Выходные данные для этого запроса представлены на рис. 7 .2.

Аналогичный прием можно применить для того, чтобы пометить выходные данные, включив в них некоторый комментарий. Однако нужно помнить, что один и тот же комментарий будет печататься не один раз для всей таблицы, а в каждой строке выходных данных. Предположим, генерируются выходные данные для отчета, в котором фиксируется количество заказов на каждый день. Выходные данные можно пометить (см. рис. 7.3), оформив запрос следующим образом:

```
SELECT ' For ', odate, ', there are ',
COUNT ( DISTINCT onum ), 'orders.'
FROM Orders
GROUP BY odate;
```

```
=============== SQL Execution Log =============
  SELECT 'For', odate, ', there are ',
  COUNT (DISTINCT onum), ' orders.'
  FROM Orders
  GROUP BY odate;
| ====================================|
|     odate                           |
| ----------------------------------- |
| For 10/03/1990 , there are 5 orders. |
| For 10/04/1990 , there are 2 orders. |
| For 10/05/1990 , there are 1 orders. |
| For 10/06/1990 , there are 2 orders. |
=======================================
```

Рис. 7.3: Комбинация текста, значений полей и агрегатов

Грамматическую ошибку в выходных данных на 10/05/1990 можно исправить, но запрос при этом сильно усложнится. (Для этого пришлось бы использовать два запроса и операцию объединения UNION, которая будет рассмотрена в главе 14.) Вам может быть полезен единственный неизменный комментарий для каждой строки таблицы, но он ограничен. Иногда более элегантное и полезное решение состоит в том, чтобы выдать один и тот же комментарий для всех выходных данных в целом или разные комментарии для различных строк.

Многие программные продукты, использующие SQL, часто предоставляют пользователям генераторы отчетов, которые применяются для форматирования и улучшения формы выходных данных. Встроенный SQL тоже может употреблять средства форматирования того языка, в который он встроен. SQL предназначен прежде всего для обработки данных. Его выходными данными является информация, а программа, использующая SQL, может принимать эту информацию и выводить ее в более наглядной форме. Однако, это уже лежит за пределами самого SQL.

Упорядочение выходных полей

Таблицы являются неупорядоченными множествами, и исходящие из них данные необязательно представляются в какой-либо определенной последовательности. В SQL применяется команда ORDER BY, позволяющая внести некоторый порядок в выходные данные запроса. Она их упорядочивает в соответствии со значениями одного или нескольких выбранных столбцов. Множество столбцов упорядочиваются один внутри другого, как в случае применения GROUP BY, и можно задать возрастающую (ASC) или убывающую (DESC) последовательность сортировки для каждого из столбцов. По умолчанию принята возрастающая последовательность сортировки.

Таблица заявок (Orders), упорядоченная по номеру заявки, (обратить внимание на значения в столбце cnum) выглядит так:

```
SELECT *
FROM Orders
ORDER BY cnum DESC;
```

Выходные данные представлены на рис. 7.4.

```
=============== SQL Execution Log ==============
  SELECT *
  FROM Orders
  ORDER BY cnum DESC;
| ==================================== |
| onum     amt     odate     cnum  snum |
| ------------------------------------ |
| 3001    18.69  10/03/1990  2008  1007 |
| 3006  1098.16  10/03/1990  2008  1007 |
| 3002  1900.10  10/03/1990  2007  1004 |
| 3008  4723.00  10/05/1990  2006  1001 |
| 3011  9891.88  10/06/1990  2006  1001 |
| 3007    75.75  10/04/1990  2004  1002 |
| 3010  1309.95  10/06/1990  2004  1002 |
| 3005  5160.45  10/03/1990  2003  1002 |
| 3009  1713.23  10/04/1990  2002  1003 |
| 3003   767.19  10/03/1990  2001  1001 |
=========================================
```

Рис. 7.4: Упорядочение вывода с помощью убывания поля

Упорядочение по множеству столбцов

Внутри уже произведенного упорядочения по полю cnum можно упорядочить таблицу и по другому столбцу, например, amt (выходные данные представлены на рис. 7.5):

```
SELECT •
FROM Orders
ORDER BY cnum DESC, amt DESC;
```

```
=============== SQL Execution Log ==============
  SELECT *
  FROM Orders
  ORDER BY cnum DESC, amt DESC;
| ==================================== |
| onum     amt      odate    cnum  snum |
| ------------------------------------ |
| 3006  1098.16  10/03/1990  2008  1007 |
| 3001    18.69  10/03/1990  2008  1007 |
| 3002  1900.10  10/03/1990  2007  1004 |
| 3011  9891.88  10/06/1990  2006  1001 |
| 3008  4723.00  10/05/1990  2006  1001 |
| 3010  1309.95  10/06/1990  2004  1002 |
| 3007    75.75  10/04/1990  2004  1002 |
| 3005  5160.45  10/03/1990  2003  1002 |
| 3009  1713.23  10/04/1990  2002  1003 |
| 3003   767.19  10/03/1990  2001  1001 |
========================================
```

Рис. 7.5: Упорядочение выходных данных по множеству полей

Так можно использовать ORDER BY одновременно для любого количества столбцов. Во всех случаях столбцы, по которым выполняется сортировка, входят в число выбранных. Этому требованию стандарта ANSI удовлетворяет большинство систем. Например, следующая команда неверна:

```
SELECT cname, city
FROM Customers
ORDER BY cnum;
```

Поскольку поле cnum отсутствует в списке выбранных полей, предложение ORDER BY не может его найти для упорядочения выходных данных. Даже если система позволяет это сделать, значимость такого упорядочения неочевидна, поскольку само поле, по которому выполняется сортировка, не представлено в выходных данных. Поэтому включение в них всех столбцов, используемых в предложении ORDER BY, весьма желательно.

Упорядочение составных групп

ORDER BY может использоваться с GROUP BY для упорядочения групп. ORDER BY всегда выполняется последней. Вот пример из предыдущей главы с добавлением предложения ORDER BY. До этого выходные данные были

сгруппированы, но порядок групп был произвольным; теперь группы выстроены в определенной последовательности:

```
SELECT snum, odate, MAX(amt)
FROM Orders
GROUP BY snum, odate
ORDER BY snum;
```

Выходные данные представлены на рис. 7.6.

Поскольку в команде не указан способ упорядочения, по умолчанию применяется возрастающий.

```
=============== SQL Execution Log ==============
  SELECT snum, odate, MAX (amt)
  FROM Orders
  GROUP BY snum, odate
  ORDER BY snum ;
| ========================= |
| snum    odate       amt   |
| ------------------------- |
| 1001  10/06/1990   767.19 |
| 1001  10/05/1990  4723.00 |
| 1001  10/05/1990  9891.88 |
| 1002  10/06/1990  5160.45 |
| 1002  10/04/1990    75.75 |
| 1002  10/03/1990  1309.95 |
| 1003  10/04/1990  1713.23 |
| 1004  10/03/1990  1900.10 |
| 1007  10/03/1990  1098.16 |
=============================
```

Рис. 7.6: Упорядочение групп

Упорядочение результата по номеру столбца

Вместо имен столбцов для указания полей, по которым упорядочиваются выходные данные, можно использовать номера. Но, ссылаясь на них, следует иметь в виду, что это номера в определении выходных данных, а не столбцов в таблице. Т.е. первое поле, имя которого указано в SELECT, является для предложения ORDER BY полем с номером 1, независимо от его расположения в таблице. Например, можно применить следующую команду, чтобы увидеть определенные поля таблицы Salespeople, упорядоченные по убыванию поля commission (comm) (выходные данные представлены на рис. 7.7):

```
SELECT sname, comm
FROM Salespeople
ORDER BY 2 DESC;
```

```
=============== SQL Execution Log ============
  SELECT sname, comm
  FROM Salespeople
  ORDER BY 2 DESC;
| ============= |
```

```
| sname    comm |
| ------------- |
| Peel     0.17 |
| Serres   0.13 |
| Rifkin   0.15 |
=================
```

Рис. 7.7: Упорядочение с использованием номеров столбцов

Мы рассматриваем это свойство ORDER BY для того, чтобы продемонстрировать возможность его использования со столбцами выходных данных; эта процедура аналогична применению ORDER BY со столбцами таблицы. Столбцы, полученные с помощью функций агрегирования, константы или выражения в предложении запроса SELECT, можно применить и с ORDER BY, если на них ссылаются по номеру. Например, чтобы подсчитать заявки (orders) для каждого продавца (salespeople) и вывести результаты в убывающем порядке, как показано на рис. 7.8:

```
SELECT snum, COUNT (DISTINCT onum)
FROM Orders
GROUP BY snum
ORDER BY 2 DESC;
```

```
=============== SQL Execution Log ==============
  SELECT snum, COUNT (DISTINCT onum)
  FROM Orders
  GROUP BY snum
  ORDER BY 2 DESC;
| ======= |
| snum    |
| ------- |
| 1001  3 |
| 1002  3 |
| 1007  2 |
| 1003  1 |
| 1004  1 |
===========
```

Рис. 7.8: Упорядочение выходных столбцов

В этом случае был использован номер столбца, но так как выходной столбец не имеет имени, саму функцию агрегирования применять не понадобилось. В соответствии со стандартом ANSI SQL, следующий запрос не работает, хотя в некоторых системах он воспринимается без проблем:

```
SELECT snum, COUNT (DISTINCT onum)
FROM Orders
GROUP BY snum
ORDER BY COUNT (DISTINCT onum) DESC;
```

Многими системами такая команда воспринимается как ошибочная.

ORDER BY с NULL-значениями

Если в поле, которое используется для упорядочения выходных данных, существуют NULL-значения, то все они следуют в конце или предшествуют всем остальным значениям этого поля. Конкретный вариант не оговаривается стандартом ANSI, вопрос решается индивидуально для каждого программного продукта, и один из этих вариантов принимается.

РЕЗЮМЕ

Теперь с помощью запросов вы можете получить нечто большее, чем простые значения полей и функций агрегирования для данных, представленных в таблице. Значения полей используются в выражениях: например, можно умножить числовое поле на 10 или умножить его на другое числовое поле. Кроме того, константы, в том числе и символьные, можно включать в состав выходных данных. Все это дает возможность выводить текст непосредственно в результат запроса наравне с данными, содержащимися в таблице, что, в свою очередь, позволяет помечать или комментировать выходные данные различными способами.

Вы научились управлять порядком вывода результатов запроса. Несмотря на то, что сама таблица базы данных остаётся неупорядоченной, предложение ORDER BY позволяет управлять порядком вывода строк выходных данных конкретного запроса. Порядок представления выходных данных запроса может быть возрастающим или убывающим, и столбцы можно упорядочить один внутри другого.

В этой главе введено понятие выходных столбцов, которые можно применять для упорядочения результатов запроса, но эти столбцы не поименованы, поэтому для ссылки на них в предложении ORDER BY используется порядковый номер выходного столбца из предложения SELECT.

В главе 8 будут рассмотрены более сложные запросы. Вы узнаете, как объединить в одной единственной команде запросы к множеству таблиц базы данных, установив между ними связи.

Глава 8

ИСПОЛЬЗОВАНИЕ МНОЖЕСТВА ТАБЛИЦ В ОДНОМ ЗАПРОСЕ

Соединение таблиц

Одна из наиболее важных черт запросов SQL состоит в их способности определять связи между множеством таблиц и отображать содержащуюся в них информацию в терминах этих связей в рамках единственной команды. Операция такого рода называется соединение *(join)* и является одной из самых мощных операций для реляционных баз данных. Как уже говорилось, преимущество реляционного подхода заключается в связях (relationships), которые можно установить между элементами данных в таблице. С помощью

соединений непосредственно связывается информация, содержащаяся в таблицах, независимо от их числа, а также между отдельными частями любой таблицы.

При операции соединения таблицы перечисляются в предложении запроса FROM; имена таблиц разделяются запятыми. Предикат запроса может ссылаться на любой столбец любой из соединяемых таблиц и, следовательно, может использоваться для установления связей между ними. Обычно предикат сравнивает значения в столбцах различных таблиц для того, чтобы определить, удовлетворяется ли условие WHERE.

Имена таблиц и столбцов

Полное имя столбца состоит из имени таблицы, непосредственно за которым стоит точка, а за ней – имя столбца. Приведем несколько примеров:

```
Salespeople.snum
Customers.city
Orders.odate
```

В приводимых ранее примерах имена таблиц можно было опускать, поскольку запросы адресовались только к одной таблице, и SQL выполнял подстановку имени соответствующей таблицы в качестве префикса. Даже при формулировке запроса к множеству таблиц их имена можно опустить, если все столбцы этих таблиц различны. Однако так бывает далеко не всегда. Например, есть две простые таблицы с одинаковыми именами столбцов city. Если для них необходимо выполнить операцию соединения, то следует указать Salespeople.city или Customers.city, что дает возможность SQL однозначно определить, о каком столбце идет речь.

Выполнение операции соединения (join)

Предположим, нужно установить связь между продавцами (Salespeople) и покупателями (Customers) в соответствии с местом их проживания, чтобы получить все возможные комбинации продавцов и покупателей из одного города. Для этого необходимо взять продавца из таблицы Salespeople и выполнить по таблице Customers поиск всех покупателей, имеющих то же значение в столбце city. Это можно сделать, введя следующую команду (выходные данные представлены на рис. 8.1):

```
SELECT Customers.cname, Salespeople.sname, Salespeople.city
FROM Salespeople, Customers
WHERE Salespeople.city = Customers.city;
```

```
=============== SQL Execution Log ============
  SELECT Customers.cname, Salespeople.sname,
  Salespeople.city
  FROM Salespeople, Customers
  WHERE Salespeople.city = Customers.city
| ========================== |
```

```
| cname     sname   city     |
| ------------------------- |
| Hoffman   Peel    London   |
| Hoffman   Peel    London   |
| Liu       Serres  San Jose |
| Cisneros  Serres  San Jose |
| Hoffman   Motika  London   |
| Clemens   Motika  London   |
==============================
```

Рис. 8.1: Объединение двух таблиц

Поскольку поле city присутствует в каждой из таблиц Salespeople и Customers, имена таблиц используются перед именем city в качестве префиксов. Это необходимо в том случае, когда два или более поля имеют одинаковые имена, но для ясности и полноты картины полезно включать в соединения имя таблицы. В дальнейшем имена таблиц будут использоваться там, где это необходимо, чтобы было понятно, где они нужны, а где нет.

Выполняя операцию соединения, необходимо генерировать все возможные сочетания строк для двух или более таблиц и проверять истинность предиката на каждом таком сочетании. В предыдущем примере SQL берет строку, соответствующую продавцу Peel из таблицы Salespeople, и комбинирует ее с каждой строкой таблицы Customers, выбирая по одной строке из этой таблицы. Если на данной комбинации строк предикат имеет значение "истинно", т.е. поле city строки таблицы Customers содержит значение London такое же, как и у Peel, то указанные в предложении SELECT поля из комбинации этих строк являются выходными данными. Те же действия предпринимаются относительно каждого продавца из таблицы Salespeople (некоторые из них не имеют покупателей, находящихся в том же городе).

Операция соединения таблиц посредством ссылочной целостности

Эта операция применяется для использования связей, встроенных в базу данных. В предыдущем примере связь между таблицами была установлена с помощью операции соединения. Но эти таблицы уже связаны по значениям полем snum. Такая связь называется состоянием ссылочной целостности. Стандартное применение операции соединения состоит в извлечении данных в терминах этой связи. Чтобы показать соответствие имен покупателей именам продавцов, обслуживающих этих покупателей, используется следующий запрос:

```
SELECT Customers.cname, Salespeople.sname
FROM Customers, Salespeople
WHERE Salespeople.snum = Customers.snum;
```

Выходные данные для этого запроса представлены на рис. 8.2.

```
=============== SQL Execution Log ============
  SELECT Customers.cname, Salespeople.sname,
  FROM Salespeople, Customers
  WHERE Salespeople.snum = Customers.snum
| ================= |
```

```
| cname     sname   |
| ----------------- |
| Hoffman   Peel    |
| Giovanni  Axelrod |
| Liu       Serres  |
| Grass     Serres  |
| Clemens   Peel    |
| Cisneros  Rifkin  |
| Pereira   Motika  |
=====================
```

Рис. 8.2: Объединение продавцов с их покупателями

Это также пример соединения, в котором столбцы, используемые в формулировке предиката запроса, – в данном случае это столбцы snum в обеих таблицах – опущены из выходных данных. Выходные данные показывают, какие покупатели обслуживаются какими продавцами. Значения snum, на основании которых устанавливается связь, в данном случае не представлены, поскольку здесь они не являются существенными. Однако, действуя таким образом, нужно либо иметь уверенность в том, что выходные данные сами по себе ясны, либо дать им какие-то объяснения.

Эквисоединение и другие виды соединений

Соединение, использующее предикаты, основанные на равенствах, называется эквисоединением. Рассмотренные в данной главе примеры относятся именно к этой категории, поскольку все условия в предложении WHERE базируются на математических выражениях, использующих символ равенства. "City = 'London'" и "Salespeople.snum = Orders.snum" – примеры применения символа равенства в предикатах. Эквисоединение является, по-видимому, наиболее распространенным типом соединения, но существуют и другие. Фактически в соединении можно использовать любой оператор сравнения. Вот пример соединения другого рода (выходные данные для него представлены на рис. 8.3):

```
SELECT sname, cname
FROM Salespeople, Customers
WHERE sname < cname
   AND rating < 200;
```

```
=============== SQL Execution Log ============
  SELECT sname, cname
  FROM Salespeople, Customers
  WHERE sname < cname
  AND rating < 200;
| ================ |
| sname    cname   |
| ---------------- |
| Peel     Pereira |
| Motika   Pereira |
| Axelrod  Hoffman |
```

```
| Axelrod  Clemens |
| Axelrod  Pereira |
====================
```

Рис. 8.3: Объединение, основанное на неравенстве

Эта команда полезна далеко не всегда. Она генерирует все комбинации имен продавцов и покупателей так, что первые предшествуют последним в алфавитном порядке, а последние имеют рейтинг меньше чем 200. Обычно такие сложные связи нет необходимости конструировать, и поэтому вам полезно знать также и о других возможностях.

Соединение более чем двух таблиц

Можно конструировать запросы путем соединения более чем двух таблиц. Предположим, нужно найти все заявки покупателей, не находящихся в том же городе, что и их продавец. Для этого потребуется связать все три рассматриваемые таблицы (выходные данные представлены на рис. 8.4):

```
SELECT onum, cname, Orders.cnum, Orders.snum
FROM Salespeople, Customers, Orders
WHERE Customers.city <> Salespeople.city
   AND Orders.cnum = Customers.cnum
   AND Orders.snum = Salespeople.snum;
```

```
=============== SQL Execution Log ==============
  SELECT onum, cname, Orders.cnum, Orders.snum
  FROM Salespeople, Customers, Orders
  WHERE Customers.city <> Salespeople.city
     AND Orders.cnum = Customers.cnum
     AND Orders.snum = Salespeople.snum;
| ========================== |
| onum  cname     cnum  snum |
| -------------------------- |
| 3001  Cisneros  2008  1007 |
| 3002  Pereira   2007  1004 |
| 3006  Cisneros  2008  1007 |
| 3009  Giovanni  2002  1003 |
| 3007  Grass     2004  1002 |
| 3010  Grass     2004  1002 |
==============================
```

Рис. 8.4: Соединение трех таблиц

Хотя команда выглядит достаточно сложно, следуя ее логике, легко убедиться, что в выходных данных перечислены покупатели и продавцы, расположенные в разных городах (они сравниваются по полю snum), и что указанные заказы сделаны именно этими покупателями (подбор заказов устанавливается в соответствие с полями cnum и snum таблицы Orders).

РЕЗЮМЕ

Теперь вы можете не ограничиваться рассмотрением лишь одной таблицы в некоторый момент времени, а умеете сравнивать любые поля произвольного

числа таблиц и применять полученные результаты для поиска нужной информации. Эта техника настолько полезна для установления связей, что используется и для их конструирования внутри единственной таблицы. В следующей главе будет рассмотрена эффективная процедура соединения двух копий одной таблицы.

Глава 9

ОПЕРАЦИЯ СОЕДИНЕНИЯ, ОПЕРАНДЫ КОТОРОЙ ПРЕДСТАВЛЕНЫ ОДНОЙ ТАБЛИЦЕЙ

Как выполняется операция соединения двух копий одной таблицы

Соединение таблицы с её же копией означает следующее: любую строку таблицы (одну в каждый момент времени) можно комбинировать с её копией и с любой другой строкой этой же таблицы. Каждая такая комбинация оценивается в терминах предиката, как и в случае соединения нескольких различных таблиц. Это позволяет легко конструировать определенные виды связей между различными записями внутри единственной таблицы – например, осуществлять поиск пар строк с общим значением поля.

Соединение таблицы со своей копией можно представить себе следующим образом: реально таблица не копируется, но SQL выполняет команду так, как будто бы делалось именно это. Другими словами, подобный тип соединения не отличается от обычного соединения двух таблиц, за исключением того, что в данном случае они идентичны.

Алиасы

Синтаксис команды соединения таблицы с её же копией тот же, что и для различных таблиц, с единственным исключением. В рассматриваемом случае все имена столбцов повторяются независимо от использования имени таблицы в качестве префикса. Чтобы сослаться на столбцы запроса, нужно иметь два различных имени для одной и той же таблицы. Для этого надо определить временные имена, называемые *переменными области определения, переменными корреляции* или просто *алиасами*. Они определяются в предложении запроса FROM. Для этого указывается имя таблицы, ставится пробел, а затем указывается имя алиаса для данной таблицы.

Приведем пример поиска всех пар продавцов, имеющих одинаковый рейтинг (выходные данные представлены на рис. 9.l):

```
SELECT first.cname, second.cname, first.rating
FROM Customers first, Customers second
WHERE first.rating = second.rating;
```

```
========== SQL Execution Log ==========
| Giovanni  Giovanni  200 |
| Giovanni  Liu       200 |
| Liu       Giovanni  200 |
```

```
| Liu       Liu       200 |
| Grass     Grass     300 |
| Grass     Cisneros  300 |
| Clemens   Hoffman   100 |
| Clemens   Clemens   100 |
| Clemens   Pereira   100 |
| Cisneros  Grass     300 |
| Cisneros  Cisneros  300 |
| Pereira   Hoffman   100 |
| Pereira   Clemens   100 |
| Pereira   Pereira   100 |
===========================
```

Рис. 9.1: Объединение таблицы с собой

(заметим, что на рис. 9.1 , как и в последующих примерах, видна только часть выходных данных запроса, поскольку реально все они в пределах одного окна не умещаются).

В приведенном примере команды SQL ведет себя так, как будто в операции соединения участвуют две таблицы, называемые "first" (первая) и "second" (вторая). Обе они в действительности являются таблицей Customers, но алиасы позволяют рассматривать её как две независимые таблицы. Алиасы first и second были определены в предложении запроса FROM непосредственно за именем таблицы. Алиасы применяются также в предложении SELECT, несмотря на то, что они не определены вплоть до предложения FROM. Это совершенно оправдано. SQL сначала примет какой-либо из таких алиасов на веру, но затем отвергнет команду, если в предложении FROM запроса алиасы не определены. Время жизни алиаса зависит от времени выполнения команды. После выполнения запроса используемые в нем алиасы теряют свои значения.

Получив две копии таблицы Customers для работы, SQL выполняет операцию JOIN, как для двух разных таблиц: выбирает очередную строку из одного алиаса и соединяет ее с каждой строкой другого алиаса.

Исключение избыточности

Выходные данные включают каждую комбинацию значений дважды, причем во второй раз – в обратном порядке. Это объясняется тем, что значение появляется один раз для каждого алиаса, а предикат является симметричным. Следовательно, значение A в алиасе first выбирается в комбинации со значением B в алиасе second, и значение A в алиасе second – в комбинации со значением B в алиасе first. В данном примере Hoffman был выбран с Clemens, а затем Clemens был выбран с Hoffman. То же самое произошло с Cisneros и Grass, Lie и Giovanni и т.д. Кроме того, каждая запись присоединяется сама к себе в выходных данных, например, Lie и Lie.

Простой способ исключить повторения – задать порядок для двух значений так, чтобы одно значение было меньше, чем другое, или предшествовало в алфавитном порядке. Это делает предикат ассиметричным, и одни и те же значения не извлекаются снова в обратном порядке, например:

```
SELECT first.cname, second.cname, first.rating
FROM Customers first, Customers second
WHERE first.rating = second.rating
   AND first.cname < second.cname;
```

Выходные данные для запроса представлены на рис. 9.2.

```
============== SQL Execution Log ==============
  SELECT first.cname, second.cname, first.rating
  FROM Customers first, Customers second
  WHERE first.rating = second.rating
      AND first.cname < second.cname
| ======================== |
| cname     cname  rating |
| ------------------------ |
| Hoffman   Pereira   100 |
| Giovanni  Liu       200 |
| Clemens   Hoffman   100 |
| Gisneros  Grass     300 |
============================
```

Рис. 9.2: Исключение избыточных выходных данных при операции соединения с собственной копией

Hoffman предшествует Periera в алфавитном порядке, эта комбинация удовлетворяет обоим условиям предиката и появляется в составе выходных данных. Когда та же самая комбинация появляется в обратном порядке (т.е. когда Periera из таблицы с алиасом first приписывается Hoffman из таблицы с алиасом second), второе условие не выполняется. С другой стороны, Hoffman не выбирается сам по себе, как имеющий тот же рейтинг, потому что его имя не предшествует ему же самому в алфавитном порядке. Если нужно включить в запрос соединение строки с ее копией, достаточно использовать <= вместо <.

Выявление ошибок

Рассмотренное свойство SQL можно использовать для выявления ошибок определенного рода. Если посмотреть на таблицу Orders, станет ясно, что поля cnum и snum используются для определения связи. Поскольку каждому покупателю (customer) может быть назначен один и только один продавец (salesperson), в любой момент времени определенному номеру покупателя для строки из таблицы Orders соответствует строка с таким же номером продавца. Следующая команда позволяет выявить любые несоответствия такого плана:

```
SELECT first.onum, first.cnum, first.snum, second.onum, second.cnum,
   second.snum
FROM Orders first, Orders second
WHERE first.cnum = second.cnum
   AND first.snum <> second.snum;
```

Команда выглядит сложной, но её логика весьма прозрачна. Она берёт первую строку таблицы Orders и запоминает её под именем алиаса first, затем проверяет её в комбинации с каждой строкой таблицы Orders под именем алиаса second. Если комбинация строк удовлетворяет предикату, она

включается в состав выходных данных. В нашем случае просматривается строка, в которой поле cnum равно 2008, а поле snum равно 1007; затем осуществляется выбор каждой строки, в поле cnum которой содержится такое же значение. Если обнаруживается, что в поле snum любой из этих строк содержится другое (отличное от 1007) значение, то предикат принимает значение "истина", и в состав выходных данных включаются те поля из текущей комбинации строк, имена которых указаны в предложении SELECT. Если все значения поля snum для данного значения cnum в этой таблице одинаковы, приведенная выше команда не генерирует выходных данных.

Ещё про алиасы

Хотя соединение таблиц со своими копиями – это первый встретившийся случай, когда потребовалось понятие алиаса, его употребление не лимитировано использованием для разграничения различных копий одной и той же таблицы. Алиасы можно применять при создании альтернативных имен таблиц в команде SELECT. Например, если таблицы имеют очень длинные и сложные имена, то можно определить простые, состоящие из одной буквы алиасы, например, A или B, и использовать их вместо имен таблиц в предложении SELECT и в предикате. Их можно также применять со связанными подзапросами.

Некоторые более сложные операции соединения

В запросе можно использовать любое количество алиасов для единственной таблицы, хотя применение более двух в одном предложении SELECT нетипично. Предположим, продавцам (salespeople) еще не назначили покупателей (customers). Политика компании состоит в том, чтобы назначить всем продавцам по три покупателя, каждому из которых приписывается одно из трех возможных значений рейтинга. Необходимо решить, как осуществить такое распределение, и использовать следующие запросы для просмотра всех возможных комбинаций назначаемых покупателей (выходные данные представлены на рис. 9.3):

```
SELECT a.cnum, b.cnum, c.cnum
FROM Customers a, Customers b, Customres c
WHERE a.rating = 100
   AND b.rating = 200
   AND c.rating = 300;
```

```
========= SQL Execution Log =======
| cnum  cnum  cnum |
| ---------------- |
| 2001  2002  2004 |
| 2001  2002  2008 |
| 2001  2003  2004 |
| 2001  2003  2008 |
| 2006  2002  2004 |
| 2006  2002  2008 |
```

```
| 2006  2003  2004 |
| 2006  2003  2008 |
| 2007  2002  2004 |
| 2007  2002  2008 |
| 2007  2003  2004 |
| 2007  2003  2008 |
====================
```

Рис. 9.3: Комбинирование покупателей с различными значениями рейтинга

Этот запрос находит все возможные комбинации покупателей (customers) с тремя значениями рейтинга таким образом, что в первом столбце расположены покупатели с рейтингом 100, во втором столбце – покупатели с рейтингом 200, в третьем столбце – покупатели с рейтингом 300. Они повторены во всех возможных комбинациях. Это своего рода группирование данных, которое нельзя выполнить средствами GROUP BY или ORDER BY, поскольку они сравнивают значения только из одного столбца.

Каждый алиас или таблицы, имена которых упомянуты в предложении FROM запроса SELECT, использовать необязательно. Иногда алиас или таблица запрашиваются таким образом, что на них ссылаются предикаты запроса. Например, следующий запрос находит всех покупателей (customers), расположенных в городах, где действует продавец (salesperson) Serres (snum 1002) (выходные данные представлены на рис. 9.4):

```
SELECT b.cnum, b.cname
FROM Customers a, Customers b
WHERE a.snum = 1002
  AND b.city = a.city;
```

```
=============== SQL Execution Log ============
  SELECT b.cnum, b.cname
  FROM Customers a, Customers b
  WHERE a.snum = 1002
  AND b.city = a.city;
| ===============|
| cnum  cname    |
| ------ --------|
| 2003  Liu      |
| 2008  Cisneros |
| 2004  Grass    |
==================
```

Рис. 9.4: Поиск покупателей, расположенных в тех городах, где действует продавец Serres

Алиас *a* сделает предикат ложным, за исключением случаев, когда значение столбца snum равно 1002. Так алиас исключает всех покупателей, кроме покупателей продавца Serres. Алиас *b* принимает значение "истина" для всех строк с тем же значением города (city), что и текущее значение города (city) в *a*; в процессе выполнения запроса строка с алиасом *b* делает предикат истинным всякий раз, когда в поле city этой строки представлено то же значение, что и в поле city строки с алиасом *a*. Поиск строк алиаса *b* выполняется исключительно для сравнения значений с алиасом a, из строк с

алиасом *b* никакого реального выбора данных не выполняется. Покупатели (customers) продавца Serres расположены в одном и том же городе, значит выбор их из алиаса *a* не является необходимым. Итак, алиас *a* локализует строки покупателей (customers) Serres, Liu и Grass. Алиас *b* находит всех покупателей (customers), расположенных в одном из городов (San Jose и Berlin соответственно), включая, конечно, самих Liu и Grass.

Можно конструировать соединения (joins), которые содержат различные таблицы и алиасы единственной таблицы. Следующий запрос соединяет таблицу Customers с ее копией для нахождения всех пар покупателей, обслуживаемых одним и тем же продавцом. В любой момент времени он соединяет покупателя (customer) с таблицей Salespeople для того, чтобы определить имя продавца (salesperson) (выходные данные для запроса представлены на рис. 9.5):

```
SELECT sname, Salespeople.snum, first.cname, second.cname
FROM Customers first, Customers second, Salespeople
WHERE first.snum = second.snum
   AND Salespeople.snum = first.snum
   AND first.cnum < second.cnum;
```

```
=============== SQL Execution Log ==================
  SELECT sname, Salespeople.snum, first.cname, second.cname
  FROM Customers first, Customers second, Salespeople
  WHERE first.snum = second.snum
     AND Salespeople.snum = first.snum
     AND first.cnum < second.cnum;
| ================================|
| sname   snum  cname    cname    |
| ------------------------------ |
| Serres  1002  Liu      Grass    |
| Peel    1001  Hoffman  Clemens  |
==================================
```

Рис. 9.5: Соединение таблицы с ее копией и с другой таблицей

РЕЗЮМЕ

Вы узнали, что такое операции соединения (Joins), как их применять для конструирования связей внутри одной таблицы, различных таблиц или двух одинаковых таблиц, где эти возможности могут оказаться полезными. Теперь вам знакомы термины – диапазон переменных, переменные связи, алиасы (в различных программных продуктах и у разных авторов изложения материала по SQL терминология варьируется, поэтому здесь объяснены все три термина). Вы больше знаете о том, как в действительности работают запросы.

Следующий шаг после комбинации в запросе множества таблиц или множества копий единственной таблицы состоит в таком объединении множества запросов, при котором один запрос генерирует выходные данные, управляющие работой другого запроса. Подробнее об этой эффективной операции SQL мы поговорим в 10-й и последующих главах.

Глава 10

ВЛОЖЕНИЕ ЗАПРОСОВ

В конце прошлой главы мы отметили, что одни запросы могут управлять другими. В большинстве случаев это можно сделать, размещая один запрос внутри предиката, помещенного в другом, и используя выходные данные вложенного запроса для определения истинности или ложности предиката. Из этой главы вы узнаете, какого рода операторы могут использовать подзапросы, как их применять с DISTINCT, агрегатными функциями и выражениями вывода; как использовать подзапросы с предложением HAVING и получать указатели правильного способа применения подзапросов.

Как выполняются подзапросы?

SQL позволяет вкладывать запросы друг в друга. Обычно внутренний запрос генерирует значения, которые тестируются на предмет истинности предиката. Предположим, известно имя, но мы не знаем значения поля snum для продавца Motica. Необходимо извлечь все её заказы из таблицы Orders. Перед вами один из способов решения этой проблемы.

```
SELECT *
FROM Orders
WHERE snum =
   (SELECT snum
    FROM Salespeople
    WHERE sname = 'Motika');
```

Выходные данные представлены на рис. 10.1:

```
=============== SQL Execution Log ==============
  SELECT *
  FROM Orders
  WHERE snum =
    (SELECT snum
     FROM Salespeople
     WHERE sname = 'Motika');
|=======================================|
| onum  amt       odate      cnum  snum |
| ------------------------------------- |
| 3002  1900.10  10/03/1990  2007  1004 |
=========================================
```

Рис. 10.1: Использование подзапроса

Чтобы оценить внешний (основной) запрос, SQL прежде всего должен оценить внутренний запрос (или подзапрос) в предложении WHERE. Эта оценка осуществляется так, как если бы запрос был единственным: просматриваются все строки таблицы Salespeople и выбираются все строки, для которых значение поля sname равно Motika, для таких строк выбираются значения поля snum.

В результате выбранной оказывается единственная строка с snum = 1004. Однако вместо простого вывода этого значения SQL подставляет его в предикат основного запроса вместо самого подзапроса, теперь предикат читается следующим образом:

```
WHERE snum = 1004
```

Затем основной запрос выполняется как обычный, и его результат точно такой же, как на рис. 10.1.

Подзапрос должен выбирать один и только один столбец, а тип данных этого столбца должен соответствовать типу значения, указанному в предикате. Часто выбранное поле и это значение имеют одно и то же имя (в данном случае, snum).

Если бы было известно значение персонального номера продавца (salesperson number) Motika, то можно было бы указать:

```
WHERE snum = 1004
```

и тем самым освободиться от подзапроса, однако использование подзапроса делает процедуру более гибкой. Вариант с подзапросом сработает и в случае изменения персонального номера продавца Motika. Простая замена имени продавца в подзапросе позволяет использовать его во множестве вариантов.

Значения, получаемые в процессе выполнения подзапросов

Весьма полезно то, что подзапрос в данном случае возвращает одно и только одно значение. Если вместо WHERE sname = 'Motika' подставить WHERE city = 'London', то в результате выполнения подзапроса получается несколько значений. Это делает невозможной оценку предиката основного запроса на предмет истинности или ложности, что приводит к оценке запроса как ошибочного.

При использовании подзапросов, основанных на операторах отношения (равенства или неравенства), нужно быть уверенным, что выходными данными подзапроса является только одна строка. Если применяется подзапрос, не генерирующий никаких значений, то это не является ошибкой, однако в настоящем случае и основной запрос не даст никаких выходных данных. Подзапросы, не генерирующие никаких выходных данных (или NULL-выход), приводят к тому, что предикат оценивается не как истинный или ложный, а как имеющий значение "неизвестно" (unknown). Предикат со значением "неизвестно" работает как и предикат со значением "ложь": основной запрос не выбирает ни одной строки (информацию по поводу unknown-предиката см. в главе 5). Попытку использовать нечто вроде

```
SELECT *
FROM Orders
WHERE snum =
  (SELECT snum
   FROM Salespeople
   WHERE city = 'Barcelona');
```

нельзя признать удачной. Если в городе Barcelona есть только один продавец (salesperson) Mr.Rifkin, то подзапрос выберет единственное значение snum, и, следовательно, будет воспринят. Однако это справедливо только для текущих данных. В результате изменения состава данных таблицы Salespeople вполне реальной может стать ситуация, когда в городе (city) Barcelona появятся два продавца, тогда в результате выполнения подзапроса получим два значения, и, следовательно, запрос будет признан ошибочным.

DISTINCT с подзапросами

В некоторых случаях можно использовать DISTINCT для гарантии получения единственного значения в результате выполнения подзапроса. Предположим, нужно найти все заказы (orders), с которыми работает продавец, обслуживающий покупателя Hoffman (cnum = 2001). Вот один из вариантов решения этой задачи (выходные данные представлены на рис. 10.2):

```
SELECT *
FROM Orders
WHERE snum =
   (SELECT DISTINCT snum
   FROM Orders
   WHERE cnum = 2001);
```

Подзапрос выясняет, что значение поля snum для продавца, обслуживающего Hoffman, равно 1001; следовательно, основной запрос извлекает из таблицы Orders всех покупателей с тем же значением поля snum. Поскольку каждый покупатель обслуживается только одним продавцом, каждая строка таблицы Orders с данным значением cnum имеет то же значение поля snum. Однако, поскольку может быть любое количество таких строк, подзапрос может дать в результате множество значений snum (возможно, одинаковых) для данного cnum. Если бы подзапрос возвращал более одного значения, получилась бы ошибка в данных. Аргумент DISTINCT предотвращает такую ситуацию.

```
=============== SQL Execution Log ==============
  SELECT *
  FROM Orders
  WHERE snum =
     (SELECT DISTINCT snum
     FROM Orders
     Where cnum = 2001);
| ======================================== |
| onum     amt     odate       cnum  snum |
| ---------------------------------------- |
| 3003     767.19  10/03/1990  2001  1001 |
| 3008    4723.00  10/05/1990  2006  1001 |
| 3011    9891.88  10/06/1990  2006  1001 |
==========================================
```

Рис. 10.2. Использование DISTINCT с целью получить единственное значение в результате выполнения подзапроса

Альтернативный способ действия – сослаться в подзапросе на таблицу Customers, а не на таблицу Orders. Поскольку cnum – первичный ключ таблицы Customer, есть гарантия, что в результате выполнения запроса будет получено единственное значение. Однако, если пользователь имеет доступ к таблице Orders, а не к таблице Customers, остается вариант, рассмотренный ранее.

Надо помнить, что приведенный в предыдущем примере приём приемлем только тогда, когда есть уверенность, что в двух различных полях содержатся одни и те же значения. Такая ситуация является скорее исключением, а не правилом для реляционных баз данных.

Предикаты с подзапросами являются неперемещаемыми

Предикаты, включающие подзапросы, используют форму *<скалярное выражение> <оператор> <подзапрос>*, а не *<подзапрос> <оператор> <скалярное выражение>* или *<подзапрос> <оператор> <подзапрос>*. Предыдущий пример нельзя записать в таком виде:

```
SELECT  *
FROM Orders
WHERE (SELECT DISTINCT snum
   FROM Orders
   WHERE cnum = 2001)
= snum;
```

В соответствии с соглашениями ANSI, эта запись является ошибочной, хотя некоторые программы ее понимают. Согласно ограничению ANSI запрещен также вариант команды, в котором оба значения, участвующие в сравнении, получаются в результате выполнения подзапросов.

Использование агрегатных функций в подзапросах

Одним из видов функций, которые автоматически выдают в результате единственное значение для любого количества строк, конечно, являются агрегатные функции. Любой запрос, использующий единственную агрегатную функцию без предложения GROUP BY, дает в результате единственное значение для использования его в основном предикате. Например, нужно узнать все заказы, стоимость которых превышает среднюю стоимость заказов за 4 октября 1990 г. (выходные данные представлены на рис. 10.3):

```
SELECT *
FROM Orders
WHERE amt >
   (SELECT AVG (amt)
    FROM Orders
    WHERE odate = 10/04/1990);
```

```
=============== SQL Execution Log ==============
  SELECT *
  FROM Orders
  WHERE amt >
```

```
    (SELECT AVG (amt)
     FROM Orders
     WHERE odate = 01/04/1990);
| ================================== |
| onum    amt      odate     cnum  snum |
| ---------------------------------- |
| 3002  1900.10  10/03/1990  2007  1004 |
| 3005  2345.45  10/03/1990  2003  1002 |
| 3006  1098.19  10/03/1990  2008  1007 |
| 3009  1713.23  10/04/1990  2002  1003 |
| 3008  4723.00  10/05/1990  2006  1001 |
| 3010  1309.95  10/06/1990  2004  1002 |
| 3011  9891.88  10/06/1990  2006  1001 |
========================================
```

Рис. 10.3: Выбор всех сумм со значением выше средней на 10.04.1990

Средняя стоимость заказов за 4 октября 1990 г. составляет 1788.98 (1713.23 + 75.75), деленное на 2, что равно 894.49. Строки, имеющие в поле amt (amount) значение, превышающее 894.49, выбираются в качестве результата запроса с вложенным подзапросом.

Сгруппированные, то есть примененные с предложением GROUP BY агрегатные функции могут дать в результате множество значений. Поэтому их нельзя применять в подзапросах. Такие команды отвергаются в принципе, несмотря на то, что применение GROUP BY и HAVING в некоторых случаях дает единственную группу в качестве выходных данных подзапроса. Для исключения ненужных групп следует применить единственную агрегатную функцию с предложением WHERE. Например, следующий запрос, составленный с целью найти средние комиссионные (comm) для продавцов (salespeople}, находящихся в Лондоне,

```
SELECT AVG (comm)
FROM Salespeople
GROUP BY city
HAVlNG city = "London";
```

нельзя использовать в качестве подзапроса! Это вообще не лучший способ формулировки запроса. Вот вариант, который нужен в данном случае:

```
SELECT AVG (comm)
FROM Salespeople
WHERE city = 'London';
```

Применение подзапросов, которые формируют множественные строки с помощью *IN*

Можно формулировать подзапросы, в результате выполнения которых получается любое количество строк, применяя в основном запросе специальный оператор IN (операторы BETWEEN, LIKE, IS NULL в подзапросах применять нельзя). IN определяет множество значений, которые тестируются на совпадение с другими значениями для определения истинности предиката.

Когда IN применяется в подзапросе, SQL просто строит это множество из выходных данных подзапроса. Следовательно, можно использовать IN для выполнения подзапроса, который не работал бы с реляционным оператором, и найти все заявки (Orders) для продавцов (salespeople) из London (выходные данные представлены на рис. 10.4):

```
SELECT *
FROM Orders
WHERE snum IN
  (SELECT snum
   FROM Salespeople
   WHERE city = 'London');
```

```
=============== SQL Execution Log ==============
 SELECT *
 FROM Orders
 WHERE snum IN
   (SELECT snum
    FROM Salespeople
    WHERE city = 'London');
| ==================================== |
| onum    amt       odate    cnum  snum |
| ------------------------------------ |
| 3003   767.19  10/03/1990  2001  1001 |
| 3002  1900.10  10/03/1990  2007  1004 |
| 3006  1098.19  10/03/1990  2008  1007 |
| 3008  4723.00  10/05/1990  2006  1001 |
| 3011  9891.88  10/06/1990  2006  1001 |
========================================
```

Рис. 10.4: Использование подзапроса с IN

В подобной ситуации пользователю легче понять, а компьютеру проще (в конечном счете и быстрее) выполнить подзапрос, чем решать эту же задачу, применяя join:

```
SELECT onum, amt, odate, cnum, Orders.snum
FROM Orders, Salespeople
WHERE Orders.snum = Salespeople.snum
   AND Salespeople.city = 'London';
```

Выходные данные совпадают с результатом выполнения подзапроса, но SQL просматривает все возможные комбинации строк из двух таблиц и проверяет, удовлетворяет ли каждая из них составному предикату. Проще и эффективнее извлечь из таблицы Salespeople значение поля snum для тех строк, где city = 'London', а затем выполнить поиск этих значений в таблице Orders; именно по такой схеме выполняется вариант с вложенным подзапросом. Вложенный запрос дает номера 1001 и 1004. Внешний запрос дает строки таблицы Orders, в которых значение поля snum совпадает с одним из полученных (1001 или 1004).

Эффективность варианта с использованием подзапроса зависит от *выполнения* – от особенностей реализации той программы, с которой идет работа. В любом коммерческом программном продукте есть часть программы,

называемая *оптимизатором*, которая пытается найти самые эффективные способы выполнения запросов. Хороший оптимизатор преобразует версию с join в версию с подзапросом, но простого способа проверки, сделано это или нет, не существует. Поэтому при написании запросов лучше использовать заведомо более эффективный вариант, чем полностью полагаться на возможности оптимизатора.

Можно применять IN и в тех ситуациях, когда есть абсолютная уверенность в получении единственного значения в результате выполнения подзапроса. IN можно также использовать там, где применим реляционный оператор сравнения. В отличие от реляционных операторов IN не приводит к ошибке выполнения команды, когда в результате выполнения подзапроса получается не одно, а несколько значений (выходных данных). В этом есть и плюсы, и минусы. Результаты выполнения подзапроса непосредственно не видны. При абсолютной уверенности в том, что в результате выполнения подзапроса будет получено только одно значение, а в действительности их получается несколько, невозможно объяснить разницу в выходных данных, полученных в результате выполнения основного запроса. Рассмотрим следующую команду, сходную с командой предыдущего примера:

```
SELECT onum, amt, odate
FROM Orders
WHERE snum =
   (SELECT DISTINCT snum
    FROM Orders
    WHERE cnum = 2001);
```

Можно отказаться от DISTINCT, воспользовавшись IN вместо равенства. В результате получается:

```
SELECT onum, amt, odate
FROM Orders
WHERE snum IN
   (SELECT snum
    FROM Orders
    WHERE cnum = 2001);
```

Если допущена ошибка и один из заказов (orders) был адресован нескольким продавцам (salesperson), версия с IN выдаст все заказы для обоих продавцов. Ошибку обнаружить невозможно, и отчет или решения, принятые на основе полученных данных, были бы неверными. С другой стороны, вариант с равенством, будет просто признан ошибочным, в результате чего возникнет проблема. Затем можно выполнить подзапрос сам по себе и внимательно рассмотреть его выходные данные.

Если есть уверенность в получении единственного значения в результате выполнения подзапроса, следует использовать равенство. IN подходит для тех случаев, когда запрос может генерировать одно или несколько значений, независимо от того, что именно ожидается. Предположим, нужно знать комиссионные всех продавцов (salespeople), обслуживающих покупателей (customers) в Лондоне (London):

```
SELECT comm
FROM Salespeople
WHERE snum IN
   (SELECT snum
    FROM Customers
    WHERE city = "London");
```

Выходные данные для запроса, представленные на рис. 10.5, показывают комиссионные продавца PEEL (snum = 1001), обслуживающего обоих лондонских покупателей. Однако, такой результат получается только на основе текущих данных. Нет (очевидной) причины, в силу которой невозможно, чтобы кого-то из лондонских покупателей обслуживал другой продавец. Следовательно, для данного запроса наиболее логичным вариантом является использование IN.

```
=============== SQL Execution Log ==============
  SELECT comm
  FROM Salespeople
  WHERE snum IN
    (SELECT snum
     FROM Customers
     WHERE city = 'London');
| ===== |
|  comm |
| ----- |
|  0.12 |
=========
```

Рис. 10.5: Использование IN в подзапросе, результатом которого является единственное значение

Отказ от использования префиксов таблиц в подзапросах. Префикс имени таблицы для поля city в предыдущем примере не является необходимым, несмотря на то, что поле city есть в таблице Salespeople и в таблице Customers. SQL всегда сначала пытается найти поля в таблице (таблицах), заданной (заданных) в предложении FROM текущего запроса (подзапроса). Если поле с указанным именем здесь не найдено, то анализируется внешний запрос. В рассмотренном примере предполагается, что "city" в предложении WHERE относится к столбцу city таблицы Customers (Customers.city). Поскольку имя таблицы Customers указано в предложении FROM текущего запроса, предположение оказывается верным. От такого предположения можно избавиться, указывая имя таблицы или алиас в качестве префикса; речь об этом пойдет позднее, при обсуждении связанных подзапросов. Если есть хоть какой-то шанс допустить ошибку, то лучше всего использовать префиксы.

Подзапросы используют единственный столбец. Общая черта всех подзапросов, рассмотренных в этой главе, состоит в том, что они выбирают единственный столбец. Это существенно, так как выходные данные вложенного SELECT-предложения сравниваются с единственным значением. Из этого следует, что вариант SELECT * нельзя использовать в подзапросе.

Исключением из этого правила являются подзапросы с оператором EXISTS, который рассматривается в главе 12.

Использование выражений в подзапросах. В предложении подзапроса SELECT можно использовать выражения, основанные на столбцах, а не сами столбцы. Это можно сделать, применяя операторы отношения или IN. Например, следующий запрос использует оператор отношения = (выходные данные для этого запроса представлены на рис. 10.6)

```
SELECT *
FROM Customers
WHERE cnum =
   (SELECT snum + 1000
    FROM Salespeople
    WHERE sname = 'Serres');
```

```
=============== SQL Execution Log ============
 SELECT *
 FROM Customers
 WHERE cnum =
   (SELECT snum + 1000
    WHERE Salespeople
    WHERE sname = 'Serres'
| ================================ |
| cnum   cname    city rating snum |
| -------------------------------- |
| 2002  Giovanni  Rome  200   1002 |
====================================
```

Рис. 10.6: Использование подзапроса с выражением

Запрос находит всех покупателей, для которых cnum на 1000 превосходит значение поля snum для Serres. В данном случае предполагается, что в столбце snum нет повторяющихся значений (этого можно добиться, применяя либо UNIQUE INDEX, обсуждаемый в главе 17, либо ограничение UNIQUE, обсуждаемое в главе 18); в противном случае результатом выполнения подзапроса может оказаться множество значений. Запрос, рассмотренный в данном примере, вероятно, не очень полезен, если только не предполагается, что поля snum и cnum имеют значение, отличное от того, что просто служат первичными ключами. Но всё это не проясняет сути дела.

Подзапросы с *HAVING*

Подзапросы можно применять также внутри предложения HAVING. В самих таких подзапросах можно использовать их собственные агрегатные функции, если они не дают множества значений, а также GROUP BY или HAVING. Например (выходные данные представлены на рис. 10.7):

```
SELECT rating, COUNT (DISTINCT cnum)
FROM Customers
GROUP BY rating
HAVING rating >
   (SELECT AVG (rating)
```

```
   FROM Customers
   WHERE city = 'San Jose');
```

```
=============== SQL Execution Log ============
  SELECT rating, count (DISTINCT cnum)
  FROM Customers
  GROUP BY rating
  HAVING rating >
    (SELECT AVG (rating)
     FROM Customers
     WHERE city = 'San Jose');
|========= |
| rating   |
| -------- |
| 300    2 |
============
```

Рис. 10.7: Поиск покупателей (customers) с рейтингом (raiting), превышающим среднее значение для San Jose

Эта команда подсчитывает количество покупателей с рейтингом, превышающим среднее значение для покупателей города San Jose. Если бы были другие рейтинги, отличные от указанного значения 300, то каждое отдельное значение рейтинга выводилось бы вместе с указанием числа покупателей, имеющих такой рейтинг.

РЕЗЮМЕ

Теперь вы можете применять запросы в иерархическом порядке и знаете, насколько использование выходных данных одного запроса для управления другим облегчает выполнение многих действий, как можно использовать подзапросы с операторами отношения, а также со специальным оператором IN в предложении WHERE или HAVING внешнего запроса.

Далее будет продолжен анализ подзапросов. В следующей главе рассматриваются подзапросы, выполняемые отдельно для каждой строки, определённой во внешнем запросе.

Глава 11

СВЯЗАННЫЕ ПОДЗАПРОСЫ

В данной главе вводятся подзапросы нового типа, которые не рассматривались ранее, – связанные подзапросы. Вы сможете использовать их в предложениях запросов WHERE и HAVING. Мы рассмотрим сходство и различия между связанными запросами и соединениями (join), а также особенности алиасов и префиксов имен таблиц (когда они необходимы и как ими пользоваться).

Как формировать связанные подзапросы

Когда в SQL используются подзапросы, во внутреннем запросе (вложенном запросе) можно ссылаться на таблицу, имя которой указано в предложении FROM внешнего запроса, тем самым формируя связанный подзапрос *(correlated subquery*). В этом случае подзапрос выполняется повторно, по одному разу для каждой строки таблицы из основного запроса. Связанные запросы относятся, из-за сложности их оценок, к числу наиболее неясных понятий SQL. Однако, начав работать с ними, вы поймете, что они являются весьма мощным средством, так как могут выполнять очень сложные функции при достаточно компактных командах.

Вот, например, один из способов отыскать всех покупателей, сделавших заказы 3 октября 1990 года (выходные данные представлены на рис. 11.1):

```
SELECT *
FROM Customers outer
WHERE 10/03/1990 IN
    (SELECT odate
     FROM Orders inner
     WHERE outer.cnum = inner.cnum);
```

```
=============== SQL Execution Log ============
 SELECT *
 FROM Customers outer
 WHERE 10/03/1990 IN
   (SELECT odate
    FROM Orders inner
    WHERE outer.cnum = inner.cnum);
| ==================================== |
| cnum  cname     city      rating snum |
| ------------------------------------ |
| 2001  Hoffman   London    100    1001 |
| 2003  Liu       San Jose  200    1002 |
| 2008  Cisneros  San Jose  300    1007 |
| 2007  Pereira   Rome      100    1004 |
========================================
```

Рис. 11.1: Использование соотнесенного подзапроса

Как работают связанные подзапросы

В данном примере "inner" и "outer" являются алиасами. Эти имена здесь выбраны для большей ясности (inner – внутренний, outer – внешний); они относятся к значениям внутреннего и внешнего запросов соответственно. Поскольку значения в поле cnum внешнего запроса изменяются, внутренний запрос должен выполняться отдельно для каждой строки внешнего запроса. Строка внешнего запроса, для которой выполняется внутренний запрос, называется текущей строкой-кандидатом *(current candidate row*). Процедура выполнения связанного подзапроса состоит в следующем:

1. Выбрать строку из таблицы, имя которой указано во внешнем запросе. Это текущая строка-кандидат.
2. Сохранить значения этой строки в алиасе, имя которого указано в предложении FROM внешнего запроса.
3. Выполнить подзапрос. Всякий раз, когда алиас, заданный для внешнего запроса, найден (в нашем случае "outer"), его значение применяется к текущей строке-кандидату. Использование в подзапросе значения из строки-кандидата внешнего запроса называется *внешней ссылкой (outer reference).*
4. Оценить предикат внешнего запроса на основе результатов подзапроса, выполненного на шаге 3. Это позволяет определить, будет ли строка-кандидат включена в состав выходных данных.
5. Повторять процедуру для следующей строки-кандидата таблицы до тех пор, пока не будут проверены все строки таблицы.

В предыдущем примере SQL выполняет следующую процедуру:

1. Извлекается строка Hoffman из таблицы Customers.
2. Эта строка сохраняется как текущая строка-кандидат при алиасе "outer".
3. Затем выполняется подзапрос: просматривается вся таблица Orders с целью найти строки, в которых значение поля cnum совпадает со значением outer.cnum; в данном случае это значение 2001 (значение поля cnum строки Hoffman). Затем извлекается поле odate из каждой строки таблицы Orders, для которой предикат принимает значение true (истина), и строится множество результирующих значений odate.
4. После получения множества всех значений odate, для которых cnum равно 2001, выполняется тестирование предиката основного запроса, с целью выявить, есть ли в этом множестве 3 октября 1990 года. В случае, если такая дата обнаружена (а это так и есть), строка Hoffman выбирается для включения в состав выходных данных основного запроса.
5. Процедура повторяется со строкой Giovanni, которая выбирается в качестве строки-кандидата, а затем с каждой строкой до тех пор, пока не будет проверена вся таблица Customers.

Согласно этим простым инструкциям SQL выполняет весьма сложные действия. Эту задачу можно было бы решить и с помощью операции соединения (join), например, следующим образом (выходные данные для этого запроса представлены на рис. 11.2):

```
SELECT *
FROM Customers first, Orders second
WHERE first.cnum = second.cnum
   AND second.odate = '10/03/1990';
```

```
=============== SQL Execution Log ============
  SELECT *
  FROM Customers first, Orders second
  WHERE first.cnum = second.cnum
    (SELECT COUNT (*)
     FROM Customers
     WHERE snum = main.snum;
| ============= |
| cnum  cname   |
| ------------- |
| 1001  Peel    |
| 1002  Serres  |
=================
```

Рис. 11.2: Использование соединения вместо связанного подзапроса

Здесь Cisneros была выбрана дважды, по одному разу на каждый заказ, который она сделала в указанный день. Этого можно было бы избежать, используя SELECT DISTINCT вместо SELECT. Однако, такое действие не является необходимым при использовании версии с подзапросами. Оператор IN, используемый в версии с подзапросами, не делает разницы между значениями, выбираемыми в подзапросе однажды и повторно. Следовательно, в таком варианте запроса DISTINCT не является необходимым.

Предположим, нужно узнать имена и номера всех продавцов, имеющих более одного покупателя. Для этого можно воспользоваться таким запросом (выходные данные представлены на рис. 11.3):

```
SELECT snum, sname
FROM Salespeople main
WHERE 1 <
   (SELECT COUNT (*)
    FROM Customers
    WHERE snum = main.snum);
```

```
=============== SQL Execution Log ============
  SELECT snum, sname
  FROM Salespeople main
  WHERE 1 <
    (SELECT COUNT (*)
     FROM Customers
     WHERE snum = main.snum);
| ============ |
| snum  sname  |
| ------------ |
| 1001  Peel   |
| 1002  Serres |
================
```

Рис. 11.3: Поиск продавца для множества покупателей

В этом примере предложение FROM в подзапросе не использует алиас. При отсутствии префикса в виде имени таблицы или алиаса SQL сначала предполагает, что извлекаются поля той таблицы, имя которой указано в предложении FROM текущего запроса. Если в этой таблице нет полей с указанным именем (в данном случае – snum), то SQL переходит к просмотру внешних запросов. Поэтому префиксы, содержащие имя таблицы, обычно необходимы в связанных подзапросах для того, чтобы избавится от подобных предположений. Алиасы часто применяются и для того, чтобы можно было ссылаться на одну и ту же таблицу во внутреннем и внешнем запросах без каких-либо недоразумений.

Использование связанных подзапросов для поиска ошибок

Иногда полезно выполнить запросы, предназначенные специально для поиска ошибок. В базе данных всегда может появиться ошибочная информация, и ее бывает трудно обнаружить. Следующий запрос не генерирует выходных данных. Он проверяет таблицу Orders с целью установить, соответствует ли соотношение полей snum и cnum каждой ее строки соотношению этих же полей в таблице Customers, и осуществляет вывод любой строки, для которой такое соответствие не обнаружено. Другими словами, он позволяет контролировать корректность связей продавцов с обслуживаемыми ими покупателями (предполагается, что cnum, как первичный ключ таблицы Customers, не имеет повторяющихся значений в этой таблице).

```
SELECT *
FROM Orders main
WHERE NOT snum =
   (SELECT snum
    FROM Customers
    WHERE cnum = main. cnum);
```

Можно выявить ряд ошибок такого рода, используя механизм ссылочной целостности. Однако такой механизм не всегда доступен и не во всех случаях приемлем, поэтому и полезны подзапросы, ориентированные на поиск ошибок.

Связывание таблицы со своей копией

Можно использовать подзапросы, основанные на той же таблице, что и основной запрос. Это позволяет извлекать определенные сложные виды производной информации. Например, можно найти все заказы, величина которых превышает среднюю величину заказа для данного покупателя (выходные данные представлены на рис. 11.4):

```
SELECT *
FROM Orders outer
WHERE amt >
   (SELECT AVG (amt)
    FROM Orders inner
```

```
    WHERE inner.cnum = outer.cnum);
============== SQL Execution Log ==============
  SELECT *
  FROM Orders outer
  WHERE amt >
    (SELECT AVG (amt)
     FROM Orders inner
     WHERE inner.cnum = outer.cnum)
| ================================== |
| onum   amt       odate     cnum  snum |
| ---------------------------------- |
| 3006  1098.19  10/03/1990  2008  1007 |
| 3010  1309.00  10/06/1990  2004  1002 |
| 3011  9891.88  10/06/1990  2006  1001 |
========================================
```

Рис. 11.4: Связывание таблицы с ее копией

В маленькой таблице, рассматриваемой в качестве примера, где большинство покупателей имеют только один заказ, большинство значений совпадает со средним и, следовательно, не извлекается. Можно ввести команду по-другому (выходные данные представлены на рис. 11.5):

```
SELECT *
FROM Orders outer
WHERE amt >=
   (SELECT AVG (amt)
    FROM Orders inner
    WHERE inner.cnum = outer.cnum);
============== SQL Execution Log ==============
  SELECT *
  FROM Orders outer
  WHERE amt >=
    (SELECT AVG (amt)
     FROM Orders inner
     WHERE inner.cnum = outer.cnum);
| ================================== |
| onum    amt      odate     cnum  snum |
| ---------------------------------- |
| 3003   767.19  10/03/1990  2001  1001 |
| 3002  1900.10  10/03/1990  2007  1004 |
| 3005  5160.45  10/03/1990  2003  1002 |
| 3006  1098.19  10/03/1990  2008  1007 |
| 3009  1713.23  10/04/1990  2002  1003 |
| 3010  1309.95  10/06/1990  2004  1002 |
| 3011  9891.88  10/06/1990  2006  1001 |
========================================
```

Рис. 11.5: Выбор заказов, величина которых превышает среднее значение для их покупателей или совпадает с ним

Разница заключается в том, что здесь оператор отношения главного предиката включает значения, равные средней величине заказов (это обычно означает, что данные заказы являются единственными у данных покупателей).

Связанные подзапросы в *HAVING*

В предложении HAVING могут использоваться и связанные подзапросы. В этом случае необходимо ограничить внешние ссылки теми элементами,

которые могут непосредственно применяться в самом предложении HAVING. Как стало известно из главы 6, предложение HAVING может использовать только функции агрегирования из предложения SELECT или поля из предложения GROUP BY. Это и есть единственные внешние ссылки, которые можно делать, потому что предикат предложения HAVING оценивается для каждой группы из внешнего запроса, а не для каждой строки. Следовательно, подзапрос будет выполняться один раз для каждой группы выходных данных внешнего запроса, а не для каждой отдельной строки.

Предположим, необходимо суммировать значения поля amounts (amt) таблицы Orders, сгруппировав их по датам и исключив те дни, когда сумма не превышает максимальное значение, по крайней мере на 2000.00:

```
SELECT odate, SUM (amt)
FROM Orders a
GROUP BY odate
HAVING SUM (amt) >
   (SELECT 2000.00 + MAX (amt)
    FROM Orders b
    WHERE a.odate = b.odate);
```

Подзапрос вычисляет максимальное (MAX) значение для всех строк с одной и той же датой, совпадающей с датой, для которой сформирована очередная группа основного запроса. Это должно быть сделано, как показано в данном примере, с помощью предложения WHERE. В самом подзапросе не должно быть предложений GROUP BY и HAVING.

Связанные подзапросы и соединения

Связанные подзапросы имеют близкое сходство с соединениями, так как оба варианта включают сравнение каждой строки таблицы с каждой строкой другой (или алиасом той же самой) таблицы. Сходство заключается и в том, что многие операции, которые можно выполнить с помощью одного варианта, выполнимы и с помощью другого.

Различие в их применении заключается в ранее упомянутой необходимости использовать иногда DISTINCT в команде соединения (join), тогда как этого не требуется в подзапросах. Есть также ряд вещей, которые можно сделать только с помощью одного из этих вариантов. Подзапросы, например, могут использовать агрегатные функции в предикате, позволяя выполнить операции, подобные рассмотренной в предыдущем примере, когда извлекались заказы, величина которых превышала среднее значение для данного покупателя. С другой стороны, соединения позволяют получать строки из двух таблиц, участвующих в сравнении, тогда как выходные данные подзапросов могут использоваться только в предикатах внешних запросов. Согласно простому правилу, лучше, вероятно, использовать ту форму запроса, которая кажется интуитивно более понятной, однако предпочтительно знать оба варианта на случай, если один из них окажется неприемлемым.

РЕЗЮМЕ

В главе было рассмотрено одно из сложных понятий SQL – связанные подзапросы. Мы выяснили, как они соотносятся с операцией соединения, как их можно использовать с функциями агрегирования и с предложением HAVING. Вы ознакомились со всеми типами подзапросов.

Следующий шаг – введение некоторых специальных операторов, использующих, как и оператор IN, подзапросы в качестве аргументов, но в отличие от IN применяемых только с подзапросами. Первый из них, EXISTS, рассмотрен в главе 12.

Глава 12

ИСПОЛЬЗОВАНИЕ ОПЕРАТОРА **EXISTS**

В этой главе речь пойдет о специальных операторах, всегда использующих подзапросы в качестве аргументов, в частности, об операторе EXISTS.

Этот оператор применяется для образования предиката, фиксирующего, будет ли подзапрос генерировать выходные данные. Вы научитесь использовать оператор EXISTS в обычных и связанных подзапросах. Будут рассмотрены специальные случаи его употребления для агрегатов, NULL-значений и булевых значений. Вы сможете усовершенствовать навыки работы с подзапросами, рассматривая более сложные способы их применения.

Как работает оператор *EXISTS?*

EXISTS – оператор, генерирующий значение "истина" или "ложь", другими словами, булево выражение. Это значит, что его можно применять отдельно в предикате или комбинировать с другими булевыми выражениями с помощью операторов AND, OR и NOT. Используя подзапрос в качестве аргумента, этот оператор оценивает его как истинный, если он генерирует выходные данные, а в противном случае как ложный. В отличие от прочих операторов и предикатов, он не может принимать значение unknown. Например, нужно извлечь данные из таблицы Customers в том случае, если один (или более) покупатель из нее находится в San Jose (выходные данные для запроса представлены на рис. 12.l):

```
SELECT cnum, cname, city
FROM Customers
WHERE EXISTS
   (SELECT *
   FROM Customers
   WHERE city = 'San Jose');
```

```
=============== SQL Execution Log ============
  SELECT snum, sname, city
  FROM Customers
```

```
  WHERE EXISTS
    (SELECT *
     FROM Customers
     WHERE city = 'San Jose');
| ========================|
| cnum   cname     city    |
| ------------------------|
| 2001  Hoffman   London   |
| 2002  Giovanni  Rome     |
| 2003  Liu San   Jose     |
| 2004  Grass     Berlin   |
| 2006  Clemens   London   |
| 2008  Cisneros  San Jose |
| 2007  Pereira   Rome     |
============================
```

Рис. 12.1 Использование оператора EXISTS

Внутренний запрос выбрал все данные обо всех покупателях благодаря тому, что нашёлся покупатель из San Jose. Оператор EXISTS внешнего предиката отметил, что подзапрос генерирует выходные данные, и поскольку выражение EXISTS является единственным в этом предикате, он принимает значение "истинно". Здесь подзапрос (не являющийся связанным) выполняется только один раз для всего внешнего запроса и, следовательно, имеет единственное значение для всех случаев. Поскольку EXISTS в данном примере делает предикат истинным или ложным для всех строк сразу, применять его в таком стиле для извлечения специфической информации не стоит.

Выбор столбцов с помощью *EXISTS*

Несущественно, сколько столбцов извлекает EXISTS, поскольку он вообще не применяет полученных значений, а лишь фиксирует наличие выходных данных подзапроса.

Использование *EXISTS* со связанными подзапросами

При применении связанных подзапросов предложение EXISTS, как и другие предикатные операторы, оценивается отдельно для каждой строки таблицы, на которую есть ссылка во внешнем запросе. Это позволяет использовать EXISTS как правильный предикат, генерирующий различные ответы для каждой строки таблицы, на которую есть ссылка в основном запросе. Следовательно, при таком способе применения EXISTS информация из внутреннего запроса сохраняется, если непосредственно не выводится. Например, можно сделать запрос на поиск тех продавцов, которые имеют нескольких покупателей (выходные данные такого запроса представлены на рис. 12.2):

```
SELECT DISTINCT snum
FROM Customers outer
WHERE EXISTS
  (SELECT *
```

```
    FROM Customers inner
    WHERE inner.snum = outer.snum
      AND inner.cnum <> outer.cnum);

=============== SQL Execution Log ============
  SELECT DISTINCT snum
  FROM Customers outer
  WHERE EXISTS
    (SELECT *
     FROM Customers inner
     WHERE inner.snum = outer.snum
        AND inner.cnum <> outer.cnum);
| ===== |
| snum  |
| ----- |
| 1001  |
| 1002  |
=========
```

Рис. 12.2: Использование `EXISTS` со связанными подзапросами

Для каждой строки-кандидата внешнего запроса (представляющей рассматриваемого в настоящий момент покупателя), внутренний запрос находит строки, имеющие соответствующее значение snum (имеет того же продавца), но не значение cnum (соответствует другому покупателю). Если строка, удовлетворяющая подобному критерию, найдена во внутреннем запросе, это означает, что различные покупатели обслуживаются данным продавцом (т.е. продавцом, обслуживающим покупателя, указанного в строке-кандидате внешнего запроса). Следовательно, предикат EXISTS истинен для текущей строки, и номер поля продавца (snum) из таблицы внешнего запроса включается в состав выходных данных. Если бы не был задан DISTINCT, каждый из таких продавцов выбирался бы один раз для каждого покупателя, которого он обслуживает.

Комбинирование *EXISTS* и соединений

Иногда кроме номера требуется получить о каждом продавце больше информации. Это можно сделать соединением таблиц Customers и Salespeople (выходные данные представлены на рис. 12.3):

```
SELECT DISTINCT first.snum, sname, first.city
FROM Salespeople first, Customers second
WHERE EXISTS
    (SELECT *
    FROM Customers third
    WHERE second.snum = third.snum
        AND second.cnum <> third.cnum)
AND first.snum = second.snum;
```

Внутренний запрос тот же, что и в предыдущем примере, изменены только алиасы. Внешний запрос является соединением таблиц Salespeople и Customers.

Новое предложение основного предиката (AND first.snum = second.snum) оценивается на том же уровне, что и предложение EXISTS. Это функциональный предикат самого соединения, сравнивающий две таблицы из внешнего запроса в терминах общего для них поля snum. Поскольку используется булев оператор AND, оба условия, сформулированные в предикатах основного запроса, должны быть истинными для истинности этого предиката. Следовательно, результаты подзапроса действуют в случаях, когда вторая часть запроса истинна и соединение выполняется. Комбинирование соединений и подзапросов указанным способом является весьма эффективным методом обработки данных.

```
=============== SQL Execution Log ============
  SELECT DISTINCT first.snum, sname, first.city
  FROM Salespeople first, Customers second
  WHERE EXISTS
     (SELECT *
      FROM Customers third
      WHERE second.snum = third.snum
        AND second.cnum <> third.cnum)
  AND first.snum = second.snum;
| ======================|
| snum  sname   city    |
| ----------------------|
| 1001  Peel    London  |
| 1002  Serres  San Jose |
=========================
```

Рис. 12.3: Комбинация `EXISTS` с объединением

Использование *NOT EXISTS*

Из предыдущего примера ясно, что EXISTS можно комбинировать с булевыми операторами. С EXISTS легче всего применять – и чаще всего применяется – оператор NOT. Один из способов поиска всех продавцов, имеющих только одного покупателя, – это поставить NOT перед EXISTS в предыдущем примере (см. рисунок 12.2) (выходные данные для запроса представлены на рис. 12.4):

```
SELECT DISTINCT snum
FROM Customers outer
WHERE NOT EXISTS
   (SELECT *
   FROM Customers inner
   WHERE inner.snum = outer.snum
      AND inner.cnum <> outer.cnum);
```

```
=============== SQL Execution Log ============
  SELECT DISTINCT snum
  FROM Salespeople outer
  WHERE NOT EXISTS
    (SELECT *
     FROM Customers inner
```

```
     WHERE inner.snum = outer.snum
        AND inner.cnum <> outer.cnum);
| ========================================== |
| snum |
| -----|
| 1003 |
| 1004 |
| 1007 |
========
```

Рис. 12.4: Использование EXISTS с NOT

EXISTS и агрегаты

EXISTS не может использовать агрегатные функции в своем подзапросе. Это естественно. Если функция агрегирования находит строки для работы с ними, то EXISTS принимает значение "истина", и ему безразлично реальное значение функции; если функция не находит никаких строк, то значение EXISTS – "ложь". Попытка использовать функции агрегирования с EXISTS подобным образом свидетельствует о том, что проблема не была правильно понята.

Подзапрос для предиката EXISTS тоже может иметь в своем составе один или несколько подзапросов любого типа. Эти подзапросы, как и любые другие, входящие в их состав, могут использовать агрегатные функции, если только не существует каких-либо иных причин, по которым это невозможно. Пример, иллюстрирующий такую ситуацию, приведен в следующем разделе.

В любом случае можно получить тот же результат проще – выбрать поле, которое используется в агрегатной функции, вместо применения самой функции. Другими словами, предикат

```
EXISTS (SELECT COUNT (DISTINCT sname) FROM salespeople)
```

эквивалентен предикату

```
EXISTS (SELECT sname FROM Salespeople),
```

причем первый из них тоже допустим. (*На мой взгляд, сомнительно – COUNT может дать 0, а это тоже результат, т.е. запись – В.А.С.*)

Примеры более сложных подзапросов

Возможности применения запросов весьма разнообразны. В одном запросе можно разместить один или несколько запросов, и даже один внутри другого. Перед вами пример, в котором извлекаются строки для всех продавцов, имеющих покупателей, сделавших более одного заказа. Решение этой проблемы представляет интерес с точки зрения демонстрации преимуществ логики SQL. Нужную информацию можно получить путем связывания всех трех рассматриваемых нами в примерах таблиц:

```
SELECT *
FROM Salespeople first
WHERE EXISTS
   (SELECT *
```

```
    FROM Customers second
   WHERE first.snum = second.snum
     AND 1 <
        (SELECT COUNT(*)
        FROM Orders
        WHERE Orders.cnum = second.cnum));
```

Выходные данные представлены на рис. 12.5.

```
=============== SQL Execution Log ============
  SELECT *
  FROM Salespeople first
  WHERE EXISTS
    (SELECT *
     FROM Customers second
     WHERE first.snum = second.snum
       AND 1 <
         (SELECT COUNT (*)
          FROM Orders
          WHERE Orders.cnum = second.cnum));
| ============================ |
| cnum  cname    city      comm |
| ---------------------------- |
| 1001  Peel     London    0.17 |
| 1002  Serres   San Jose  0.13 |
| 100   Rifkin   Barselona 0.15 |
================================
```

Рис. 12.5: Использование EXISTS в сложном подзапросе

Можно рассмотреть оценку данного запроса следующим образом. Взять каждую строку таблицы Salesperson в качестве строки-кандидата (внешний запрос) и выполнить подзапросы. Для каждой строки-кандидата из внешнего запроса взять каждую строку из таблицы Customers (средний запрос). Если текущая строка покупателя (customer) не соответствует текущей строке продавца (т.е. если first.snum <> second.snum), то предикат среднего запроса ложен. Как только в среднем запросе найдется покупатель, который соответствует продавцу во внешнем запросе, нужно перейти к самому внутреннему запросу, чтобы определить, истинен ли предикат среднего запроса. Самый внутренний запрос выполняет подсчёт заказов (orders) для текущего покупателя (из среднего запроса). Если это число превышает 1, то предикат среднего запроса принимает значение "истина", и строка выбирается. Это делает предикат EXISTS внешнего запроса истинным для текущей строки продавца, что означает, что, по крайней мере, один из покупателей данного продавца имеет более одного заказа.

Этот запрос намного сложнее тех, с которыми мы обычно встречаемся в жизни. Основное назначение подобных примеров – показать те дополнительные средства, которые вам могут понадобиться. После работы в таких сложных ситуациях простые запросы, используемые наиболее часто в SQL, покажутся элементарными.

Кроме того, этот запрос связывает три различные таблицы и выдает информацию, которую было бы трудно получить более простым способом. Возможно, на практике такая информация вам будет нужна регулярно, например, если продавец, обеспечивший в течение недели несколько заказов одного покупателя, получает в результате вознаграждение. В этом случае понадобится применение множества команд для изменяющихся данных (для этого удобно использовать *представления – view*).

РЕЗЮМЕ

Несмотря на свою кажущуюся простоту, EXISTS относится к наиболее сложным, но весьма гибким и мощным операторам SQL. В этой главе вы познакомились с многочисленными возможностями этого оператора и существенно расширили свои знания в области логики сложных подзапросов.

Следующий шаг – рассмотрение трех других специальных операторов, использующих подзапросы в качестве аргументов: ANY, ALL, SOME. Далее вы узнаете, что они могут служить альтернативой уже известным операторам, а во многих случаях даже более полезны.

Глава 13

ИСПОЛЬЗОВАНИЕ ОПЕРАТОРОВ *ANY, ALL* И *SOME*

В данной главе вы познакомитесь еще с тремя специальными операторами, ориентированными на подзапросы. (Реально их два, поскольку ANY и SOME совпадают по назначению и использованию.) Этими операторами исчерпываются возможные типы предикатов SQL, используемых в подзапросах. Мы рассмотрим множество способов формулирования одного и того же запроса, применяя различные типы предикатов в подзапросах; вы узнаете преимущества и недостатки каждого подхода.

ANY, ALL и SOME так же, как и EXISTS, используют в качестве аргументов подзапросы; однако, от EXISTS они отличаются тем, что применяются в конъюнкции с операторами отношения. В этом плане они сходны с оператором IN, т.е. берут все значения, полученные в подзапросе и рассматривают их как единое целое. Однако, в отличие от IN, их можно применять только с подзапросами.

Специальный оператор *ANY* или *SOME*

Начнем с операторов ANY или SOME. Независимо от применения, они выполняются абсолютно одинаково и являются взаимозаменяемыми. Различие в терминологии отражает попытку ориентации на интуитивные представления пользователя. Однако, такой подход проблематичен, поскольку интуитивная интерпретация этих операторов может привести к ошибке.

Представляем новый способ найти продавцов с покупателями, находящимися в одних городах (выходные данные для этого запроса представлены на рис. 13.1):

```
SELECT *
FROM Salespeople
WHERE city = ANY
   (SELECT city
    FROM Customers);
```

```
=============== SQL Execution Log ============
  SELECT *
  FROM Salespeople
  WHERE city = ANY
    (SELECT city
     FROM Customers);
| ============================ |
| snum  sname    city     comm |
| ---------------------------- |
| 1001  Peel     London   0.12 |
| 1002  Serres   San Jose 0.13 |
| 1004  Motika   London   0.11 |
================================
```

Рис. 13.1: Использование оператора ANY

Оператор ANY берет все значения поля city в таблице Customers, полученные в подзапросе, и оценивает результат как истину, если какое-либо (ANY) значение совпадает со значением поля city из текущей строки внешнего запроса. Это означает, что подзапрос должен выбирать значения того же типа, которые сравнились в основном предикате. В этом отношении ANY отличается от EXISTS, который просто определяет реально не используемые результаты.

Использование *IN* или *EXISTS* вместо *ANY*

Чтобы сформулировать предыдущий запрос, можно воспользоваться оператором IN:

```
SELECT *
FROM Salespeople
WHERE city IN
  (SELECT city
   FROM Customers);
```

Запрос генерирует выходные данные, представленные на рис. 13.2.

```
=============== SQL Execution Log ============
  SELECT *
  FROM Salespeople
  WHERE city IN (SELECT city
    FROM Customers);
| ============================ |
| snum  sname    city     comm |
| ---------------------------- |
| 1001  Peel     London   0.12 |
```

```
| 1002  Serres  San Jose  0.13 |
| 1004  Motika  London    0.11 |
================================
```

Рис. 13.2: Использование IN в качестве альтернативы к ANY

Оператор ANY может использовать другие операторы отношения кроме равенства и, следовательно, выполняет сравнения, отличные от сравнений в IN. Например, можно найти всех продавцов, имеющих покупателей, имена которых следуют в алфавитном порядке за именем продавца (выходные данные представлены на рис. 13.3):

```
SELECT *
FROM Salespeople
WHERE sname < ANY
   (SELECT cname
    FROM Customers);
```

```
=============== SQL Execution Log ============
  SELECT *
  FROM Salespeople
  WHERE sname < ANY
     (SELECT cname
      FROM Customers);
| ============================== |
| snum  sname    city      comm |
| ------------------------------ |
| 1001  Peel     London    0.12 |
| 1004  Motika   London    0.11 |
| 1003  Axelrod  New York  0.10 |
================================
```

Рис. 13.3: Использование ANY с неравенством

Все строки, которые были выбраны, сохраняются для Serres и Rifkin, поскольку они не имеют покупателей, имена которых следуют за их именами в алфавитном порядке. Это эквивалентно следующему запросу с EXISTS, выходные данные для которого представлены на рис. 13.4:

```
SELECT *
FROM Salespeople outer
WHERE EXISTS
   (SELECT *
    FROM Customers inner
    WHERE outer.sname < inner.cname);
```

```
=============== SQL Execution Log ============
  SELECT *
  FROM Salespeople outer
  WHERE EXISTS
    (SELECT *
     FROM Customers inner
     WHERE outer.sname < inner.cname);
| ============================== |
| snum  sname    city      comm |
| ------------------------------ |
| 1001  Peel     London    0.12 |
| 1004  Motika   London    0.11 |
```

```
| 1003  Axelrod  New York  0.10 |
================================
```

Рис. 13.4 Использование EXISTS как альтернативы оператору ANY

Любой запрос, сформулированный с ANY (или с ALL), можно сформулировать и с EXISTS, хотя обратное утверждение неверно. Строго говоря, версии с EXISTS не совсем идентичны версиям с ANY или ALL. Различие заключается в обработке NULL-значений (эта проблема будет рассмотрена позже в этой главе). Можно обойтись без ANY и ALL, если умело применять EXISTS (и IS NULL). Многие пользователи считают, что использовать ANY и ALL проще, чем EXISTS, который требует связанных подзапросов. В зависимости от практической реализации, ANY и ALL могут, по крайней мере теоретически, быть более эффективными по сравнению с EXISTS. Подзапрос с ANY или ALL можно выполнять один раз для каждой строки основного запроса и предоставлять выходные данные для определения предиката. С другой стороны, EXISTS использует связанный подзапрос, который требует повторного выполнения всего подзапроса для каждой строки основного запроса. SQL пытается найти наиболее эффективный способ выполнения любой команды, поэтому он может пытаться преобразовать менее действенную формулировку запроса в более эффективную (но нельзя рассчитывать на то, что при этом будет найдена наиболее эффективная формулировка).

Основная причина предложения формулировки с EXISTS, как альтернативы ANY и ALL, состоит в противоречии ANY и ALL интуиции, связанной со спецификой применения этих терминов в английском языке. Если вы научитесь применять разные способы формулировки данного запроса, у вас появится возможность разработать множество процедур для сложных или неясных случаев.

Неоднозначности при использовании *ANY*

ANY не является полностью интуитивно очевидным. Если запрос формулируется для выбора покупателей, имеющих рейтинг, превышающий рейтинг любого покупателя в Rome, можно получить выходные данные, которые отличаются от ожидаемых (как показано на рис. 13.5):

```
SELECT *
FROM Customers
WHERE rating > ANY
   (SELECT rating
    FROM Customers
    WHERE city = 'Rome');
=============== SQL Execution Log ============
  SELECT *
  FROM Customers
  WHERE rating > ANY
     (SELECT rating
      FROM Customers
```

```
      WHERE city = 'Rome');
| ================================== |
| cnum   cname     city  rating  snum |
| ---------------------------------- |
| 2002  Giovanni  Rome      200  1003 |
| 2003  Liu       San Jose  200  1002 |
| 2004  Grass     Berlin    300  1002 |
| 2008  Cisneros  San Jose  300  1007 |
======================================
```

Рис. 13.5: Больше, чем ANY в интерпретации SQL

В английском языке выражение "рейтинг больше, чем *любой (any)* другой (где поле city равно Rome)", обычно интерпретируется так: значение данного рейтинга должно быть выше, чем значение рейтинга для *каждого (every)* случая, когда значение поля city равно Rome. Однако в SQL оператор ANY интерпретируется иначе. ANY оценивается как истина, если подзапрос находит *любое (any)* значение (любые значения), удовлетворяющее (удовлетворяющие) условию.

Если бы ANY интерпретировалось как обычное слово английского языка, покупатели с рейтингом 300 оказались бы выше Giovanni, находящегося в Rome и имеющего рейтинг 200. Однако, подзапрос с ANY находит также и Periera из Rome с рейтингом 100. Все покупатели с рейтингом 200 были выбраны (поскольку их рейтинги превышают 100) несмотря на то, что в Rome был другой покупатель (Giovanni), рейтинг которого выше 200 (то, что один из выбранных покупателей тоже находится в Rome, здесь значения не имеет). Они были выбраны, поскольку подзапрос сгенерировал значение, сделавшее предикат истинным для этих строк.

Другой пример: предположим, нужно выбрать все заказы, величина которых превосходит величину по крайней мере одного из заказов, сделанных 6 октября 1990 года:

```
SELECT *
FROM Orders
WHERE amt > ANY
   (SELECT amt
    FROM Orders
    WHERE odate = '10/06/1990');
```

Выходные данные для запроса представлены на рис. 13.6.

```
=============== SQL Execution Log ==============
  SELECT *
  FROM Orders
  WHERE amt > ANY
    (SELECT amt
     FROM Orders
     WHERE odate = '10/06/1990)';
| ================================== |
| onum   amt       odate     cnum  snum |
| ---------------------------------- |
| 3002  1900.10  10/03/1990  2007  1004 |
| 3005  5160.45  10/03/1990  2003  1002 |
| 3009  1713.23  10/04/1990  2002  1003 |
```

```
| 3008  4723.00  10/05/1990  2006  1001 |
| 3011  9891.88  10/06/1990  2006  1001 |
========================================
```

Рис. 13.6: Выбор записей, значение поля amt которых превышает ANY на 6 октября 1990 г.

Даже если наибольшая величина заказов, представленных в таблице (9891.88), приходится на 6 октября, предшествующие строки имеют значения, превышающие значение поля amounts в другой строке таблицы за 6 октября 1990 года – 1309.95. Если бы использовался оператор отношения >= вместо просто >, эта строка тоже была бы выбрана, так как она равна сама себе.

Можно использовать ANY с другими SQL-средствами, например, с соединениями. Этот запрос отыскивает все заказы, величина которых меньше величины заказа любого покупателя в San Jose (выходные данные представлены на рис. 13.7):

```
SELECT *
FROM Orders
WHERE amt < ANY
   (SELECT amt
    FROM Orders a, Customers b
    WHERE a.cnum = b.cnum
       AND b.city = 'San Jose');
============== SQL Execution Log ==============
  SELECT *
  FROM Orders
  WHERE amt > ANY
    (SELECT amt
     FROM Orders a, Customers b
     WHERE a.cnum = b.cnum
        AND b.city = 'San Jose');
| ==================================== |
| onum    amt      odate     cnum  snum |
| ------------------------------------ |
| 3001    18.69  10/03/1990  2008  1007 |
| 3003   767.10  10/03/1990  2001  1001 |
| 3002  1900.10  10/03/1990  2007  1004 |
| 3006  1098.10  10/03/1990  2008  1007 |
| 3009  1713.23  10/04/1990  2002  1003 |
| 3007    75.10  10/04/1990  2004  1002 |
| 3008  4723.00  10/05/1990  2006  1001 |
| 3010  1309.88  10/06/1990  2004  1002 |
========================================
```

Рис. 13.7: Использование ANY с объединением

Даже если минимальное значение в таблице указано в заказе из San Jose, имеется другое, превышающее его, поэтому почти все строки были выбраны. Легко запомнить, что < ANY означает "меньше, чем наибольшее выбранное значение", а > ANY означает "больше, чем наименьшее выбранное значение". Фактически, такую команду можно сформулировать следующим образом (выходные данные представлены на рис. 13.8):

```
SELECT *
```

```
FROM Orders
WHERE amt <
   (SELECT MAX(amt)
    FROM Orders a, Customers b
    WHERE a.cnum = b.cnum
       AND b.city = 'San Jose');
=============== SQL Execution Log ==============
  SELECT *
  FROM Orders
  WHERE amt <
    (SELECT MAX (amt)
     FROM Orders a, Customers b
     WHERE a.cnum = b.cnum
        AND b.city = 'San Jose');
| ==================================== |
| onum    amt      odate     cnum  snum |
| ------------------------------------ |
| 3002  1900.10  10/03/1990  2007  1004 |
| 3005  5160.45  10/03/1990  2003  1002 |
| 3009  1713.23  10/04/1990  2002  1003 |
| 3008  4723.00  10/05/1990  2006  1001 |
| 3011  9891.88  10/06/1990  2006  1001 |
========================================
```

Рис. 13.8: Использование агрегатной функции вместо ANY

Специальный оператор *ALL*

Предикат с ALL принимает значение "истина", если *каждое (every)* значение, выбранное в процессе выполнения подзапроса, удовлетворяет условию, заданному в предикате внешнего запроса. Если бы было нужно, чтобы в предыдущем примере в состав выходных данных включались только те покупатели, рейтинг которых превышает рейтинг каждого покупателя в Rome, то необходимо было бы ввести следующую команду для получения результата, представленного на рис. 13.9:

```
SELECT *
FROM Customers
WHERE rating > ALL
   (SELECT rating
    FROM Customers
    WHERE city = 'Rome');
=============== SQL Execution Log ============
  SELECT *
  FROM Customers
  WHERE rating > ALL
    (SELECT rating
     FROM Customers
     WHERE city = 'Rome');
| ================================== |
| cnum  cname      city  rating  snum |
| ---------------------------------- |
| 2004  Grass     Berlin    300  1002 |
| 2008  Cisneros  San Jose  300  1007 |
```

```
=====================================
```

Рис. 13.9: Использование оператора ALL

Это предложение проверяет значения рейтинга для всех покупателей в Риме, а затем находит тех покупателей, рейтинг которых превышает рейтинг каждого покупателя из Рима. Наибольший рейтинг в Риме имеет Giovanni, его значение составляет 200. Следовательно, в состав выходных данных включаются только те покупатели, рейтинг которых превышает 200.

EXISTS можно использовать, как в случае с ANY, для получения альтернативной формулировки того же запроса (выходные данные представлены на рис. 13.10):

```
SELECT *
FROM Customers outer
WHERE NOT EXISTS
   (SELECT *
    FROM Customers inner
    WHERE outer. rating <= inner. rating
      AND inner.city = 'Rome');
```

```
=============== SQL Execution Log ============
  SELECT *
  FROM Customers outer
  WHERE NOT EXISTS
    (SELECT *
     FROM Customers inner
     WHERE outer rating = inner.rating
        AND inner.city = 'Rome');
| ================================== |
| cnum  cname      city  rating  snum |
| ---------------------------------- |
| 2004  Grass     Berlin    300  1002 |
| 2008  Cisneros  San Jose  300  1007 |
=====================================
```

Рис. 13.10: Использование EXISTS в качестве альтернативы к ALL

Равенства и неравенства

ALL, как правило, используется с неравенствами, а не с равенствами, поскольку значение "равно всем" (equal to all), которое должно получиться в результате выполнения подзапроса, может быть получено в случае, если все результаты идентичны. Следующий запрос выглядит так:

```
SELECT *
FROM Customers
WHERE rating = ALL
   (SELECT rating
    FROM Customers
    WHERE city = 'San Jose');
```

Команда задана корректно, но в настоящем примере выходные данные не будут получены. Единственный случай, когда этот запрос продуцирует выходные данные, – значения рейтинга всех покупателей из San Jose одинаковы. Тогда запрос можно сформулировать иначе:

```
SELECT *
FROM Customers
WHERE rating =
   (SELECT DISTINCT rating
    FROM Customers
    WHERE city = 'San Jose');
```

Основное различие заключается в признании последней команды ошибочной, если в результате выполнения подзапроса будет получено множество значений; в этом случае вариант с ALL просто не генерирует никаких выходных данных. Поскольку в действительности база данных постоянно изменяется, неразумно заведомо делать какие-либо предположения относительно ее содержания.

Тем не менее, ALL можно эффективно использовать с неравенствами, то есть с оператором <>. Однако, фраза "значение неравно всем результатам подзапроса" выражается не так, как в привычном английском языке. Если подзапрос генерирует множество различных значений, как чаще всего это и бывает, то никакое одно значение не может быть равно всем значениям в обычном смысле. В SQL <> ALL реально означает "не равно ни одному" из результатов подзапроса. Другими словами, предикат имеет значение истина, если значение не принадлежит множеству результатов подзапроса. Следовательно, предыдущий запрос можно сформулировать, например, так (выходные данные для запроса представлены на рис. 13.11):

```
SELECT *
FROM Customers
WHERE rating <> ALL
   (SELECT rating
    FROM Customers
    WHERE city = 'San Jose');
```

```
=============== SQL Execution Log ============
  SELECT *
  FROM Customers
  WHERE rating <> ALL
    (SELECT rating
     FROM Customers
     WHERE city = 'San Jose');
| ================================ |
| cnum   cname   city  rating snum |
| -------------------------------- |
| 2001  Hoffman  London  100  1001 |
| 2006  Clemens  London  100  1001 |
| 2007  Pereira  Rome    100  1004 |
====================================
```

Рис. 13.11: Использование ALL с символом <>

Этот подзапрос выбирает все рейтинги, для которых в поле city указано значение San Jose. В результате получается множество из двух значений: 200 (для Liu) и 300 (для Cisneros). Основной запрос затем выбирает все строки, в которых значение поля rating отличается от этих значений, т.е. все строки со значением рейтинга l00. Этот же запрос можно сформулировать с NOT IN:

```
SELECT *
FROM Customers
WHERE rating NOT IN
   (SELECT rating
    FROM Customers
    WHERE city = 'San Jose');
```

Можно также использовать ANY:

```
SELECT *
FROM Customers
WHERE NOT rating ANY
   (SELECT rating
     FROM Customers
    WHERE city = 'San Jose');
```

Выходные данные одинаковы для трех последних запросов.

Непосредственная поддержка *ANY* и *ALL*

В языке SQL выражение "значение больше (или меньше), чем любое (ANY) из множества значений" эквивалентно выражению "значение больше (или меньше) какого-либо из множества значений". Соответственно, "значение не равно всему множеству значений (not equal ALL)" означает "в этом множестве нет значений, с которым оно совпадает".

Функционирование *ANY, ALL* и *EXISTS* при потере данных или с неизвестными данными

Ранее упоминалось, что существуют некоторые различия между EXISTS и введенными в этой главе операторами, касающиеся обработки NULL-значений. ANY и ALL также отличаются друг от друга по своей реакции, когда в результате выполнения подзапроса не получено значений, которые могут использоваться в сравнении. Если эти отличия не принять во внимание, они могут привести к неожиданным результатам.

Когда подзапрос возвращает значение *EMPTY*

Важное различие между ALL и ANY заключается в их реакции на ситуацию, когда подзапрос не генерирует никакого значения. Когда

правильный подзапрос не генерирует выходных данных, ALL автоматически принимает значение "истина", а ANY – "ложь".

Это означает, что следующий запрос:

```
SELECT *
FROM Customers
WHERE rating > ANY
   (SELECT rating
    FROM Customers
    WHERE city = 'Boston');
```

не генерирует выходных данных, тогда как запрос:

```
SELECT *
FROM Customers
WHERE rating > ALL
   (SELECT rating
    FROM Customers
    WHERE city = 'Boston');
```

полностью воспроизведет таблицу Customers. Поскольку в городе Boston нет никаких покупателей, все эти сравнения не имеют большого значения.

ANY и *ALL* вместо *EXISTS* с *NULL*-значениями

NULL-значения также создают некоторые проблемы для рассматриваемых операторов. Когда SQL сравнивает два значения, одно из которых – NULL-значение, результат принимает значение unknown. Unknown-предикат, также как и false-предикат, создает ситуацию, когда строка не включается в состав выходных данных, но результат различен для различных типов запросов, в зависимости от использования в них ALL или ANY вместо EXISTS. Рассмотрим приведенные ранее примеры:

```
SELECT *
FROM Customers
WHERE rating > ANY
   (SELECT rating
    FROM Customers
    WHERE city = 'Rome');
```

и:

```
SELECT *
FROM Customers outer
WHERE EXISTS
   (SELECT *
    FROM Customers inner
    WHERE outer.rating > inner.rating
       AND inner.city = 'Rome');
```

Оба эти запроса ведут себя совершенно одинаково. Предположим, что в таблице Customers есть строка с NULL-значением в столбце rating:

CNUM	*CNAME*	*CITY*	*RATING*	*SNUM*
2003	**Liu**	**San Jose**	**NULL**	**1002**

В версии с ANY, когда выбирается поле rating для Mr. Liu в основном запросе, из-за NULL-значения предикат принимает значение unknown, а строка с Liu не включается в состав выходных данных. Однако, когда версия NOT EXISTS выбирает эту строку в основном запросе, NULL-значение используется в предикате подзапроса, присваивая ему всякий раз значение unknown. Это значит, что в результате выполнения подзапроса не будет получено ни одного значения и EXISTS примет значение "ложь". Это, в свою очередь, сделает NOT EXISTS истинным. Следовательно, строка с Mr. Liu включается в состав выходных данных. Противоречие вытекает из того, что в отличие от других типов предикатов, значение EXISTS – всегда "истина" или "ложь", а никогда не unknown.

Это и является основанием для использования ANY. NULL-значение не превосходит любого реального значения. Более того, результат будет тот же, и необходимо искать меньшее значение.

Использование *COUNT* вместо *EXISTS*

Было отмечено, что ANY и ALL могут быть (приближенно) заменены на EXISTS, тогда как обратное неверно. Верно и то, что подзапросы с EXISTS и NOT EXISTS могут быть заменены теми же самыми подзапросами с COUNT(*) в предложении подзапроса SELECT. Если в состав выходных данных входит более чем 0 строк, то это эквивалентно ситуации EXISTS; в противном случае это то же самое, что NOT EXISTS. Рассмотрим пример (выходные данные для запроса представлены на рис. 13.12):

```
SELECT *
FROM Customers outer
WHERE NOT EXISTS
   (SELECT *
    FROM Customers inner
    WHERE outer.rating <= inner.rating
       AND inner.city = 'Rome');
=============== SQL Execution Log ============
  SELECT *
  FROM Customers outer
  WHERE NOT EXISTS
    (SELECT *
     FROM Customers inner
     WHERE outer.rating <= inner.rating
        AND inner.city = 'Rome');
| ================================== |
| cnum   cname     city  rating  snum |
| ---------------------------------- |
| 2004  Grass     Berlin    300  1002 |
```

```
| 2008  Cisneros  San Jose  300  1007 |
=======================================
```

Рис. 13.12: Использование EXISTS со связанным подзапросом

Его можно представить и так:

```
SELECT *
FROM Customers outer
WHERE 1 >
   (SELECT COUNT(*)
    FROM Customers inner
    WHERE outer.rating <= inner.rating
       AND inner.city = 'Rome');
```

Выходные данные для этого запроса изображены на рис. 13.13.

```
=============== SQL Execution Log ============
  SELECT *
  FROM Customers outer
  WHERE 1 >
    (SELECT COUNT (*)
     FROM Customers inner
     WHERE outer.rating <= inner.rating
        AND inner.city = 'Rome');
| ================================== |
| cnum   cname    city    rating  snum |
| ---------------------------------- |
| 2004  Grass     Berlin    300   1002 |
| 2008  Cisneros  San Jose  300   1007 |
=======================================
```

Рис. 13.13: Использование COUNT вместо EXISTS

РЕЗЮМЕ

Эта глава содержит большой объем информации. Подзапросы – тема непростая, поэтому по мере изложения материала обсуждались и возможные варианты, и неоднозначности. Вы узнали о разных методах работы с ошибками и NULL-значениями. У вас есть теперь несколько способов решения проблемы и возможность выбора оптимального варианта.

После разбора наиболее важного и сложного аспекта SQL, оставшийся материал сравнительно прост для понимания. Следующая глава тоже посвящена запросам. Вы научитесь комбинировать выходные данные для любого количества запросов в единственном теле с целью формирования объединения множества запросов с использованием предложения UNION.

Глава 14

ИСПОЛЬЗОВАНИЕ ПРЕДЛОЖЕНИЯ *UNION*

В предшествующих главах обсуждались различные варианты запросов с расположением "один внутри другого". Существует другой способ комбини-

рования множества запросов – их объединение. В этой главе объясняются предложения UNION в SQL. *Объединения (unions)* отличаются от подзапросов тем, что любой из двух (или большего числа) запросов не может управлять другим запросом. В объединении все запросы выполняются независимо, но их выходные данные затем объединяются.

Объединение множества запросов в один

Можно задать множество запросов одновременно и комбинировать их выходные данные с использованием предложения UNION. UNION объединяет выходные данные двух или более SQL-запросов в единое множество строк и столбцов. Для того чтобы получить сведения обо всех продавцах (salespeople) и покупателях (customers) Лондона в виде выходных данных одного запроса, следует ввести:

```
SELECT snum, sname
   FROM Salespeople
   WHERE city = 'London'

   UNION

SELECT cnum, cname
   FROM Customers
   WHERE city = 'London';
```

(выходные данные представлены на рис. 14.l).

```
=============== SQL Execution Log ============
  SELECT snum, sname
    FROM Salespeople
    WHERE city = 'London'

    UNION

  SELECT cnum, cname
    FROM Customers
    WHERE city = 'London';
| ==============|
|               |
| ------------- |
| 1001  Peel    |
| 1004  Motika  |
| 2001  Hoffman |
| 2006  Climens |
=================
```

Рис. 14.1: Формирование объединения из двух запросов

Столбцы, выбранные с помощью двух команд, представлены в выходных данных так, как если бы они выбирались с помощью одного запроса. Заголовки столбцов опущены, поскольку в результат объединения входят столбцы из нескольких разных таблиц. Таким образом, столбцы в выходных данных не поименованы.

Только последний запрос заканчивается точкой с запятой. Отсутствие этого знака дает возможность SQL распознать, что следует еще один запрос.

Когда можно выполнить объединение запросов?

Для того, чтобы два или более запроса можно было объединить (выполнить команду UNION), их столбцы, входящие в состав выходных данных, должны быть *совместимы по объединению (union compatible)*. Это значит, что в каждом из запросов может быть указано одинаковое количество столбцов в таком порядке: первый, второй, третий и т.д., – причем, первые столбцы каждого из запросов являются сравнимыми, вторые столбцы каждого из них – также сравнимы и т.д. по всем столбцам, включаемым в состав выходных данных. Значение термина "столбцы сравнимы" может меняться. ANSI определяет его очень просто: числовые поля должны иметь полностью совпадающие тип и размер. Символьные поля должны иметь точно совпадающее количество символов (это означает, что одинаковое количество выделено, но вовсе не обязательно заполнено).

Некоторые программные продукты SQL используют более гибкие определения. Например, нестандартные с точки зрения ANSI типы, такие как DATE и BINARY, обычно сравнимы со столбцами этих же нестандартных типов. Длина столбцов тоже является проблемой. Многие программные продукты не требуют совпадения длин, но такие столбцы нельзя использовать в UNION. С другой стороны, некоторые продукты (и ANSI) требуют, чтобы длины символьных полей точно совпадали. По этим причинам всегда следует ознакомиться с документацией по конкретному программному продукту.

Другое ограничение на сравнимость состоит в том, что если NULL-значения запрещены для любого столбца в объединении, то они должны быть запрещены для всех соответствующих столбцов в других запросах объединения. NULL-значения запрещаются с помощью ограничения NOT NULL. Нельзя использовать UNION в подзапросах, также как и функции агрегирования в предложениях SELECT запросов в объединении (многие программные продукты смягчают эти ограничения).

***UNION* и устранение дублирования**

UNION автоматически исключает из выходных данных дублирующиеся строки. Не возбраняется, но и не имеет особого смысла применение в SQL оператора DISTINCT в отдельных запросах для исключения повторяющихся значений. Примером может служить следующий запрос, выходные данные для которого изображены на рис. 14.2:

```
SELECT snum, city
FROM Customers;
```

```
=============== SQL Execution Log ============
  SELECT snum, city
  FROM Customers;
```

```
| ============== |
| snum  city     |
| -------------- |
| 1001  London   |
| 1003  Rome     |
| 1002  San Jose |
| 1002  Berlin   |
| 1001  London   |
| 1003  Rome     |
| 1007  San Jose |
==================
```

Рис. 14.2: Простой запрос с повторяющимися выходными данными

Среди них есть повторяющаяся комбинация значений (1001 с London), поскольку в запросе SQL не требуется исключать дубликаты (повторяющиеся значения). Однако, если применяется UNION для комбинации этого запроса с таким же запросом для таблицы Salespeople, избыточная комбинация исключается. На рис. 14.3 представлены выходные данные для следующего запроса:

```
SELECT snum, city
  FROM Customers

  UNION

SELECT snum, city
  FROM Salespeople;
```

```
=============== SQL Execution Log ============
  SELECT snum, city
    FROM Customers

    UNION

  SELECT snum, sity
    FROM Salespeople;
| =============== |
|                 |
| --------------- |
| 1001  London    |
| 1002  San Jose  |
| 1002  Berlin    |
| 1007  San Jose  |
| 1007  New York  |
| 1003  Rome      |
| 1002  Barcelona |
===================
```

Рис. 14.3: Объединение исключает повторяющиеся выходные данные

Можно добиться того же (в некоторых программных продуктах SQL), указав UNION ALL вместо UNION:

```
SELECT snum, city
  FROM Customers

  UNION ALL
```

```
SELECT snum, city
  FROM Salespeople;
```

Использование строк и выражений с *UNION*

Иногда можно вставлять константы и выражения в предложения SELECT, использующие UNION. Это не соответствует в точности стандарту ANSI, но часто и оправданно применяется. Однако применяемые константы и выражения должны при этом удовлетворять стандарту сравнимости, о котором упоминалось ранее. Такая процедура может оказаться полезной, например, для формулировки комментария, определяющего, из какого конкретно запроса получена данная строка.

Предположим, необходимо сделать отчет, содержащий сведения для каждого продавца о его максимальном и минимальном заказах по каждой дате. Можно объединить два запроса, вставив соответствующий текст в качестве комментария, для того чтобы различить каждый из двух случаев (минимальный заказ и максимальный заказ).

```
SELECT a.snum, sname, onum, 'Highest on', odate
   FROM Salespeople a, Orders b
   WHERE a.snum = b.snum
      AND b.amt =
        (SELECT MAX (amt)
            FROM Orders c
            WHERE c.odate = b.odate)

UNION

SELECT a.snum, sname, onum, 'Lowest on ', odate
   FROM Salespeople a, Orders b
   WHERE a.snum = b.snum
       AND b.amt =
          (SELECT MIN (amt)
             FROM Orders c
             WHERE c.odate = b.odate);
```

Выходные данные для этих команд представлены на рис. 14.4.

```
| ------------------------------------ |
| 1001 Peel    3008 Highest on 10/05/1990 |
| 1001 Peel    3008 Lowest on  10/05/1990 |
| 1001 Peel    3011 Highest on 10/06/1990 |
| 1002 Serres  3005 Highest on 10/03/1990 |
| 1002 Serres  3007 Lowest on  10/04/1990 |
| 1002 Serres  3010 Lowest on  10/06/1990 |
| 1003 Axelrod 3009 Highest on 10/04/1990 |
| 1007 Rifkin  3001 Lowest on  10/03/1990 |
==========================================
```

Рис. 14.4: Выбор наибольшей (highest) и наименьшей (lowest) заявок с поясняющими строками текста

Необходимо добавить дополнительный пробел в строку 'Lowest on', для того чтобы длина этой строки соответствовала длине строки 'Highest on'. Peel выбран дважды, как имеющий максимальный и минимальный заказы на 5 октября 1990 г. Это произошло потому, что вставляемые строки для двух запросов различны, следовательно, строки выходных данных не удаляются автоматически как дублирующиеся.

Использование *UNION* с *ORDER BY*

До сих пор мы не придавали значения порядку представления данных множества запросов в выходных данных. Сначала представлялись выходные данные для первого запроса, затем – для второго. Нельзя считать, что данные автоматически будут следовать именно в таком порядке. Этот способ расположения (представления) выходных данных выбран исключительно для более простого восприятия результатов выполнения команды. Однако, предложение ORDER BY применяется и для упорядочения выходных данных объединения, так же как это делалось для отдельных (индивидуальных) запросов. Можно пересмотреть последний пример, предположив необходимость упорядочения выходных данных запроса. В этом случае станет ясно, почему, например, имя Peel в выходных данных предыдущего запроса многократно повторялось:

```
SELECT a.snum, sname, onum, 'Highest on', odate
   FROM Salespeople  a, Orders b
   WHERE a.snum = b.snum
      AND b.amt =
        (SELECT MAX (amt)
            FROM Orders c
            WHERE c.odate = b.odate)

UNION

SELECT a.snum, sname, onum, 'Lowest on ', odate
   FROM Salespeople a, Orders b
   WHERE a.snum = b.snum
      AND b.amt =
        (SELECT MIN (amt)
            FROM Orders c
            WHERE c.odate = b.odate)
ORDER BY 3;
```

Выходные данные для этого запроса представлены на рис. 14.5.

```
| ======================================== |
|                                          |
```

```
| ---------------------------------------- |
| 1007  Rifkin   3001  Lowest on  10/03/1990 |
| 1002  Serres   3005  Highest on 10/03/1990 |
| 1002  Serres   3007  Lowest on  10/04/1990 |
| 1001  Peel     3008  Highest on 10/05/1990 |
| 1001  Peel     3008  Lowest on  10/05/1990 |
| 1003  Axelrod  3009  Highest on 10/04/1990 |
| 1002  Serres   3010  Lowest on  10/06/1990 |
| 1001  Peel     3011  Highest on 10/06/1990 |
==========================================
```

Рис. 14.5: Формирование объединения с использованием ORDER BY

Поскольку способ упорядочения по возрастанию (ASC) для ORDER BY принят по умолчанию, он специально не указывается. Можно упорядочить выходные данные в соответствии со значениями нескольких полей: для каждого из полей независимо по возрастанию или убыванию (ASC или DESC), как это делалось для выходных данных одного запроса. Число 3 в предложении ORDER BY задает номер столбца в упорядоченном списке предложения SELECT. Поскольку столбцы выходных данных, полученных в результате выполнения объединения, являются непоименованными, на столбец можно сослаться только по номеру, определяющему его место расположения среди столбцов выходных данных.

Внешнее соединение

Часто бывает полезна операция объединения двух запросов, в которой второй запрос выбирает строки, исключенные первым. Обычно это приходится делать для исключения строк, не удовлетворяющих предикату при выполнении операции соединения таблиц. Это называется *внешним соединением (outer join).* Предположим, у некоторых из покупателей еще нет продавцов. Можно посмотреть имена и города всех покупателей с указанием имен их продавцов, не отбрасывая покупателей, еще не имеющих продавцов. Можно получить желаемые сведения, сформировав объединение двух запросов, один из которых выполняет объединение, а второй выбирает покупателей с NULL-значением в поле snum. Последний может вставлять пробелы в поле, соответствующее полю sname первого запроса.

Для идентификации запроса, с помощью которого получена данная строка, можно вставлять строки текста в выходные данные. Использование этой возможности во внешнем соединении позволяет применять предикаты для классификации выходных данных, а не для их исключения.

Пример поиска продавцов (salespeople) с покупателями (customers), расположенными в тех же городах, рассматривался ранее. Теперь необходимо в составе выходных данных увидеть список всех продавцов и пометить тех, кто не имеет покупателей, находящихся в их городе (city), так же как и тех, кто таких покупателей имеет. Следующий запрос, выходные данные для которого представлены на рис.14.6, позволяет сделать это:

```
SELECT Salespeople.snum, sname, cname, comm
```

```
   FROM Salespeople, Customers
   WHERE Salespeople.city = Customers.city

   UNION

SELECT snum, sname, 'NO MATCH    ', comm
   FROM Salespeople
   WHERE NOT city = ANY
     (SELECT city
      FROM Customers)
ORDER BY 2 DESC;
```

```
| -------------------------- |
| 1002 Serres  Cisneros  0.13 |
| 1002 Serres  Liu       0.13 |
| 1007 Rifkin  NO MATCH  0.15 |
| 1001 Peel    Clemens   0.12 |
| 1001 Peel    Hoffman   0.12 |
| 1004 Motika  Clemens   0.11 |
| 1004 Motika  Hoffman   0.11 |
| 1003 Axelrod NO MATCH  0.10 |
==============================
```

Рис. 14.6: Внешнее соединение

Строка 'NO MATCH' дополнена пробелами так, чтобы она соответствовала полю cname по длине (практически в этом нет необходимости для всех программных реализаций SQL). Второй запрос выбирает строки, не соответствующие предикату первого запроса.

В запрос можно добавить комментарий или выражение в качестве дополнительного поля. Для этого необходимо включить некоторый совместимый комментарий или выражение в соответствующую позицию списка имен полей предложения SELECT для каждого запроса в операторе объединения. Сравнимость по объединению предотвращает ситуации, когда дополнительное поле добавляется к одному из запросов, но не добавляется к другому. Вот пример запроса, который добавляет строки к выбранным полям. Текст этих строк сообщает о наличии для данного продавца назначенного ему покупателя из его же города ('MATCHED' – 'NO MATCH'):

```
SELECT a.snum, sname, a.city, 'MATCHED '
  FROM Salespeople a, Customers b
  WHERE a.city = b.city

  UNION

SELECT snum, sname, city, 'NO MATCH'
   FROM Salespeople
   WHERE NOT city = ANY
     (SELECT city
        FROM Customers)
```

ORDERD BY 2 DESC;

Выходные данные для этого запроса представлены на рис. 14.7.

```
| ------------------------------ |
| 1002 Serres   San Jose  MATCHED  |
| 1007 Rifkin   Barselona NO MATCH |
| 1001 Peel     London    MATCHED  |
| 1004 Motika   London    MATCHED  |
| 1003 Axelrod  New York  NO MATCH |
====================================
```

Рисунок 14.7: Внешнее соединение с полями комментария

Это неполное внешнее соединение, поскольку оно включает только не назначенные (unmatched) поля для одной из участвующих в соединении таблиц. Полное внешнее соединение должно включать всех покупателей, которые как имеют, так и не имеют продавцов в их городах. Очевидно, что это гораздо сложнее (выходные данные для следующего запроса представлены на рис. 14.8):

```
(SELECT snum, city, 'SALESPERSON – MATCHED'
   FROM Salespeople
   WHERE city = ANY
     (SELECT city
        FROM Customers)

UNION

SELECT snum, city, 'SALESPERSON – NO MATCH'
   FROM Salespeople
   WHERE NOT city = ANY
      (SELECT city
         FROM Customers))

UNION

(SELECT cnum, city, 'CUSTOMER – MATCHED'
    FROM Customers
    WHERE city = ANY
      (SELECT city
         FROM Salespeople))

UNION

SELECT cnum, city, 'CUSTOMER – NO MATCH'
   FROM Customers
   WHERE NOT city = ANY
     (SELECT city
```

```
        FROM Salespeople))
ORDER BY 2 DESC;
```

(Эта формулировка, использующая ANY, эквивалентна соединению в предыдущем примере.)

```
| ---- -------- ---------------------- |
| 2003 San Jose  CUSTOMER — MATCHED     |
| 2008 San Jose  CUSTOMER — MATCHED     |
| 2002 Rome      CUSTOMER — NO MATCH    |
| 2007 Rome      CUSTOMER — NO MATCH    |
| 1003 New York  SALESPERSON — MATCHED  |
| 1003 New York  SALESPERSON — NO MATCH |
| 2001 London    CUSTOMER — MATCHED     |
| 2006 London    CUSTOMER — MATCHED     |
| 2004 Berlin    CUSTOMER — NO MATCH    |
| 1007 Barcelona SALESPERSON — MATCHED  |
| 1007 Barcelona SALESPERSON — NO MATCH |
=========================================
```

Рисунок 14.8: Сложное внешнее соединение

Сокращенные внешние соединения, с которых началось рассмотрение, оказываются полезными чаще, чем полные (рассмотрено в последнем примере). По поводу последнего примера можно заметить: если объединение выполняется более чем для двух запросов, то для упорядочения вычислений нужно использовать круглые скобки. Иначе говоря, вместо того, чтобы задать

```
query X UNION query Y UNION query Z;
```

необходимо конкретизировать либо

```
(query X UNION query Y) UNION query Z;
```

либо

```
query X UNION (query Y UNION query Z);
```

Это необходимо, потому что UNION и UNION ALL можно комбинировать для исключения одних дубликатов без устранения других. Предложение

```
(query X UNION ALL query Y) UNION query Z;
```

необязательно генерирует те же выходные данные, что и предложение

```
query X UNION ALL (query Y UNION query Z);
```

если имеются дубликаты строк, подлежащие исключению из выходных данных.

РЕЗЮМЕ

Теперь вы знаете, как использовать предложение UNION, позволяющее комбинировать любое количество запросов в единственном теле запроса. Если есть ряд сходных таблиц, т.е. содержащих одинаковую информацию, но принадлежащих разным пользователям и имеющих различную специфику, объединение может стать легким способом слить и упорядочить выходные данные. Внешнее соединение – это новый способ применять условия, не

исключая выходные данные, а только помечая или выделяя их части как удовлетворяющие и неудовлетворяющие данному условию.

Глава 15

ВВОД, УДАЛЕНИЕ И ИЗМЕНЕНИЕ ЗНАЧЕНИЙ ПОЛЕЙ

В этой главе рассматриваются команды, управляющие представленными в таблице значениями в любой момент времени. Вы научитесь размещать столбцы в таблице, исключать их и изменять отдельные значения, представленные в каждой строке. Вы познакомитесь здесь с применением запросов для генерации групп строк для вставки, а также с использованием предикатов для управления изменениями значений и удалением строк. Материал этой главы содержит большой объем сведений, необходимых для манипулирования информацией в базе данных. Некоторые более сложные способы построения предикатов будут рассмотрены в следующей главе.

Команды обновления *DML*

Данные заносятся в поля и исключаются из них с помощью трёх команд языка манипулирования данными (Data Manipulation Language – DML): INSERT (вставить), UPDATE (обновить) и DELETE (удалить). В SQL их часто называют командами обновления *(update commands).*

Ввод значений

Все строки в SQL вводятся при помощи команды обновления INSERT. В простейшем случае команда INSERT имеет такой синтаксис:

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

Например, для того чтобы ввести строку в таблицу Salespeople, можно использовать следующее предложение:

```
INSERT INTO Salespeople
VALUES (1001, 'Peel', 'London', 0.12);
```

Команды языка манипулирования данными не генерируют выходных данных, но программа должна уведомлять пользователя о том, что данные добавлены. Имя таблицы (в данном случае Salespeople) должно быть определено предварительно (до выполнения команды INSERT) с помощью команды CREATE TABLE (см. главу 17), а каждое значение в списке значений должно иметь тип данных, соответствующий типу данных столбца, в который это значение должно быть вставлено. Согласно стандарту ANSI, эти значения не могут включать выражения. Отсюда следует, что значение 3 допустимо, а 1 + 2 – недопустимо. Значения, конечно, вводятся в таблицу в порядке следования столбцов таким образом, что первое из значений, указанных в списке VALUES

команды INSERT, вводится автоматически в столбец с номером 1, второе – в столбец с номером 2 и т.д.

Вставка NULL-значений

Если нужно вставить NULL-значение, необходимо указать его как обычное значение. Предположим, значение поля city для Ms.Peel еще неизвестно. В этом случае для нее можно вставить строку, указав значение NULL в столбце city, следующим образом:

```
INSERT INTO Salespeople
VALUES (1001, 'Peel', NULL, 0.12);
```

Поскольку NULL является специальным символом, а не символьным значением, оно указано без одиночных кавычек.

Именование столбцов для *INSERT*

Для указания имен столбцов, в которые необходимо ввести значения, порядок столбцов в таблице неважен. Предположим, значения для таблицы Customers берутся из напечатанного отчета, в котором интересующие сведения представлены в таком порядке: city, cname, cnum. Для простоты желательно вводить значения в порядке, указанном в напечатанном отчете. Можно воспользоваться командой:

```
INSERT INTO Customers (city, cname, cnum)
VALUES ('London', 'Hoffman', 2001);
```

Столбцы с именами rating и snum опущены. Это означает, что для каждого из них автоматически назначаются значения по умолчанию. По умолчанию может быть установлено либо значение NULL, либо вполне определенное значение.

Вставка результатов запроса

Команду INSERT можно применить для того, чтобы извлечь значения из одной таблицы и разместить их в другой, воспользовавшись для этого запросом. Для этого достаточно заменить предложение VALUES на соответствующий запрос, как в примере:

```
INSERT INTO Londonstaff
  SELECT *
    FROM Salespeople
    WHERE city = 'London';
```

По этой команде все значения, полученные с помощью запроса (т.е. все строки таблицы Salespeople, для которых значение поля city = 'London'), размещаются в таблице с именем Londonstaff. Чтобы избежать проблем при выполнении команды, таблица Londonstaff должна удовлетворять следующим условиям:

- Она должна быть уже создана с помощью команды CREATE TABLE.
- Она должна иметь четыре столбца, соответствующих столбцам таблицы Salespeople в смысле типов данных: т.е. первый, второй и т.д. столбцы каждой из таблиц должны иметь один и тот же тип (использования одних и тех же имен не требуется).

Основное правило, в соответствии с которым столбцы таблицы вставляются, заключается в соответствии столбцов таблицы столбцам выходных данных запроса, в данном случае, таблице Salespeople целиком.

Londonstaff является здесь независимой таблицей, имеющей ряд значений, совпадающих со значениями таблицы Salespeople. Если значения в таблице Salespeople изменить, то эти изменения не отразятся на значениях, хранящихся в таблице Londonstaff. Поскольку и запрос, и команда INSERT могут указывать столбцы по имени, при желании можно переставить выбираемые столбцы (указать имена в команде SELECT) или указать имена вставляемых столбцов в произвольном порядке (указать имена в команде INSERT).

Предположим, решено создать новую таблицу с именем Daytotals, которая будет отслеживать объемы заказов за каждый день. Допустим, необходимо ввести эти данные независимо от таблицы Orders, воспользовавшись при этом уже имеющимися в ней данными. Предположим, таблица Orders содержит данные по последнему финансовому году, а не за несколько дней, как в нашем примере. В этом случае преимущества использования предложения INSERT для вычисления и ввода значений вполне очевидны:

```
INSERT INTO Daytotals (date, total)
  SELECT odate, SUM (amt)
    FROM Orders
    GROUP BY odate;
```

Заметим, что имена столбцов для двух таблиц Orders и Daytotals не совпадают. Но если date и total являются единственными столбцами таблицы и следуют в ней в указанном порядке, то имена в предложении INTO можно опустить.

Исключение строк из таблицы

Строки из таблицы можно исключить с помощью команды обновления DELETE. По этой команде исключаются только целые строки, а не отдельные значения полей. Для исключения всех строк таблицы Salespeople, следует ввести следующее предложение:

```
DELETE FROM Salespeople;
```

В результате выполнения этой команды таблица становится пустой и ее можно удалить по команде DROP TABLE (команда объясняется в главе 17).

Обычно из таблицы требуется удалить только некоторые указанные строки. Чтобы их определить, можно, как и для запросов, использовать предикат. Например, чтобы исключить продавца Axelrod из таблицы, следует ввести:

```
DELETE FROM Salespeople
   WHERE snum = 1003;
```

Поле snum используется вместо поля sname, поскольку наилучший способ при удалении единственной строки – это указать значение ее первичного ключа. Применение первичного ключа гарантирует удаление единственной строки.

Можно употребить и предикат, выбирающий группу строк, например:

```
DELETE FROM Salespeople
   WHERE city = 'London';
```

Изменение значений полей

По команде UPDATE можно изменять некоторые или все значения в существующей строке. Эта команда содержит предложение UPDATE, позволяющее указать имя таблицы, для которой выполняется операция, и SET предложение, определяющее изменение (изменения), которое необходимо выполнить для определенного столбца (столбцов). Например, для того чтобы изменить для всех покупателей рейтинг на 200, следует ввести:

```
UPDATE Customers
   SET rating = 200;
```

Обновление только определённых строк

Замена значения столбца во всех строках таблицы, как правило, не нужна. Поэтому в команде UPDATE, как и в команде DELETE, можно использовать предикат. Для выполнения указанной замены значений столбца rating, для всех покупателей, которые обслуживаются продавцом Peel (snum = 1001), следует ввести:

```
UPDATE Customers
   SET rating = 200
   WHERE snum = 1001;
```

***UPDATE* для множества столбцов**

Нет нужды ограничиваться обновлением значения единственного столбца в результате выполнения команды UPDATE. В предложении SET можно указать любое количество значений для столбцов, разделенных запятыми. Все указанные изменения выполняются для каждой строки таблицы, удовлетворяющей предикату. В каждый момент времени обрабатывается одна строка таблицы. Предположим, Motika заменена новым продавцом (salesperson), необходимо сохранить ее персональный номер, но в соответствующую строку таблицы внести данные о новом продавце:

```
UPDATE Salespeople
   SET sname = 'Gibson', city = 'Boston', comm = 0.10
   WHERE snum = 1004;
```

В результате выполнения команды все покупатели продавца Motika со своими заказами перейдут к Gibson, поскольку они связаны с Motika по значению поля snum. Однако невозможно обновить множество таблиц с помощью единственной команды, так как нельзя использовать имя таблицы в качестве префикса имени столбца в предложении SET. Т.е. нельзя указать:

```
... SET Salespeople.sname = 'Gibson' …
```

в команде UPDATE, можно только:

```
… SET sname = 'Gibson' ...
```

Использование выражений в *UPDATE*

В предложении SET команды UPDATE можно использовать скалярные выражения, указывающие способ изменения значений поля в отличие от предложения VALUES команды INSERT, в котором нельзя использовать выражения. Это весьма полезная характеристика. Предположим, решено удвоить комиссионные продавцов. Можно использовать следующее выражение:

```
UPDATE Salespeople
   SET comm = comm*2;
```

Поскольку есть ссылка на значение существующего столбца в предложении SET, для каждой текущей строки выбирается значение указанного столбца, над которым выполняется заданная операция (в данном случае значение увеличивается в два раза). Можно комбинировать отдельные компоненты предложения UPDATE. Например, можно изменить значение комиссионных только для продавцов из Лондона с помощью предложения:

```
UPDATE Salespeople
   SET comm = comm*2
   WHERE city = 'London';
```

Применение *UPDATE* к NULL-значениям

Предложение SET не является предикатом. В нем можно указать значение NULL без использования какого-либо специального синтаксиса (например, такого как IS NULL). Таким образом, если нужно установить все рейтинги покупателей из Лондона (city = 'London') равными NULL-значению, необходимо ввести следующее предложение:

```
UPDATE Customers
   SET rating = NULL
   WHERE city = 'London';
```

В результате выполнения этой команды значения рейтинга для всех покупателей из Лондона станут неопределенными (имеющими NULL-значение).

РЕЗЮМЕ

Вы овладели средствами манипулирования содержимым базы данных с помощью трех простых команд. INSERT используется для замены содержимого строк в базе данных, DELETE – для удаления строк, UPDATE – для замены значений в строках таблицы. Вы можете применять предикаты с UPDATE и DELETE для определения конкретных строк таблицы, на которые воздействует команда. Предикаты не имеют значения для команды INSERT, поскольку рассматриваемая в команде строка не существует до тех пор, пока команда не выполнится. Однако, можно использовать запросы с INSERT для добавления множества строк в таблицу во время выполнения одной команды INSERT. Эти операции осуществляются при любом порядке расположения строк. Вы узнали, что значения, присваиваемые "по умолчанию", заносятся в те столбцы, значения которых явно не заданы. В операции может участвовать NULL-значение. Вновь обращаем ваше внимание на то, что в команде UPDATE можно использовать выражения, а в INSERT – нельзя.

Глава 16

ИСПОЛЬЗОВАНИЕ ПОДЗАПРОСОВ С КОМАНДАМИ ОБНОВЛЕНИЯ

Из этой главы вы узнаете, как применять подзапросы в командах обновления. Эта операция похожа на использование подзапросов в запросах. Зная, как подзапросы применяются в командах SELECT, довольно просто, несмотря на некоторые отличия, освоить их использование в командах обновления.

Подзапросы полностью являются командами SELECT, а не предикатами, поэтому данная операция отличается от использования предикатов с командами обновления. Простые запросы уже применялись для получения значения для INSERT, но теперь необходимо расширить эти запросы включением в них подзапросов.

Важный принцип, который нужно помнить при использовании команд обновления, состоит в том, что в предложении FROM любого подзапроса нельзя ссылаться на таблицу, изменяемую в основной команде. Это применимо ко всем трем командам обновления. Существует множество ситуаций, в которых было бы полезно использовать запросы к изменяемой таблице в процессе ее модификации, но при этом, как полагают разработчики SQL, может возникнуть двусмысленность и определенная сложность в практической реализации.

Использование подзапросов в *INSERT*

INSERT является простейшим из рассматриваемых случаев. Вы уже знаете, как вставить результаты запроса в таблицу. Можно использовать подзапросы

внутри любого запроса, генерирующего значения для команды INSERT так же, как для других запросов – внутри предиката или предложения HAVING.

Предположим, есть таблица с именем SJpeople, определения столбцов которой полностью соответствуют определениям таблицы Salespeople. Известно, как заполнять таблицу, подобную этой, для всех покупателей (Customers), расположенных (city) в San Jose:

```
INSERT INTO SJpeople
   SELECT *
      FROM Salespeople
      WHERE city = 'San Jose';
```

Теперь можно использовать подзапрос, для того чтобы добавить в таблицу SJpeople всех продавцов, имеющих покупателей в San Jose, независимо от места проживания продавца:

```
INSERT INTO SJpeople
   SELECT *
      FROM Salespeople
      WHERE snum = ANY
        (SELECT snum
           FROM Customers
           WHERE city = 'San Jose');
```

Оба запроса в этой команде действуют так, будто они не являются частью выражения INSERT. Подзапрос отыскивает все строки для покупателей в San Jose и создает множество значений поля snum. Внешний запрос выбирает те строки из таблицы Salespeople, для которых найдены совпадающие значения snum. В данном примере в таблицу SJpeople вставляются строки для продавцов Rifkin и Serres, которым назначены покупатели из San Jose: Liu и Cisneros.

Включение (предотвращение включения) одинаковых строк

Последовательность команд в предыдущем разделе может показаться проблематичной. Продавец Serres расположен в San Jose и, следовательно, будет добавлен в таблицу после выполнения первой команды. Вторая команда будет пытаться добавить его снова, поскольку он имеет покупателя в San Jose. Если имеются какие-либо ограничения для SJpeople, в соответствии с которыми значения должны быть уникальными, это второе появление (вторая вставка той же самой строки) может оказаться ошибочным. Дублирование строк – не самая удачная идея. Для того, чтобы проконтролировать наличие в таблице значения, которое вставляется добавлением другого подзапроса (с использованием операторов EXISTS, IN, <>ALL и т.д.) для предиката, нужна ссылка на саму таблицу SJpeople в предложении FROM этого нового подзапроса, но нельзя ссылаться на таблицу, находящуюся в процессе формирования (как единого целого), в любом подзапросе команды обновления. В случае с INSERT, это также исключает возможность использования связанных подзапросов, базирующихся на таблице, в которую вносятся

значения, что существенно, поскольку при использовании INSERT создается новая строка таблицы. "Текущая" строка не существует до тех пор, пока INSERT не закончит ее обрабатывать.

Использование подзапросов, основанных на таблицах внешних запросов

Запрет на использование ссылок на таблицу, находящуюся в процессе изменения командой INSERT, не мешает применять подзапросы, которые ссылаются на таблицу (таблицы), используемую в предложении FROM внешней команды SELECT. Таблица, из которой осуществляется выборка данных для получения значения для INSERT, не находится в состоянии изменения командой, и на нее можно ссылаться любым известным способом, как если бы это было отдельным запросом.

Предположим, существует таблица Samecity, в которой хранятся сведения о продавцах (salespeople), имеющих покупателей в их же городах (cities). Можно заполнить таблицу, применяя связанные подзапросы:

```
INSERT INTO Samecity
   SELECT *
     FROM Salespeople outer
     WHERE city IN
        (SELECT city
            FROM Customers inner
            WHERE inner.snum = outer.snum);
```

Здесь таблицу Samecity нельзя использовать во внешних или внутренних запросах для INSERT.

Другой пример: предположим, назначено вознаграждение для каждого продавца, имеющего максимальный заказ на каждый день. Сведения о них можно формировать в таблице Bonus, содержащей значение поля snum для продавца, дату (odate) и объем заказа (amount, amt). Эту таблицу можно заполнить информацией, хранящейся в таблице Orders, используя следующую команду:

```
INSERT INTO Bonus
   SELECT snum, odate, amt
      FROM Orders a
      WHERE amt =
         (SELECT MAX (amt)
            FROM Orders b
            WHERE a.odate = b.odate);
```

Несмотря на то, что команда имеет подзапрос, базирующийся на той же таблице, что и внешний запрос, она не ссылается на таблицу Bonus, на которую воздействует эта команда. Следовательно, ее можно использовать. Логика

запроса заключается в просмотре таблицы Orders, и для каждой строки осуществляется поиск максимальной заявки для конкретной даты. Если ее величина совпадает со значением поля amt текущей строки, то эта текущая строка и интересна, и ее данные заносятся в таблицу Bonus.

Использование подзапросов с *DELETE*

В предикате команды DELETE можно использовать подзапросы. Это дает возможность формулировать достаточно сложные критерии удаления строк, что требует особой внимательности. Например, если необходимо закрыть лондонский офис, можно использовать следующий запрос для исключения всех покупателей, назначенных продавцам в London:

```
DELETE
   FROM Customers
   WHERE snum = ANY
     (SELECT snum
        FROM Salespeople
        WHERE city = 'London');
```

В соответствии с этой командой из таблицы Customers будут исключены строки Hoffman, Clemens (обе назначены Peel) и строка Periera (назначенная Motika). Желательно удостовериться в верном выполнении этой операции, прежде чем реально удалить или изменить строки с Peel и Motika.

Внимание! Когда выполняется изменение в базе данных, которое влечет другие изменения, первое желание – это выполнить сначала основное изменение, а затем – трассировку тех изменений, которые последуют в связи с первым. Этот пример показывает, почему более эффективно выполнять работу в обратной последовательности, т.е. сначала осуществить вторичные изменения. Если изменение значения поля city началось с выдачи новых значений (назначений) продавцам, то выполнение трассировки всех их покупателей окажется делом более сложным. Поскольку реальные базы данных существенно превосходят простые таблицы, которые рассматриваются здесь в качестве примера, при работе с ними могут возникнуть серьезные проблемы. Вам может оказаться полезен механизм ссылочной целостности SQL, но он не всегда применим и доступен.

Хотя нельзя сослаться в предложении FROM (подзапроса) на таблицу, из которой осуществляется удаление, в предикате можно ссылаться на текущую строку-кандидат из таблицы, т.е. на ту строку, которая в настоящее время проверяется в основном предикате. Другими словами, можно использовать связанные подзапросы. Они отличаются от применяемых с INSERT тем, что действительно базируются на строках-кандидатах из таблицы, на которую воздействует команда, а не на запросе для некоторой другой таблицы.

```
DELETE FROM Salespeople
   WHERE EXISTS
```

```
(SELECT *
    FROM Customers
    WHERE rating = 100
       AND Salespeople.snum = Customers.snum);
```

Часть AND предиката внутреннего запроса ссылается на таблицу Salespeople. Это означает, что целый подзапрос будет выполняться отдельно для каждой строки таблицы Salespeople, как и в случае других связанных подзапросов. Команда удаляет всех продавцов, имеющих по крайней мере одного покупателя с рейтингом 100, из таблицы Salespeople. Для достижения этого существуют и другие способы. Приведем один из них:

```
DELETE FROM Salespeople
   WHERE 100 IN
     (SELECT rating
        FROM Customers
        WHERE Salespeople.snum = Customers.snum);
```

Запрос находит все рейтинги покупателей каждого продавца и удаляет продавца, имеющего покупателя с рейтингом 100.

Можно применять также обычные связанные подзапросы, т.е. связанные с таблицей, на которую есть ссылка во внешнем запросе (а не в самом предложении DELETE). Например, можно найти наименьший заказ за каждый день и удалить продавца, которому такой заказ был адресован, с помощью следующей команды:

```
DELETE FROM Salespeople
  WHERE snum IN
    (SELECT snum
      FROM Orders a
      WHERE amt =
        (SELECT MIN (amt)
          FROM Orders b
          WHERE a.odate = b.odate));
```

Подзапрос в предикате DELETE использует связанный подзапрос. Этот внутренний запрос находит минимальный заказ на каждую дату для каждой строки внешнего запроса. Если его величина совпадает с величиной заказа текущей строки, предикат внешнего запроса принимает значение "истина". Это значит, что текущая строка содержит минимальную заявку на данную дату. Поле snum для продавца, ответственного за эту заявку, извлекается и подставляется в основной предикат самой команды DELETE, которая затем удаляет все строки с этим значением поля snum из таблицы Salespeople. (Поскольку snum – это первичный ключ таблицы Salespeople, найдется единственная строка, подлежащая удалению в соответствии с этим запросом. Однако, если окажется, что таких строк больше, все они будут удалены.) Таким

образом в данном случае будут удалены строки со следующими значениями snum: 1007 – минимум за 3 октября 1990 г.; 1002 – минимум за 4 октября 1990 г.; 1001 – минимальная и единственная заявка за 5 октября 1990 г. (команда кажется особенно жесткой, потому что удаляет Peel, как единственную заявку за 5 октября 1990 г., но она хорошо иллюстрирует суть дела).

Для того чтобы сохранить Peel, необходимо добавить другой подзапрос, как в следующем примере:

```
DELETE FROM Salespeople
   WHERE snum IN
      (SELECT snum
          FROM Orders a
          WHERE amt =
             (SELECT MIN (amt)
                FROM Orders b
                WHERE a.odate = b.odate)
          AND 1 <
              (SELECT COUNT (onum)
                 FROM Orders b
                 WHERE a.odate = b.odate));
```

В этом случае для дат, когда поступила только одна заявка, будет получено значение счетчика равное 1 во втором связанном подзапросе, что делает предикат внешнего запроса ложным, и, следовательно, эти значения поля snum не удовлетворяют основному предикату.

Использование подзапросов с *UPDATE*

UPDATE, как и DELETE, использует подзапросы внутри предиката. Можно применять связанные подзапросы в любой форме, приемлемой для DELETE: связанными либо с таблицей, которую следует модифицировать, либо с таблицей, на которую есть ссылка во внешнем запросе. Например, используя связанный подзапрос для таблицы, подлежащей обновлению, можно повысить комиссионные для всех продавцов, которые обслуживают, по крайней мере, двух покупателей:

```
UPDATE Salespeople
   SET comm = comm + 0.01
   WHERE 2 <=
      (SELECT COUNT (cnum)
         FROM Customers
         WHERE Customers.snum = Salespeople.snum);
```

Теперь для продавцов Peel и Serres, имеющих множество покупателей, будут увеличены комиссионные.

Приведем модификацию последнего примера для раздела, в котором рассматривалась команда DELETE. Здесь уменьшаются комиссионные для продавцов, получивших минимальные заказы:

```
UPDATE Salespeople
   SET comm = comm – 0.01
   WHERE snum IN
     (SELECT snum
        FROM Orders a
        WHERE amt =
           (SELECT MIN (amt)
              FROM Orders b
              WHERE a.odate = b.odate));
```

Ограничения подзапросов в командах *DML*

Невозможность сослаться на таблицу, изменяемую в любом подзапросе команды обновления, исключает целую категорию возможных трансформаций. Например, нельзя просто выполнить операцию удаления всех покупателей с рейтингом ниже среднего. Сначала надо выполнить запрос, получающий значение среднего рейтинга, а затем удалить все строки таблицы Customers, в которых рейтинг ниже этого значения:

Шаг 1.

```
SELECT AVG (rating)
   FROM Customers;
```

Выходными данными для этого запроса является значение 200.

Шаг 2.

```
DELETE
   FROM Customers
   WHERE rating < 200;
```

РЕЗЮМЕ

Теперь вам известны три команды, управляющие всем содержимым базы данных. Хотелось бы только пояснить несколько основных моментов, связанных с вводом и удалением значений из таблицы: когда эти команды следует выполнять данному пользователю для данной таблицы и когда сделанные изменения станут постоянными.

Итак, команда INSERT применяется для того, чтобы добавить строки в таблицу. Можно ввести имена значений для строк в предложении VALUES (оно определяет только одну добавляемую строку), либо значения можно получить из запроса (это означает, что любое количество строк может быть добавлено с помощью одной команды). Если используется запрос, то он не

может ссылаться на таблицу, в которую осуществляется вставка каким-либо из способов: либо с помощью предложения FROM, либо путем внешней ссылки (как это делается в связанных запросах). Это применимо к любым подзапросам внутри данного запроса. Запрос сохраняет свободу в применении связанных подзапросов или подзапросов, использующих имена таблиц в предложении FROM внешних запросов (это является общим случаем применения запросов).

DELETE и UPDATE используются для исключения строк из таблицы и изменения значений в них. Обе команды применяются ко всем строкам таблицы, но можно употребить и предикат для определения подмножества удаляемых или обновляемых строк. Этот предикат может содержать подзапросы, связанные с таблицей, из которой выполняется удаление или для которой выполняется обновление, с использованием внешней ссылки. Но эти подзапросы не могут ссылаться на модифицируемую таблицу, имя которой указано в предложении FROM.

Глава 17

СОЗДАНИЕ ТАБЛИЦ

В этой главе обсуждаются проблемы создания, изменения и удаления таблиц. Речь пойдет об определениях таблиц, а не о данных, в них хранящихся. Реальная потребность в выполнении этих операций может и не возникнуть, но следует на концептуальном уровне иметь представление об этих операциях, что повышает компетентность пользователя в области применения SQL и в плане понимания природы используемых таблиц. Это относится к разделу SQL, называемому языком определения данных (Data Definition Language – DDL), в котором происходит создание объектов SQL.

Обсуждается и другой тип объектов данных SQL – индексы, применяемые для более эффективной организации поиска и для того, чтобы убедиться, что значения отличаются друг от друга. Подобные операции выполняются невидимо (без участия пользователя), но при попытке ввести значения в таблицу они могут отвергаться (не восприниматься), так как не являются уникальными. Это означает, что хотя две строки могут иметь в поле одинаковое значение, поле все равно будет обладать своим уникальным индексом, в противном случае происходит нарушение ограничений целостности.

Команда *CREATE TABLE*

Таблицы определяются с помощью команды CREATE TABLE, создающей пустую таблицу – таблицу, не имеющую строк. Значения вводятся с помощью команды DML (языка манипулирования данными) INSERT (см. главу 15). Команда CREATE TABLE определяет имя таблицы и множество поименованных столбцов в указанном порядке. Для каждого столбца устанавливаются

тип и размер. Каждая таблица должна иметь, по крайней мере, один столбец. Синтаксис команды CREATE TABLE:

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

Типы данных существенно различаются в разных программных продуктах. Однако в цепях совместимости со стандартом, они, как минимум, поддерживают стандартные ANSI-типы.

Поскольку пробелы используются для разделения отдельных частей команд в SQL, их нельзя использовать как часть имени таблицы (либо как часть какого-либо другого объекта, например, индекса). Символ подчеркивания (_) наиболее часто используется для разделения слов в именах таблиц.

Значение аргумента размера зависит от типа данных. Если он не указывается, то система установит значение автоматически. Вероятно, это наиболее удачное решение для числовых значений, поскольку в данном случае все поля определенного типа имеют один и тот же размер, что позволяет впоследствии не заботиться о совместимости по объединению. Использование аргумента размера с некоторыми числовыми типами является непростой задачей. Для хранения больших чисел вы должны убедиться, что поле имеет достаточную длину.

Тип данных, для которого обязательно следует указывать размер, – это CHAR. В данном случае аргумент размера – целое число, задающее максимальное число символов, которые могут содержаться в поле. Реальное количество символов в поле может изменяться от нуля (если в поле содержится NULL-значение) до заданного максимального значения. По умолчанию это 1, т.е. в поле может содержаться единственный символ.

Следующая команда позволяет создать таблицу Salespeople:

```
CREATE TABLE Salespeople
   (snum     integer,
    sname    char( 10),
    city     char( 10),
    comm     decimal);
```

Порядок столбцов в определении таблицы существенен, он определяет порядок, в котором задаются значения элементов строк. Определения столбцов не должны задаваться в отдельных строках, но они должны разделяться запятыми.

Индексы

Индекс (index) – это упорядоченный (в алфавитном или числовом порядке) список содержимого столбцов или группы столбцов в таблице. Таблицы могут иметь большое количество строк и, поскольку строки задаются в любом произвольном порядке, поиск их по значению какого-либо из полей может

занять достаточно много времени. Индексы предназначены для решения этой проблемы и для объединения всех значений в группы из одной или нескольких строк, отличных друг от друга.

Когда создается индекс по значениям какого-либо поля для базы данных, создается упорядоченный список значений для этого поля. Предположим, таблица Customers имеет тысячи строк и нужно найти покупателя с номером 2999. Поскольку строки не упорядочены, программа должна просмотреть всю таблицу, строку за строкой, и выбрать ту, в которой значение поля cnum равно 2999. Если бы по полю cnum был организован индекс, программа могла бы сразу найти в нем значение 2999 и получить информацию о том, как обнаружить нужную строку таблицы. Это может значительно улучшить выполнение запросов, но управление индексами существенно замедляет время выполнения операций обновления (таких как INSERT и DELETE); кроме того, сам индекс занимает место в памяти. Следовательно, перед созданием индексов следует тщательно проанализировать ситуацию.

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

Синтаксис команды создания индекса обычно выглядит так:

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

Таблица должна быть уже создана и содержать столбцы, имена которых указаны в команде. Имя индекса, определенное в команде, должно быть уникальным в базе данных, то есть оно не может использоваться для каких-либо других целей любым пользователем базы данных. Будучи однажды созданным, индекс является невидимым для пользователя. SQL сам решит, когда есть смысл воспользоваться индексом, и сделает это автоматически. Например, если часто используется таблица Customers для поиска клиентов для конкретных продавцов по значениям поля snum, следует создать индекс по полю snum таблицы Customers.

```
CREATE INDEX Clientgroup ON Customers(snum);
```

Теперь вы можете быстро найти клиентов для продавцов.

Уникальные индексы

Для индекса в предыдущем примере уникальность не нужна. Данный продавец может иметь любое количество покупателей. Это становится неприемлемым, если ключевое слово UNIQUE используется перед ключевым словом INDEX. Поле cnum, как первичный ключ, может быть первым кандидатом на создание уникального индекса:

```
CREATE UNIQUE INDEX ID_Cust ON Customers(cnum);
```

Замечание: Эта команда будет отвергнута, если в поле cnum имеются одинаковые значения. Наилучший способ работы с индексами – немедленное их создание после создания таблицы и перед занесением в нее значений. Уникальный индекс, создаваемый для более чем одного поля, требует, чтобы комбинация значений во всех столбцах была уникальной.

Предыдущий пример помог выяснить, можно ли использовать поле cnum в качестве первичного ключа таблицы Customers. Для базы данных можно выполнить тщательное планирование первичного и других ключей.

Удаление индексов

Основная причина именования индексов состоит в их удалении время от времени. Обычно пользователи не знают о существовании индекса. SQL автоматически определяет его необходимость и создает его, если это нужно. При исключении индекса (вы обязательно должны знать его имя) используется такой синтаксис:

```
DROP INDEX <имя индекса>;
```

Удаление индекса не изменяет содержимого поля (полей).

Изменение таблицы, которая уже была создана

Команда ALTER TABLE (изменить таблицу), не являясь частью стандарта ANSI, широко применяется. Форма команды достаточно прозрачна, хотя ее возможности изменяются в широких границах. Обычно она осуществляет добавление столбцов в таблицу, иногда может удалять столбцы или изменять их размеры – осуществлять добавление и удаление ограничений. Обычный синтаксис команды, предназначенной для добавления столбца в таблицу выглядит следующим образом:

```
ALTER TABLE <имя таблицы> ADD <имя столбца>
    <тип данных> <размер>;
```

По этой команде для существующих в таблице строк добавляется столбец, в который заносится NULL-значение. Новый столбец становится последним столбцом в таблице. Допустимо добавление в нее нескольких столбцов с помощью одной команды; в этом случае их определения разделяются запятой. Можно исключать столбцы или изменять их описания. Часто изменение столбцов связано с изменением их размеров, добавлением или удалением ограничений. Система должна предоставить пользователю средства контроля, позволяющие удостовериться, что введенные данные, хранящиеся в таблице к моменту выполнения команды ALTER TABLE, удовлетворяют заданным в команде новым ограничениям. Для этого команда отвергается (выполнение команды завершается аварийно). Однако, наилучший вариант – возможность двойного контроля ситуации. Необходимо изучить соответствующие разделы документации по конкретной системе, прежде чем приступить к выполнению

этой операции. Из-за нестандартной природы команды ALTER TABLE следует постоянно обращаться к документации по конкретной системе, прежде чем приступать к внесению каких-либо изменений в таблицы.

ALTER TABLE становится неоценимой, когда возникает потребность переопределить таблицу, но база данных должна проектироваться так, чтобы, по возможности, избежать подобных ситуаций. Изменение структуры таблицы, используемой в настоящее время, дело рискованное. Представления таблицы, которые создаются на основе данных, хранящихся в реальных таблицах, могут не допустить выполнения этой команды; программы, использующие встроенный SQL, могут привести к ошибочной ситуации в процессе выполнения этой команды, либо могут отвергать эту команду. Кроме того, в процесс изменения таблицы могут оказаться вовлеченными все пользователи, имеющие дело с этой таблицей. По этой причине следует стараться проектировать таблицы с учетом перспективы их использования, а необходимость выполнения команды ALTER TABLE следует рассматривать как крайнюю меру.

Если система не поддерживает команду ALTER TABLE или если желательно избежать применения этой команды, можно создать новую таблицу с необходимыми изменениями в ее определении, а затем использовать команду INSERT с запросом SELECT * для передачи в новую таблицу существовавших ранее данных. Пользователи, имевшие доступ к старой таблице, автоматически наследуют право доступа к новой таблице.

Исключение таблицы

Необходимо быть владельцем (создателем) таблицы, чтобы иметь возможность ее удалить. Чтобы не причинить ущерба данным, хранящимся в базе данных, необходимо предварительно удалить все данные из таблицы, то есть сделать ее пустой, а затем уже исключить таблицу из базы данных. Таблица, имеющая строки, не может быть удалена. Синтаксис команды, осуществляющей удаление пустой таблицы (определения таблицы) из системы таков:

DROP TABLE <имя таблицы>;

После выполнения команды, имя таблицы больше не распознается как имя таблицы, команды не могут работать с объектом, имя которого было указано в команде DROP. Перед выполнением команды следует удостовериться, что эта таблица не содержит внешних ключей для какой-либо другой таблицы и что эти таблицы не используются для определения представлений.

Команда реально не является частью стандарта ANSI, но поддерживается и является полезной. Она весьма проста и не имеет различий в толковании (как команда ALTER TABLE). ANSI не оговаривает способа удаления или отказа от определений таблиц.

РЕЗЮМЕ

Эта глава вводит вас в курс определения данных. Вы можете теперь создавать, модифицировать и удалять таблицы. Поскольку только первая из перечисленных функций является частью официального SQL-стандарта, детали остальных команд существенно различаются для различных программных продуктов, особенно для команды ALTER TABLE. DROP TABLE позволяет избавиться от таблиц, потерявших свою актуальность. Она удаляет только пустые таблицы и, следовательно, не разрушает данные.

В этой главе дано общее описание индексов, процедуры их создания и удаления. SQL не предоставляет широких возможностей в плане управления процессом выполнения команд. Скорость выполнения различных команд определяется конкретной реализацией программного продукта. Индексы являются одним из средств воздействия на процесс выполнения команды в SQL.

Литература

1. Крис Фиайли (Chris Fehily) – SQL. Руководство по изучению языка. М., 2003
2. Филипп Андон, Валерий Резниченко – Язык запросов SQL. Питер, 2006
3. Боуман Дж.С., Эмерсон С.Л., Дарновски М. – Практическое руководство по SQL
4. Дж.Грофф – SQL.Полное руководство. BHV, 2001
5. Дунаев В.– Базы данных. Язык SQL для студента. СПб, 2006
6. Майкл Дж.Хернандес, Джон Л.Вьескас – SQL-запросы для простых смертных. Практическое руководство по манипулированию данными в SQL. М., 2003
7. Мартин Грубер – Понимание SQL. Изд-во ЛОРИ, 2014
8. Джеймс Р. Грофф, Пол Н. Вайнберг, Эндрю Дж. Опель – SQL. Полное руководство. Изд. Вильямс, 2015