# Реляционные базы данных

**Реляционная база данных** – база данных, основанная на реляционной модели данных. Слово «реляционный» происходит от англ. relation (отношение, зависимость, связь).

Основные свойства реляционных БД:

* В таблице не может быть двух одинаковых строк. В математике таблицы, обладающие таким свойством, называют отношениями.
* Столбцы располагаются в определенном порядке, который создается при организации таблицы. В таблице может не быть ни одной строки, но обязательно должен быть хотя бы один столбец.
* У каждого столбца есть свое уникальное имя (в пределах таблицы). При этом все значения в одном столбце имеют один тип (число, дата, текст).
* На пересечении каждого столбца и строки может находиться только атомарное значение (одно значение, не состоящее из группы значений). Таблицы, удовлетворяющие данному условию, называют нормализованными.

Рассмотрим пример. Предположим, мы захотели создать БД для форума. На форуме есть зарегистрированные пользователи, которые создают темы и оставляют сообщения в этих темах. Эта информация и должна храниться в БД.

Теоретически мы можем все это расположить в одной таблице, например:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя | E-mail | Пароль | Созданные темы | Созданные сообщения |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Но это противоречит свойству атомарности (одно значение в одной ячейке), а в столбцах Темы и Сообщения предполагается неограниченное число значений. Из этого следует, что таблицу необходимо разбить на три таблицы: Пользователи, Темы и Сообщения.

**Пользователи**

|  |  |  |
| --- | --- | --- |
| Имя | E-mail | Пароль |
|  |  |  |
|  |  |  |

**Темы**

|  |  |
| --- | --- |
| Наименование | Автор |
|  |  |
|  |  |

**Сообщения**

|  |  |
| --- | --- |
| Текст | Автор |
|  |  |
|  |  |

Таблица Пользователи удовлетворяет всем условиям. Чего не скажешь о таблицах Темы и Сообщения. Ведь в таблице не может быть двух одинаковых строк, а где гарантия, что один пользователь не оставит два одинаковых сообщения, например:

**Сообщения**

|  |  |
| --- | --- |
| Текст | Автор |
| Думаю, нужно сделать так… | Кирилл |
| Согласен | Вася |
| А еще можно сделать так… | Семен |
| Согласен | Вася |

Кроме того, мы знаем, что каждое сообщение обязательно относится к какой-либо теме. Для решения подобных проблем в реляционных БД существуют **ключи**.

Первичный ключ (primary key) – столбец, значения которого во всех строках различны. Первичные ключи могут быть логическими (естественными) и суррогатными (искусственными). Так, для таблицы Пользователи первичным ключом может стать столбец E-mail (пользователей с одинаковыми почтовыми ящиками не может быть априори). На практике лучше использовать суррогатные ключи, так как их применение позволяет абстрагировать ключи от реальных данных. Помимо этого, первичные ключи нельзя изменять, а пользователь может в любое время изменить E-mail.

Суррогатный ключ представляет собой дополнительное поле в базе данных. Как правило, это порядковый номер записи:

**Пользователи**

|  |  |  |  |
| --- | --- | --- | --- |
| ID пользователя | Имя | E-mail | Пароль |
| 1 | Кирилл | [kerya@mail.ru](mailto:kerya@mail.ru) | 1234 |
| 2 | Василий | [vasya@mail.ru](mailto:vasya@mail.ru) | 1352 |
| 3 | Семен | [senya@mail.ru](mailto:senya@mail.ru) | 6790 |

**Темы**

|  |  |  |
| --- | --- | --- |
| ID темы | Наименование | Автор |
| 1 | О рыбалке | Кирилл |
| 2 | Велосипеды | Василий |
| 3 | Ночные клубы | Семен |
| 4 | О рыбалке | Василий |

**Сообщения**

|  |  |  |
| --- | --- | --- |
| ID сообщения | Текст | Автор |
| 1 | Думаю, нужно сделать так… | Кирилл |
| 2 | Согласен | Вася |
| 3 | А еще можно сделать так… | Семен |
| 4 | Согласен | Вася |

Теперь каждая запись в наших таблицах уникальна. На осталось установить соответствие между темами и сообщениями в них. Делается это также при помощи первичных ключей.

**Сообщения**

|  |  |  |  |
| --- | --- | --- | --- |
| ID сообщения | Текст | Автор | ID темы |
| 1 | Думаю, нужно сделать так… | Кирилл | 1 |
| 2 | Согласен | Василий | 4 |
| 3 | А еще можно сделать так… | Семен | 1 |
| 4 | Согласен | Василий | 1 |

Теперь понятно, что сообщение с ID = 2 принадлежит теме «О рыбалке» (ID темы = 4), созданной Василием, а остальные сообщения принадлежат теме «О рыбалке» (ID темы = 1), созданной Кириллом. Такое поле называется **внешний ключ** (foreign key). Каждое значение этого поля соответствует какому-либо первичному ключу из таблицы Темы. Так устанавливается однозначное соответствие между сообщениями и темами, к которым они относятся.

Последний нюанс. Предположим, что у нас добавился новый пользователь, которого зовут тоже Василий:

**Пользователи**

|  |  |  |  |
| --- | --- | --- | --- |
| ID пользователя | Имя | E-mail | Пароль |
| 1 | Кирилл | [kerya@mail.ru](mailto:kerya@mail.ru) | 1234 |
| 2 | Василий | [vasya@mail.ru](mailto:vasya@mail.ru) | 1352 |
| 3 | Семен | [senya@mail.ru](mailto:senya@mail.ru) | 6790 |
| 4 | Василий | [waska@mail.ru](mailto:waska@mail.ru) | 7756 |

Как мы узнаем, какой именно Вася оставил сообщения? Для этого поля автор в таблицах Темы и Сообщения необходимо также сделать внешними ключами:

**Темы**

|  |  |  |
| --- | --- | --- |
| ID темы | Наименование | ID автора |
| 1 | О рыбалке | 1 |
| 2 | Велосипеды | 2 |
| 3 | Ночные клубы | 3 |
| 4 | О рыбалке | 1 |
| 5 | К кому обратиться | 4 |

**Сообщения**

|  |  |  |  |
| --- | --- | --- | --- |
| ID сообщения | Текст | ID автора | ID темы |
| 1 | Думаю, нужно сделать так… | 1 | 1 |
| 2 | Согласен | 2 | 4 |
| 3 | А еще можно сделать так… | 3 | 1 |
| 4 | Согласен | 2 | 1 |

Наша база данных готова.

Чтобы считаться реляционной, СУБД должна:

* Представлять всю информацию в виде таблиц.
* Поддерживать логическую структуру данных, независимо от их физического представления.
* Использовать язык высокого уровня (например, SQL) для структурирования, выполнения запросов, и изменения информации в БД.
* Поддерживать основные реляционные операции (выбор, проектирование и объединение), а также теоретико-множественные операции, такие как объединение, пересечение и дополнение.
* Поддерживать виртуальные таблицы, обеспечивая пользователям альтернативный способ просмотра данных в таблицах.
* Различать в таблицах неизвестные значения (nulls), нулевые значения и пропуски в данных.
* Обеспечивать механизмы для поддержки целостности, авторизации, транзакций и восстановления данных.

### Реляционная модель: одни таблицы

В реляционных БД таблицы состоят из горизонтальных **строк** (row) и вертикальных **столбцов** (column). Все данные представляются в табличном формате – другого способа просмотреть информацию в БД не существует.

Иногда могут встречаться такие понятия, как **отношение** (relation), **кортеж** (tuple) и **атрибут** (attribute). Это соответственно синонимы понятий таблица, строка и столбец, так же, как и **файл** (file), **запись** (record) и **поле** (field).

Каждая таблица состоит из строк и столбцов. Каждая строка описывает отдельный **объект** или **сущность** (entity) – человека, компанию, сделку и т.д. Каждый столбец описывает одну характеристику объекта – имя человека или его адрес, телефонный номер компании, дату и т.д.

Каждый элемент данных или **значение** (value) определяется пересечением строки и столбца таблицы. Чтобы найти требуемый элемент данных, необходимо знать имя содержащей его таблицы и значение его **первичного ключа** (primary key) или уникального идентификатора.

В реляционных БД существует два типа таблиц – **пользовательские** (user table) и **системные** (system table). Последние, известные также под названием **системные каталоги** (system catalog), содержат описание базы данных.

### Независимость

Реляционная модель обеспечивает независимость данных на двух уровнях – *физическом* и *логическом*. **Физическая независимость данных** (physical data independence) означает с точки зрения пользователя, что представление данных не зависит от способа их физического хранения. Как следствие, физическое перемещение данных никоим образом не может повлиять на логическую структуру БД и восприятие данных пользователем.

**Логическая независимость данных** (logical data independence) означает, что изменение взаимосвязей между таблицами, столбцами и строками не влияет на правильное функционирование программных приложений и текущих запросов.

### Язык высокого уровня

Весь диалог с БД должен вестись на едином языке. Такой язык называется SQL. Он используется для манипуляций с данными, определения данных и администрирования данных. Любая операция по выборке, модификации, определению или администрированию выполняется с помощью **оператора** (statement) или **команды** (command) SQL.

Имеется две разновидности операций по манипуляции с данными – **выборка данных** (data retrieval) и **модификация данных** (data modification). Выборка – это поиск необходимых данных, а модификация – добавление, удаление или изменение данных.

Операции по выборке (часто называемые **запросами** (queries)) наиболее эффективно извлекают нужную информацию и отображают ее. Во всех запросах SQL используется ключевое слово **SELECT**.

SQL: SELECT \* FROM PUBLISHERS

Символ **звездочки** (\*) заменяет **названия всех столбцов** в таблице. Операции по модификации выполняются соответственно с использованием ключевых слов **INSERT, DELETE** и **UPDATE.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| pub\_id | pub\_name | address | city | state |
| 0736 | New Age Books | 1 1st St | Boston | MA |
| 0877 | Binnet & Hardley | 2 2nd Ave. | Washington | DC |
| 1389 | Algodata Infosystems | 3 3rd Dr. | Maiami | FL |

С помощью следующей команды можно добавить строку в таблицу *publishers*.

SQL: INSERT INTO PUBLISHERS VALUES (‘0010’, ‘Pragmatics’, ‘4 4th Ln.’, ‘Chicago’, ‘IL’ )

Если снова просмотреть данные, воспользовавшись оператором SELECT, то в таблице отобразится новая строка.

SQL: SELECT \* FROM PUBLISHERS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| pub\_id | pub\_name | address | city | state |
| 0736 | New Age Books | 1 1st St | Boston | MA |
| 0877 | Binnet & Hardley | 2 2nd Ave. | Washington | DC |
| 1389 | Algodata Infosystems | 3 3rd Dr. | Maiami | FL |
| 0010 | Pragmatics | 4 4th Ln. | Chicago | IL |

Другие команды SQL предназначены для создания и удаления таблиц, индексов и других объектов. Следующая команда создает таблицу под названием *test* с двумя столбцами – *id* для целых чисел и *name* для хранения символьной информации (до 15 символов).

SQL: CREATE TABLE TEST (id int, name char (15))

При этом можно смело пользоваться оператором SELECT, несмотря на отсутствие в таблице *test* каких-либо данных.

SQL: SELECT \* FROM TEST

|  |  |
| --- | --- |
| id | name |

Последняя категория операторов SQL – операторы администрирования или **команды управления данными** (data control). Они позволяют координировать совместное использование БД и поддерживать ее в наиболее эффективном состоянии.

Хорошим примером административных команд является ключевое слово **GRANT**, позволяющее установить полномочия пользователей. В следующем примере пользователю с именем *karen* разрешается выбирать данные из таблицы *test*.

SQL: GRANT SELECT ON test TO karen

### Реляционные операции

В определении СУРБД упоминаются три операции по выборке данных: проектирование, выбор (иногда называемый **ограничением** (restriction)) и объединение, которые позволяют строго указать системе, какие данные необходимо отобразить. Операция проектирования выбирает столбцы, операция выбора – строки, а операция объединения собирает вместе данные из связанных таблиц. SQL можно рассматривать как **непроцедурный язык программирования** (nonprocedural language), так как он позволяет выразить то, что необходимо получить, не вдаваясь в детали самого процесса.

Все эти три операции записываются с использованием ключевого слова SELECT. В SQL ключевое слово SELECT используется не только для выбора данных, но и для выполнения операций проектирования и объединения.

Красота оператора SELECT заключается в его упрощенном синтаксисе.

SQL:

SELECT список\_выбора

FROM список\_таблиц

WHERE условия\_поиска

В следующих подразделах объясняется, как этот простой на вид оператор используется для выполнения трех реляционных операций.

#### Проектирование

Операция проектирования позволяет указать системе, какие столбцы таблицы вы хотите просмотреть. Например, если необходимо просмотреть все строки таблицы, содержащей информацию об издательствах, но возникла потребность в данных только об их названиях и идентификационных номерах, следует воспользоваться следующим оператором выборки:

SQL: SELECT pub\_id, pub\_name FROM publishers

|  |  |
| --- | --- |
| pub\_id | pub\_name |
| 0736 | New Age Books |
| 0877 | Binnet & Hardley |
| 1389 | Algodata Infosystems |
| 0010 | Pragmatics |

Результирующие таблицы иногда называют **производными таблицами** (derived table), чтобы отличать их от **базовых таблиц** (base table), содержащих исходные строки данных.

#### Выбор

Операция выбора позволяет получить из таблицы подмножества ее строк. Чтобы указать, какие строки необходимы, соответствующие условия нужно разместить в предложении WHERE. Например, если нужно получить информацию только об издательствах, расположенных во Флориде, нужно ввести следующий запрос:

SQL: SELECT \* FROM publishers WHERE state = “FL”

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1389 | Algodata Infosystems | 3 3rd Dr. | Maiami | FL |

Можно комбинировать в запросе операции проектирования и выбора, чтобы получить требуемую информацию.

#### Объединение

Операция объединения может работать одновременно с одной или несколькими таблицами, соединяя данные таким образом, что можно легко сопоставить или выделить определенную информацию в своей БД. Операция объединения обеспечивает SQL и реляционную модель необходимой мощностью и гибкостью. С ее помощью можно выявлять любую взаимосвязь, существующую между элементами данных, а не только связи, введенные при конструировании базы.

При «объединении» двух таблиц на период действия запроса они как бы становятся единой таблицей. Операция объединения соединяет данные, сравнивая значения в заданных столбцах и отражая результаты.

Предположим, необходимо узнать имена издателей всех книг в БД. Названия всех книг содержатся в таблице *titles*. Кроме того, в ней хранится и другая полезная информация о книгах, в том числе и идентификационный номер издателя. Однако в этой таблице невозможно найти имя издателя – оно хранится в таблице *publishers*.

|  |
| --- |
| *titles* |
| title\_id |
| title |
| type |
| pub\_id |
| price |
| advance |
| ytd\_sales |
| contract |
| notes |
| pubdate |

|  |
| --- |
| *publishers* |
| pub\_id |
| pub\_name |
| address |
| city |
| state |

Поставленная задача решается благодаря тому, что обе таблицы содержат идентификационные номера издателей. Чтобы получить одновременно имена издателей и названия книг можно объединить эти таблицы.

Система будет искать все случаи совпадения значений в столбцах pub\_id обеих таблиц. При каждом совпадении будет создаваться новая строка, содержащая значения из столбцов объединяемых таблиц. Ниже приведен код запроса:

SQL: SELECT title, pub\_name

FROM titles, publishers

WHERE publishers.pub\_id = titles.pub\_id

Столбец title в предложении SELECT относится к таблице *titles*, столбец pub\_name – таблице *publishers.* Операция проектирования позволяет в одном списке указывать столбцы из разных таблиц. В предложении FROM задаются две объединяемые таблицы. В предложении WHERE говорится, что будут объединяться строки этих таблиц, имеющие одинаковое значение идентификационного номера в столбце pub\_id. Ниже приводится полученный результат.

|  |  |
| --- | --- |
| title | pub\_name |
| Computer Fobic and Non-Fobic Individuals | New Age Books |
| Emotional Security | New Age Books |
| Prolonged Data Deprivation | New Age Books |
| The Psychology of Computer Cooking | Binnet & Hardley |
| Sushi, Anyone? | Binnet & Hardley |
| The Gourmet Microwave | Binnet & Hardley |
| Secrets of Silicon Valley | Algodata Infosystems |
| Net Etiquette | Algodata Infosystems |
| But Is It User Friendly | Algodata Infosystems |

### Альтернативный способ просмотра данных

**Курсор** (view) – это альтернативный способ просмотра данных из нескольких таблиц. Курсоры иногда называются **виртуальными таблицами** (virtual tables) или производными таблицами. Таблицы, на основе которых работают курсоры, называются базовыми таблицами. Курсор можно рассматривать как перемещаемую по таблицам рамку, через которую можно увидеть только необходимую часть информации. Курсор можно получить из одной или нескольких таблиц БД (включая и другие курсоры), используя любые операции выбора, проектирования и объединения. Курсоры позволяют создавать таблицы для специальных целей. С их помощью можно использовать результаты выполнения операторов выбора, проектирования и объединения как основу для последующих запросов.

Виртуальные таблицы, в отличие от базовых, физически не хранятся в БД. Подобно результатам операции выбора, курсоры напоминают обычные таблицы БД.

Для создания курсоров предназначен оператор **SQL CREATE VIEW**. Чтобы преобразовать результаты выполнения оператора SELECT в курсор, нужно поместить перед ним команду CREATE VIEW. Чтобы получить виртуальную таблицу на основе предыдущего примера, нужно воспользоваться оператором CREATE VIEW:

SQL CREATE VIEW Book\_and\_Pubs

AS

SELECT title, pub\_name

FROM titles, publishers

WHERE publishers.pub\_id = titles.pub\_id

Если применить операцию выбора к виртуальной таблице, можно увидеть результаты выполнения запроса, на основе которого она была создана.

В идеальной реляционной системе с курсорами можно оперировать как и с любыми другими таблицами. В реальных условиях различные версии SQL накладывают на курсоры определенные ограничения, в частности, на обновление. Одно из правил Кодда гласит, что в истинно реляционной системе над курсорами можно выполнять все "теоретически" возможные операции.

### Нули

Понятие нуля в БД используется для обработки пропущенной информации. "Нуль" не означает пустое поле или обычный математический нуль. Он отображает тот факт, что значение неизвестно, недоступно или неприменимо. Существенно, что использование нулей инициирует переход с двухзначной логики (да/нет или что-то/ничего) на трехзначную (да/нет/может быть или что-то/ничего/не уверен).

### Безопасность

Понятие безопасности связано с необходимостью управления доступом к информации. Команды SQL GRANT и REVOKE позволяют некоторым привилегированным пользователям устанавливать права других пользователей на просмотр и модификацию информации в БД. В большинстве реализаций SQL **правами на доступ и модификацию данных** (permission) можно управлять на уровне таблиц и столбцов. Эти права устанавливают **владельцы** (owner) БД или объектов БД. Владельцем БД является создавший ее пользователь с помощью команды SQL CREATE. Некоторые системы разрешают передавать права владения от создателя БД другому пользователю.

В многопользовательских системах обычно имеется пользователь с правами даже более высокими, чем у владельца БД – **системный администратор** (system administrator) или **администратор БД** (database administrator).

### Целостность

**Целостность** (integrity) – очень сложный и серьезный вопрос при управлении реляционными базами данных. Несогласованность или противоречивость данных может возникать вследствие сбоя системы – проблемы с аппаратным обеспечением, ошибки в ПО или логические ошибки в приложениях. РСУРБД защищают данные от такого типа несогласованности, гарантируя, что команда SQL либо будет исполнена до конца, либо будет полностью отменена. Этот процесс обычно называют **управлением транзакциями** (transaction management).

Второй тип целостности, называемый **объектной целостностью** (entity integrity), связан с корректным проектированием БД. Объектная целостность требует, чтобы ни один первичный ключ не имел нулевого значения.

Третий тип целостности, называемый **ссылочной целостностью** (referential integrity), означает непротиворечивость между частями информации, повторяющимися в разных таблицах. Например, при изменении неправильно введенного номера карточки социального страхования в одной таблице, другие таблицы, содержащие эту же информацию, продолжают ссылаться на старый номер, поэтому требуется обновление и этих таблиц. *Чрезвычайно важно, чтобы при изменении информации в одном месте, она соответственно изменялась и во всех других местах.*

Правила Кодда гласят, что СУРБД должны обеспечивать не только объектную и ссылочную целостность, но и позволять "вводить дополнительные ограничения на целостность, отражающие специальные требования". Кроме того, по определению Кодда, ограничения на целостность должны:

* определяться на языке высокого уровня, используемом системой для всех других целей;
* храниться в словаре данных, а не в программных приложениях.

# Проектирование баз данных

## Структура БД

Процесс, в ходе которого решается, какой вид будет у вновь создаваемой БД, называется **проектированием БД** (database design). Работа по проектированию БД включает выбор:

* таблиц, которые будут входить в БД,
* столбцов, принадлежащих каждой таблице,
* взаимосвязей между таблицами и столбцами.

Конструирование БД на основе реляционной модели имеет ряд важных преимуществ:

* независимость логической структуры от физического и пользовательского представления,
* гибкость структуры БД – конструктивные решения не ограничивают возможности проектировщика выполнить в будущем самые разнообразные запросы.

**Декомпозиция без потерь** (non-loss decomposition) – это разделение таблицы на несколько меньших таблиц без потери информации с целью нормализации.

Примерная последовательность шагов, которые рекомендуется выполнить в процессе проектирования базы данных:

1. Исследуйте информационную среду, которую собираетесь моделировать. Откуда поступает информация и в каком виде? Как она будет вводиться в систему и кто этим будет заниматься? Как часто она изменяется? Какие параметры системы будут наиболее критическими с точки зрения времени реакции на запрос и надежности? Также необходимо уточнить, в каком виде информация должна извлекаться из БД – в форме отчетов, заказов, статистической информации – и кому она будет предназначаться.
2. Создайте список объектов вместе с их свойствами и атрибутами. Объекты, скорее всего, имеет смысл собрать в таблицы (каждая строка таблицы будет описывать один объект, например человека, компанию или книгу), свойства объектов будут представлены столбцами таблицы (например, зарплата, адрес компании, стоимость книги). Конечно, сначала можно создать список всех возможных атрибутов, а затем сгруппировать их в объекты, а не начинать с самих объектов. Какой метод выбрать – это, скорее, дело вкуса. В любом случае необходимы ответы на следующие вопросы. Действительно ли выбранные атрибуты подходят для описания данного объекта или лучше использовать их с другим объектом? Нужны ли еще дополнительные объекты или другие атрибуты?
3. В ходе работы следует обязательно записывать все свои конструктивные решения на бумаге или при помощи текстового редактора. Разработчики БД обычно начинают с чистого листа, постепенно создавая макет таблиц и связей между ними, называемый **структурой данных** (data structure) или **диаграммой зависимостей между объектами** (E-R diagram).
4. Предварительно разобравшись с объектами и их атрибутами, нужно убедиться, что каждый объект имеет атрибут (или группу атрибутов), по которому однозначно можно идентифицировать любую строку в будущей таблице. Этот идентификатор обычно называется первичным ключом. Если такового нет, то для получения искусственного ключа необходимо создать дополнительный столбец.
5. Затем рассмотрите зависимости между объектами. Имеются ли зависимости типа один-ко-многим (один издатель может иметь множество изданных книг, но каждая книга может быть выпущена только одним издателем) или многий-ко-многим (каждый автор может написать несколько книг, и каждая книга может иметь несколько авторов)? Есть ли возможности для объединения связанных таблиц? Для этого служат **внешние ключи** (foreign key), столбцы в связанных таблицах с совпадающими значениями первичных ключей.
6. После создания чернового варианта структуры БД необходимо проанализировать ее с точки зрения правил нормализации для поиска логических ошибок. Исправьте все отклонения от нормальных форм или обоснуйте свое решение отказаться от выполнения ряда правил нормализации в интересах простоты освоения или производительности. Задокументируйте причины таких решений.
7. Теперь можно создавать БД и помещать в нее некоторые прототипы данных, используя для этого команды SQL. Обязательно необходимо провести эксперименты с запросами, изучая получаемые результаты. Возможно выполнение ряда тестов на производительность с целью проверки разных технических решений.
8. Оценить свою разработку на удовлетворенность полученными результатами.

### Что такое "хорошая структура"

Хорошая структура:

* максимально упрощает взаимодействие с БД,
* гарантирует непротиворечивость данных,
* "выжимает" максимум производительности из системы.

Выбор подходящего числа столбцов обычно является компромиссом между простотой понимания БД и правилами нормализации.

Хорошо разработанная БД предотвращает ввод противоречивой информации и случайное удаление данных. Это достигается за счет минимизации ненужного дублирования данных в таблицах и поддержки целостности.

Плохая структура БД:

* приводит к непониманию результатов выполнения запросов,
* повышает риск введения в БД противоречивой информации,
* порождает избыточные данные,
* усложняет выполнение изменений структуры созданных ранее и уже заполненных данными таблиц.

Не существует идеального решения, полностью удовлетворяющего все требования, предъявляемые при проектировании БД.

### Описание используемой в качестве примера БД

Нужно обратить внимание на то, что БД под названием bookbiz является исключительно учебной БД. Ее основное назначение – обеспечение разработчика небольшим набором интересных данных, манипулируя которыми можно изучить синтаксис и семантику языка SQL.

БД bookbiz содержит информацию о вымышленной компании, занимающейся издательской деятельностью и имеющей три дочерних издательства. В ней представлены данные, которые могут потребоваться редакторам, менеджерам, и другим сотрудникам компании – информация о книгах, их авторах, редакторах, о финансовом состоянии компании. На их основе можно получать самые разнообразные отчеты, например, о текущих продажах, сравнивать книги разных издательств, узнать, какие редакторы работали с какими авторами и т.д.

Пользователи БД bookbiz могут задавать самые разные запросы:

* Кто из авторов проживает в Калифорнии?
* Какие книги стоят больше 9,95 доллара?
* Кто написал самое большее количество книг?
* Чем мы обязаны автору книги Life Without Fear?
* Какова средняя стоимость книг по психологии?
* Как повлияет на авторский гонорар увеличение на 10% стоимости книг по кулинарии?
* Как продаются книги по компьютерам?

Важной областью исследований являются бизнес-правила и политика организации, которые могут оказывать влияние на данные. БД boobkiz учитывает следующие особенности:

* автор может написать несколько книг;
* книга может быть написана несколькими авторами;
* порядок фамилий авторов на первой странице является критической информацией, так как влияет на получаемый ими гонорар;
* редактор может работать над несколькими книгами, и в каждой книге может быть несколько редакторов;
* в заказе на покупку может быть перечислено несколько книг.

## Данные и взаимосвязи

Начнем рассмотрение структуры БД bookbiz с построения простой модели взаимосвязи объектов. В самых общих чертах такое моделирование (иногда называемое объектным моделированием) подразумевает определение:

* объектов, информация о которых будет содержаться в БД;
* свойств этих объектов;
* взаимосвязей между ними.

### Объекты

Сначала рассмотрим объекты, на основе которых будет построена БД bookbiz. Без учета финансовой информации список объектов будет выглядеть так:

* авторы, книги которых опубликованы компанией;
* сами книги;
* редакторы, работающие в компании;
* издательства, которыми владеет компания.

Каждый пункт в этом списке описывает объект, существующий независимо от других объектов в мире БД bookbiz. Каждый такой объект представляется отдельной таблицей.

Каждый из таких объектов обладает собственными свойствами, которые также записаны в БД. Среди них:

*название книги*

*стоимость книги*

*дата выхода книги*

*имя автора*

*адрес автора*

*номер телефона автора*

*имя редактора*

*адрес редактора*

*номер телефона редактора*

*название издательства*

*адрес издательства*

Каждый пункт этого списка описывает отдельное свойство или атрибут рассматриваемого объекта (автор, книга, редактор или издательство) и является потенциальным столбцом в БД. Названия столбцов должны быть предельно ясными (назначение столбца должно быть понятно из его названия) и краткими (чтобы упростить ввод их названий и уменьшить их ширину). Создание списка объектов и их свойств должно помочь решить, какие таблицы и столбцы нужно включить в БД. В итоге можно получить, например, следующий макет базы:

Таблица titles

|  |  |  |
| --- | --- | --- |
| name | price | pubdate |

Таблица authors

|  |  |  |
| --- | --- | --- |
| name | address | phone |

Таблица editors

|  |  |  |
| --- | --- | --- |
| Name | address | phone |

Таблица publishers

|  |  |
| --- | --- |
| name | address |

Другой способ представления информации используется в диаграммах взаимосвязей между объектами. Обычно каждая таблица представляется на такой диаграмме в виде прямоугольника, содержащего в себе, кроме того, названия полей.

Этот макет из четырех таблиц, каждая из которых имеет по несколько столбцов, является первым шагом на пути построения реальной БД. Вы можете представить себе, что каждая таблица будет содержать множество строк данных. Каждая строка в таблице описывает **вхождение** (occurrence) или **экземпляр** (instance) объекта – отдельную книгу, автора, редактора или издательство.

Одна из задач проектирования БД состоит в обеспечении способа идентификации различных объектов, другими словами, система должна уметь отличать друг от друга отдельные строки таблицы. Как уже отмечалось, строки отличаются друг от друга по значению первичного ключа таблицы. Неформально, первичный ключ – это столбец или набор столбцов, однозначно определяющий строку.

|  |
| --- |
| *authors*  name  address  phone |

|  |
| --- |
| *titles*  name  price  pubdate |

|  |
| --- |
| *publishers*  name  address |

|  |
| --- |
| *editors*  name  address  phone |

**Первичные ключи.** Что будет первичными ключами для каждой из этих таблиц? Рассмотрим таблицу *autors*. Среди ее столбцов очевидным кандидатом на первичный ключ является *name.* Авторов всегда можно различать по имени. Однако использовать этот столбец в качестве первичного ключа достаточно проблематично по нескольким причинам. Во-первых, значения в столбце *name* состоят из имени и фамилии автора. Объединение в одном столбце имени и фамилии обычно является не слишком дальновидным поступком хотя бы потому, что в некоторых системах бывает трудно (а иногда просто невозможно) выполнить алфавитную сортировку по фамилиям. Таким образом, первым необходимым изменением будет разбиение столбца *name* на два отдельных столбца (как в таблице *authors*, так и в таблице *editors*).

Теперь, возвращаясь к вопросу о выборе первичного ключа в таблице *authors*, для этих целей можно использовать комбинацию столбцов *au­\_lname* и *au\_fname*. Эта комбинация будет удовлетворять нас, пока таблица не разрастется до такой степени, что в ней появятся авторы с одинаковыми именами и фамилиями. Например, как только в таблице появятся две Mary Smiths, комбинация полей *au\_lname* и *au\_fname* перестанет однозначно идентифицировать каждого автора.

|  |
| --- |
| *authors*  au\_lname  au\_fname  address  phone |

|  |
| --- |
| *titles*  name  price  pubdate |

|  |
| --- |
| *publishers*  name  address |

|  |
| --- |
| *editors*  ed\_lname  ed\_fname  address  phone |

Другая проблема, возникающая при использовании имен в качестве уникальных идентификаторов, связана с их частым неправильным вводом. Та же проблема возникает и с названиями компаний. Поэтому хорошим решением обычно является создание специального столбца, который и будет использоваться в качестве первичного ключа. Реальными примерами таких уникальных идентификаторов могут быть код предприятия, номер лицензии, заказа, студенческого билета и т.д.

В таблицах *authors* и *editors* мы использовали для этих целей номера карточек социального страхования. В таблицах *titles* и *publishers* используются произвольные идентификационные номера. Чтобы показать, какие столбцы являются первичными ключами, мы подчеркиваем их.

|  |
| --- |
| *authors*  **au\_id**  au\_lname  au\_fname  address  phone |

|  |
| --- |
| *titles*  **title\_id**  name  price  pubdate |

|  |
| --- |
| *publishers*  **pub­\_id**  name  address |

|  |
| --- |
| *editors*  **ed\_id**  ed\_lname  ed\_fname  address  phone |

### Отношение один-ко-многим

К этому моменту мы имеем готовые структуры четырех таблиц БД данных *bookbiz: authors, titles, editors, publishers*. Для каждой таблицы описаны атрибуты и определены первичные ключи.

Тем не менее, пока не определены никакие зависимости между данными. Пока остается непонятным, как, например, связаны между собой книги и конкретные издательства.

Связь между книгами и издательствами может описываться отношением один-ко-многим: каждая книга имеет единственного издателя, в то время как издатель может выпустить несколько книг. Отношение один-ко-многим часто записывается в виде 1:N.

Как реализовать это отношение? Первым порывом может быть желание добавить столбец. Столбец *title\_id* будет являться внешним ключом в таблице *publishers.* Можно использовать его для ссылки на определенные строки в таблице *titles* и объединения информации о книгах и издателях. К сожалению, это решение неправильное. Вспомним моделируемое отношение – один издатель, много книг – и посмотри, что произойдет при выходе новой книги. Разработчик добавит новую строку в таблицу *titles* с названием книги, ценой и т.д.

|  |  |  |  |
| --- | --- | --- | --- |
| title\_id | title | price | date |
| BU2075 | You Can Combat Computer Stress! | 2.99 | 6/30/85 |

|  |
| --- |
| *authors*  au\_id  au\_lname  au\_fname  address  phone |

|  |
| --- |
| *titles*  title\_id  name  price  pubdate |

|  |
| --- |
| *publishers*  pub­\_id  name  address  **title\_id** |

|  |
| --- |
| *editors*  ed\_id  ed\_lname  ed\_fname  address  phone |

Для каждой новой строки таблицы *titles* необходимо добавить соответствующую строку в таблицу *publishers*. При этом в строке таблицы *publishers* будет повторена уже существующая там информация и добавлен один столбец с новой информацией (*title\_id*), отражающей появление новой книги в таблице *titles*.

|  |  |  |  |
| --- | --- | --- | --- |
| pub\_id | pub\_name | address | title\_id |
| 0736 | New Age Books | 1 1st St Boston MA | BU2075 |

Вспомните, что одной из целей процесса проектирования БД является контроль за избыточностью данных, так как избыточность данных увеличивает вероятность появления ошибок. Лучшим решением является добавление внешнего ключа к таблице *titles*.

|  |
| --- |
| *authors*  au\_id  au\_lname  au\_fname  address  phone |

|  |
| --- |
| *titles*  title\_id  name  price  pubdate  **pub\_id** |

|  |
| --- |
| *publishers*  pub­\_id  name  address |

|  |
| --- |
| *editors*  ed\_id  ed\_lname  ed\_fname  address  phone |

Теперь при выходе новой книги добавляется строка в таблицу *titles* (со столбцом *pub\_id*) и ничего не изменяется в таблице *publishers*, пока компания не откроет новый филиал.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| title\_id | title | price | date | pub\_id |
| BU2075 | You Can Combat Computer Stress! | 2.99 | 6/30/85 | 0736 |

С учетом последних изменений получались следующая структура:

* таблица *publishers* содержит по одной строке для каждого издателя;
* таблица *titles* имеет по одной строке на каждую книгу;
* идентификационный номер издателя повторяется в таблице *titles*, так как издатель может выпустить несколько книг, и при этом обеспечивается минимальная избыточность данных.

Можно использовать логическую связь между столбцами *pub\_id* в таблицах *titles* и *publishers* для объединения этих таблиц. Иными словами, структура БД разрабатывалась в расчете на применение пользователями оператора объединения для выборки информации из таблиц *titles* и *publishers* с помощью одного запроса.

В таблице *publishers* столбец *pub­\_id* является первичным ключом. В таблице *titles* этот столбец выполняет роль внешнего ключа. Таким образом, реляционная модель требует, чтобы отношение один-ко-многим реализовывалось посредством пары первичный ключ-внешний ключ.

**Внешние ключи**. Неформально, внешний ключ – это столбец или комбинация столбцов в одной таблице, значения которого совпадают со значениями первичного ключа в другой таблице.

При проектировании БД обязательно нужно обеспечить непротиворечивость между первичными и внешними ключами (ссылочную целостность). Например:

* Если в таблице изменяется или удаляется идентификационный номер издателя, система должна автоматически выполнять модификации в таблице *titles*, то есть либо изменять соответствующие значения в столбце *pub­\_id* таблицы *titles*, либо удалять из нее строки с уже несуществующими в таблице значениями *pub\_id*.
* Если в БД добавляется информация по новой книге, система должна проверить корректность введенного значения в столбце *pub\_id* (то есть, что оно присутствует в таблице *publishers*).

### Отношение многий-ко-многим

Некоторые книги написаны несколькими авторами, к тому же некоторые авторы выпустили более одной книги. Другими словами, авторы и книги связаны отношением многий-ко-многим (часто записывается в виде N:N и иногда называется **соединением** (association)). В соответствии с реляционной теорией, соединения должны представляться отдельными таблицами, то есть в БД *bookbiz* должна быть таблица для авторов, таблица для книг и таблица, описывающая связи между ними.

Таблица *titleauthors* описывает отношение типа многий-ко-многим между авторами и книгами. Это такая же базовая таблица, как *titles* и *authors*, но вместо объектов она описывает связи. Если пользователю БД *bookbiz* потребуется информация о том, кто написал какую книгу, он или она создаст запрос на объединение, в котором для связи между таблицами *titles* и *authors* будет использована таблица *titleauthors.*

Таблицы *titleauthors* и *titles* объединяются по столбцам *title\_id, titleauthors* и *authors* по столбцам *au\_id*. Другими словами, *title\_id* в таблице *titleauthors* является внешним ключом, соответствующим первичному ключу *title\_id* в таблице *titles*, а *au\_id* в таблице *titleauthors* является внешним ключом, соответствующим первичному ключу *au\_id* в таблице *authors*. По общему принципу, сформулированному К. Дж. Дейтом, "в реляционной модели участники соединения определяются внешними ключами, образующими таблицу, представляющую это соединение".

|  |
| --- |
| *authors*  au\_id  au\_lname  au\_fname  address  phone |

|  |
| --- |
| *titles*  title\_id  name  price  pubdate  pub\_id |

|  |
| --- |
| *publishers*  pub­\_id  name  address |

|  |
| --- |
| *editors*  ed\_id  ed\_lname  ed\_fname  address  phone |

|  |
| --- |
| *titleauthors*  title\_id  au\_id |

Идентификатор автора и идентификатор книги не идентифицируют однозначно строки в таблице *titleauthors*. В ее столбцах повторяются идентификаторы книг, имеющих больше одного автора, и идентификаторы авторов, написавших несколько книг. Тем не менее, уникальной является комбинация идентификатор автора-идентификатор книги. Как говорит Дейт: "Для данного соединения, как правило, комбинация всех внешних ключей его участников имеет свойство уникальности". Таким образом, первичным ключом в таблице *titleauthors* является комбинация *title\_id* и *au\_id.*

Таблицы *editors* и *titles* имеют аналогичные взаимосвязи. Редактор может работать более чем над одной книгой, а у книги может быть несколько редакторов.

### Отношение один-к-одному

Бросим последний взгляд на объекты. В случае обнаружения между таблицами отношения типа 1:1 рекомендуется объединение их в одну таблицу. Основной причиной использования отношения 1:1 является увеличение скорости исполнения запросов. К примеру, при редком использовании какой-либо информации о книге (заметок об авторских правах или списка измененных страниц), ее можно поместить в отдельную таблицу, чтобы не обрабатывать ее в общих запросах. До изучения особенностей данных в процессе проектирования БД лучше избегать использования отношений типа один-к-одному.

### Последние замечания к объектному подходу

Основные действия, которые необходимо выполнить в процессе проектирования БД:

1. Представить каждый независимый объект (книга, автор, издатель, редактор, наниматель, студент, компания и т.д.) в виде строк базовых таблиц.
2. Представить свойства объектов (адреса авторов, стоимость книг и т.п.) в виде столбцов таблиц.
3. Убедиться, что в каждой таблице имеется первичный ключ. Этим ключом может быть уже существующий столбец или искусственно добавленный (например, порядковый номер, или номер карточки соцстраховки), или же комбинация двух и более столбцов.
4. Найти все отношения типа один-ко-многим между таблицами. Проверить соответствия между внешними и первичными ключами. Установить ограничения на целостность, связанные с каждым внешним ключом.
5. Представить каждое отношение типа многий-ко-многим (соединение) в виде "соединительной" таблицы, состоящей из внешних ключей, связанных с таблицами объектов. Первичным ключом для соединительной таблицы обычно является комбинация, составленная из этих внешних ключей.

Такой взгляд на БД *boobkiz* выявляет ряд ограничений. В ней не предусмотрена возможность записи информации о позиции автора в списке авторов книги и принципах разделения гонораров. Кроме того, в ней отсутствуют описания контрактов. Также в структуре базы не учтены вопросы, связанные с продажами книг.

## Руководство по нормализации

Руководство по нормализации – это набор стандартов проектирования данных, называемых **нормальными формами** (normal form). Нормальные формы изменяются в порядке от первой до пятой. Каждая последующая форма удовлетворяет требования предыдущей. Если следовать первому правилу нормализации, данные будут представлены в первой нормальной форме. Если данные удовлетворяют третьему правилу нормализации, они будут находиться в третьей нормальной форме (а также в первой и второй формах).

Выполнение правил нормализации обычно приводит к разделению таблиц на две или больше таблиц с меньшим числом столбцов, выделению отношений первичный ключ-внешний ключ в меньшие таблицы, которые снова могут быть соединены с помощью операции объединения.

Одним из основных результатов разделения таблиц в соответствии с правилами нормализации является уменьшение избыточности данных в таблицах. При этом в смущение не должно приводить наличие в базе одинаковых столбцов первичных и внешних ключей. На самом деле поддержка непротиворечивости между первичными и внешними ключами связана с понятием целостности данных.

### Первая нормальная форма

Первая нормальная форма требует, чтобы на любом пересечении строки и столбца находилось единственное значение, которое должно быть атомарным. Кроме того, в таблице, удовлетворяющей первой нормальной форме, не должно быть повторяющихся групп.

Рассмотрим на примере новой таблицы *sales*, как составить накладную сразу на несколько книг.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Bookbiz Sales Order From Oredr #14 Store 7131 Date: 5/29/87 | | | | |
| Item# | Title# | Ordered# | Shipped | Ship date |
| 1. | PS1372 | 20 | 20 | May 29 1987 |
| 2. | PS2106 | 25 | 25 | Apr 29 1987 |
| 3. | PS3333 | 15 | 10 | May 29 1987 |
| 4. | PS7777 | 25 | 25 | Jun 13 1987 |

Нельзя записать несколько идентификаторов книг в одном поле, так как это противоречило бы первой нормальной форме. Так как в первой нормальной форме все данные должны быть представлены в виде прямоугольных таблиц, также не допускается и использование множественных значений.

Чтобы избежать повторения столбцов, необходимо использовать **главную таблицу** (master table) – *sales* и **вспомогательную таблицу** (detail table) – *salesdetail*, которая поддерживает информацию по отдельным позициям накладной. Нужно обратить внимание, что объектное моделирование приводит к тем же результатам, так как в этом случае имеется отношение один-ко-многим (одна накладная – много позиций).

Занимаясь поиском повторяющихся полей, нужно разбить на компоненты все составные столбцы: например, столбец *address* нужно разбить на два столбца – *city* и *state*.

|  |  |  |  |
| --- | --- | --- | --- |
| *titleauthors*  title\_id  au\_id  contract | *titles*  title\_id  name  price  advance  pubdate  pub\_id | *publishers*  pub\_id  name  address | *editors*  ed\_id  ed\_lname  ed\_fname  address  phone |
| *tauthors*  au\_id  au\_lname  au\_fname  address  phone  au\_ord  royalty\_per | *titles*  sonum  stor\_id  date  qty\_ordered  qty\_shipped  title\_id  date\_shipper | *salesdetails*  **sonum**  **title\_id** | *titleditors*  title\_id  ed\_id |

В этом случае контракт определяется книгой, а не отдельными авторами, поэтому необходимо переместить столбец *contract* в таблицу *titles*. Этот пример иллюстрирует сложность процесса проектирования БД: решения часто могут зависеть от конкретных принципов организации работы в компании.

### Вторая нормальная форма

Вторая нормальная форма требует, чтобы ни один неключевой столбец не зависел только от части первичного ключа. Это правило относится к случаю, когда первичный ключ образован из нескольких столбцов, и неприменимо, когда первичный ключ образован только из одного столбца.

|  |  |  |  |
| --- | --- | --- | --- |
| *titleauthors*  title\_id  au\_id | *titles*  title\_id  name  price  advance  pubdate  pub\_id  **contract** | *publishers*  pub\_id  name  address  city  state | *editors*  ed\_id  ed\_lname  ed\_fname  address  city  state  zip  phone |
| *tauthors*  au\_id  au\_lname  au\_fname  address  city  state  zip  phone  au\_ord  royaltyper | *titles*  sonum  stor\_id  date  qty\_ordered  qty\_shipped  date\_shipped | *salesdetails*  sonum  title\_id | *titleditors*  title\_id  ed\_id |

### Третья нормальная форма

Третья нормальная форма повышает требования второй нормальной формы: она не ограничивается составными первичными ключами. Третья нормальная форма требует, чтобы ни один не ключевой столбец не зависел от другого не ключевого столбца. Любой не ключевой столбец должен зависеть только от столбца первичного ключа.

В таблице *authors* первичным ключом является *au\_id*. Проверяя каждый столбец, можно обнаружить, что столбец *au\_ord* (позиция автора в списке авторов книги) не зависит от столбца авторов (*au\_id*), так как у каждого автора может быть несколько книг, и в каждой из них он может занимать разные позиции (первую, вторую и третью). На самом деле позиция определяется отношением автор-книга. Аналогичным образом обстоят дела и со столбцом *royaltyper*. Оба этих столбца переносятся в таблицу *titleauthors*.

Столбцы *qty\_ordered* и *qty\_shipped* из таблицы *sales* также иллюстрируют этот принцип. Они относятся к отдельным пунктам, а не ко всей накладной, и, следовательно, должны быть перенесены в таблицу *salesdetails*.

Столбец *date\_shipped* более загадочен.

* Если заказ выполняется только при наличии всех книг, столбец *date\_shipped* относится ко всей накладной и поэтому должен быть перемещен в таблицу *sales*.
* Если же заказ выполняется по мере выполнения отдельных пунктов, этот столбец должен принадлежать таблице *salesdetails*.

Так как книги могут быть доступны только после выхода из печати, необходимо придерживаться второй модели.

Рассматривая структуру этих таблиц, можно обнаружить, что они удовлетворяют как второй, так и третьей нормальной форме. Они удовлетворяют второй нормальной форме, так как все не ключевые столбцы зависят от всего первичного ключа, и третьей нормальной форме, так как все не ключевые столбцы не зависят друг от друга. Другими словами, любой не ключевой столбец зависит от ключа, всего ключа и ничего, кроме ключа.

|  |  |  |  |
| --- | --- | --- | --- |
| *titleauthors*  title\_id  au\_id  **au\_ord**  **royaltyper** | *titles*  title\_id  name  price  advance  pubdate  pub\_id  contract | *publishers*  pub\_id  name  address  city  state | *editors*  ed\_id  ed\_lname  ed\_fname  address  city  state  zip  phone |
| *tauthors*  au\_id  au\_lname  au\_fname  address  city  state  zip  phone | *titles*  sonum  stor\_id  date | *salesdetails*  sonum  title\_id  **qty\_ordered**  **qty\_shipped**  **date\_shipped** | *titleditors*  title\_id  ed\_id |

### Четвертая и пятая нормальные формы

Четвертая нормальная форма запрещает независимые отношения типа один-ко-многим между ключевыми и не ключевыми столбцами. В качестве примера рассмотрим несколько придуманную ситуацию: каждый автор может иметь несколько машин и несколько домашних животных, но между машинами и животными нет абсолютно никакой связи, хотя они естественным образом связаны с автором.

|  |  |  |
| --- | --- | --- |
| *au\_lname* | car | pet |
| Ringer | 1987 Chevy Nova | Rover |
| Ringer | 1994 Volvo Station Wagon |  |
| Bennet | 1990 VW Rabbit | Spot |
| Green |  | Valiant |
| Green | 1989 Toyota Corolla | Fluffy |
| Green |  | Sam |

Помещение этой разнородной информации в одну таблицу может привести к появлению в ней пустых мест, так как домашних животных может быть больше, чем машин (как в случае с Green) или наоборот (как в случае с Ringer). Удаление данных о машинах или животных также может привести к появлению пустых мест.

Проблема здесь состоит в кажущемся существовании зависимости между машинами и домашними животными, так как эти данные размещаются рядом в одной строке. Лучше было бы поместить их в разные таблицы и связать с авторами посредством внешнего ключа.

|  |  |
| --- | --- |
| *car* | au\_id |
| 1987 Chevy Nova | 998-72-3567 |
| 1994 Volvo Station Wagon | 998-72-3567 |
| 1990 VW Rabbit | 409-56-7008 |
| 1989 Toyota Corolla | 213-46-8915 |

|  |  |
| --- | --- |
| *pet* | au\_id |
| Rover | 998-72-3567 |
| Spot | 409-56-7008 |
| Valiant | 213-46-8915 |
| Fluffy | 213-46-8915 |
| Sam | 213-46-8915 |

Пятая нормальная форма доводит весь процесс нормализации до логического конца, *разбивая таблицы на минимально возможные части для устранения в них всей избыточности данных*. Нормализованные таким образом таблицы обычно содержат минимальное количество информации, помимо первичного ключа. Например:

titles table

title\_id title

authors table

au\_id au\_lname au\_fname

authors&titles table

au\_id title\_id

prices table

title\_id price

advances table

title\_id advance

pets table

pet au\_id

Преимуществом преобразования БД в пятую нормальную форму является возможность управления целостностью. Поскольку при этом любой фрагмент не ключевых данных (данных, не являющихся первичным или внешним ключом) встречается в БД только один раз, не возникает никаких проблем при их обновлении. Если, например, изменяется стоимость книги, соответствующие поправки нужно внести только в таблицу *prices* и не надо просматривать остальные таблицы на предмет поиска и изменения в них значения соответствующего поля *price*.

Однако, поскольку каждая таблица в пятой нормальной форме имеет минимальное число столбцов, необходимо дублировать в них одни и те же ключи, обеспечивая возможности для объединения таблиц и получения полезной информации.

Изменение значения единственного ключа (например, *title\_id*) уже является очень серьезной проблемой. Необходимо найти все вхождения этого значения в БД и внести соответствующие изменения. К счастью, столбцы первичных ключей обычно изменяются значительно реже, чем не ключевые.

## Обзор базы данных

Для формирования БД *bookbiz* были созданы четыре таблицы: *authors, titles, publishers* и *editors.* Далее, произошло создание еще двух базовых таблиц. Одна из них используется для связи таблиц *titles* и *authors* (*titleauthors*), другая – для связи таблиц *titles* и *editors* (*titleditors*). Позже добавились таблицы *sales* и *salesdetails*.

К этому времени рассмотрения не коснулась единственная таблица – *roysched*, которая определяет авторские гонорары в виде зависимости от количества проданных книг. Таблица *roysched* является **поисковой таблицей** (lookup table), используемой исключительно для справочных целей. Ее данные модифицируются только в случае изменений условий контракта или выхода новой книги.

Таким образом, были созданы девять таблиц.

### Последние замечания о БД bookbiz

БД *bookbiz* содержит информацию о деятельности трех филиалов издательской компании, и так как они не являются независимыми, то используют одну и ту же БД.

Таблица *publishers* содержит информацию о трех издательствах: их идентификационные номера, названия и адреса.

Информация о каждом авторе, имеющем контракт с издателем, содержится в таблице *authors*: номер карточки социального страхования, имя, фамилия и адресные данные.

Аналогичную информацию о каждом редакторе содержит и таблица *editors*. Кроме того, в ней имеется дополнительный столбец, описывающий вид выполняемой редактором работы (подбор информации или управление всем проектом).

По всем вышедшим и готовящимся к печати книгам таблицы *titles* содержит следующую информацию: идентификационный номер, название, тема, идентификационный номер издателя, стоимость, расходы, количество проданных экземпляров, состояние, контракта, дополнительные заметки и дата выхода. Числа в столбце *ytd­\_sales* должны изменяться по мере увеличения количества проданных книг одним из следующих способов:

* с помощью команд модификации данных;
* из приложений, которые будут автоматически изменять значения в столбце *ytd\_sales*, как только будут вводиться новые данные в таблицу *salesdetails*;
* Используя SQL для определения триггеров, выполняющих автоматическое обновление.

Книги и авторы представлены в разных таблицах, но могут быть связаны с помощью третьей таблицы – *titleauthors.* Для каждой книги таблица *titleauthors* содержит строку с описанием идентификатора книги, идентификатора автора, позиции автора в списке авторов книги и информацию по разделению гонорара. Таблица *titleditors* аналогично связывает книги с их редакторами. Кроме того, она описывает порядок редактирования, т.е. можно узнать, кто был первым или последним редактором.

Таблица *roysched* описывает зависимость между количеством проданных книг и размером авторского гонорара. Гонорар составляет определенную часть суммы, полученной за проданные книги.

Талица *sales* содержит общую информацию о заказах, полученных от книжных магазинов: номер квитанции на продажу (получаемый от издателя), идентификатор магазина, номер заказа на покупку (получаемый от книжного магазина) и дату выполнения заказа.

Таблица *salesdetails* содержит информацию о каждом пункте заказа на покупку (предполагая, что сразу могут быть заказаны несколько книг): название, количество заказанных книг, количество отправленных книг и дата отправления.

Конечно, реальная БД может быть значительно сложнее и содержать таблицы для описания магазинов, дистрибьюторов, производственных расходов, посредников и т.д. Вместе с тем этих девяти таблиц вполне достаточно для изучения языка SQL.

### Проверка структуры БД

После разработки структуры БД необходимо создать таблицы и заполнить их некоторыми данными. Затем следует проверить разработанную структуру, запуская на выполнение запросы и изменяя данные. Эти тесты могут выявить недочеты в исходной структуре данных.

При проектировании БД нельзя полагаться исключительно на теорию. Перед переходом к серьезным действиям следует поэкспериментировать с прототипом БД.

Обеспечение целостности – одна из главных задач в процессе проектирования БД. Необходимо быть уверенным, что изменения в одной части данных распространяются на все копии этих данных по всей базе.

# Создание и заполнение БД

С помощью команды **SQL CREATE** можно определить в своей СУРБД имя, структуру и другие характеристики реальной БД и ее объектов. Помещать информацию в БД можно при помощи команды **INSERT**.

## Синтаксис SQL

Пример реальной команды SQL:

select pub\_id, pub\_name, address, city, state

from publishers

where pub\_id = '0736'

При описании синтаксиса SQL используются следующие соглашения:

* Несмотря на то что SQL является **языком с необусловленными формами (free-form language)**, не ограничивая тем самым количество слов в одной строке и места разрывов строк, в примерах этого конспекта каждый оператор обычно начинается с новой строки. Длинные и сложные предложения разбиваются на несколько строк, выделяемых отступом.
* Слова и фразы, которые необходимо подставлять в операторы, всегда записываются строчными буквами.
* Ключевые слова и операторы SQL всегда записываются прописными буквами, несмотря на то что в большинстве версий SQL их можно вводить в любом регистре. Например, SELECT, FROM, WHERE.
* Фигурные скобки (**{}**) вокруг слов или фраз означают, что необходимо выбрать по крайней мере одну из заключенных в них опций. Если опции разделены вертикальной чертой (**|**), можно использовать только одну из них. Если опции разделены запятой (**,**), можно выбрать одну или несколько опций.
* Квадратные скобки (**[]**) означают, что заключенные в них опции необязательны. Если, кроме того, опции разделены вертикальной чертой (**|**), можно либо вообще их не использовать, либо использовать только одну из них. Если же опции разделены запятой (**,**), вы можете не использовать ни одну из них, использовать одну или несколько.

Фигурные и квадратные скобки не являются частью оператора, поэтому их не нужно набирать при вводе команды SQL. При выборе нескольких опций нужно разделить их запятыми. Многоточие (**…**) означает, что можно повторить последнюю конструкцию нужное количество раз.

Ниже приведен пример (не SQL) того, как эти элементы могут использоваться в записи команд.

BUY thing\_name = price AS {cash | check | credit}

[, thing\_name = price AS {cash | check | credit}]…

В этом выдуманном операторе BUY и AS являются ключевыми словами. Опции, заключенные в фигурные скобки и разделенные вертикальной чертой, означают, что необходимо выбрать один (и только один) способ оплаты. Также можно приобрести и ряд других вещей. Для каждой покупаемой вещи необходимо указать ее название, стоимость и способ оплаты.

{early\_lunch | no\_lunch} Выбор одного элемента

{soup | salad | sandwich} Выбор одного или нескольких элементов

[dessert] Нет выбора

[coffee | soda | wine] Можно выбрать один элемент или не выбирать вовсе

[tomato, pickle, onion] Можно выбрать любое количество элементов

Вот как будет выглядеть реальный пример:

buy lunch = 4.95 as cash, book = 34.95 as check

Если в записи оператора встречаются круглые скобки (**()**), они действительно являются его частью (в отличие от фигурных и квадратных скобок). Не следует забывать об этом!

Операторы SQL обычно требуют наличия **терминаторов (terminator)**, которые инициируют исполнение оператора в СУБД. Различные SQL используют разные терминаторы, наиболее распространенными являются точка с запятой (**;**) и слово **go**.

### Обработка ошибок

Ошибки в операторах SQL могут возникать по целому ряду причин. Наиболее распространенными являются ошибки при наборе, синтаксические ошибки, неправильное использование кавычек и имен объектов БД.

Например, нужно вводить не slect, а select.

Предложение FROM должно предшествовать предложению WHERE.

Слово "СА" должно заключаться в кавычки.

Таблица называется titles, а не title.

## Создание БД

В коммерческой версии SQL обычно включена команда **CREATE DATABASE**. Ее упрощенная форма имеет вид

CREATE DATABASE имя\_базы\_данных

В зависимости от реализации в запись этого оператора могут входить разные предложения, позволяющие разработчику или (сисадмину) управлять расположением БД (устройством БД, на котором физически расположена БД), ее размером (пространством, которое может использоваться под БД) и другими параметрами. К примеру:

CREATE DATABASE имя\_базы\_данных

[ON {DEFAULT | имя устройства} [ = SIZE]

[ , имя устройства [ = SIZE] ] ]

[LOG ON устройство\_базы\_данных [ = SIZE] ] ]

[WITH OVERRIDE]

[FOR LOAD]

В зависимости от используемой версии SQL сразу после создания БД введение следующего:

USE имя\_базы\_данных

или

DATABASE имя\_базы\_данных

Возможно также, что для выбора нужной БД потребуется использование команды CONNECT. Данные нюансы нужно уточнять в документации к системе.

## Создание таблиц

Таблицы являются основными строительными блоками БД. В них содержатся строки и столбцы данных. С помощью команд определения данных SQL можно создавать, удалять и манипулировать таблицами (добавлять, удалять, переставлять столбцы и менять их параметры).

В большинстве реализаций SQL таблицей владеет создавший ее пользователь, выдавая разрешения на ее использование другим пользователям. Чтобы создать таблицу, по меньшей мере, надо сделать следующее:

* Задать имя таблицы.
* Задать имена составляющих ее столбцов.
* Определить тип данных для каждого столбца.
* Определить (или использовать заданный по умолчанию) нулевой статус для каждого столбца – допускается или запрещается использование в столбце нулевых значений.

Для максимальной гибкости в БД bookbiz при определении таблиц используются только эти основные элементы. Тем не менее, большинство производителей предусматривают дополнительные возможности для оператора CREATE TABLЕ:

* Значения по умолчанию (автоматически вводимые значения, например, текущая дата в соответствующем столбце заказа).
* Ограничения на возможные значения в столбцах (например, в столбце *pub\_id* можно вводить только значения 0736, 0877 и 1389).
* Ограничения на определение первичных ключей.
* Проверка соответствия между первичными и внешними ключами.

Предлагаем начать с упрощенного синтаксиса оператора CREATE TABLE (имя столбца, тип данных и статус нуля). Ограничения будут рассмотрены далее.

Команда SQL для определения таблицы записывается в виде CREATE TABLE.

Упрощенный синтаксис оператора CREATE TABLE имеет следующий вид:

CREATE TABLE имя\_таблицы

(имя\_столбца тип\_данных [NULL | NOT NULL]

[ , имя\_столбца тип\_данных [NULL | NOT NULL] ] )

В качестве примера рассмотрим оператор CREATE TABLE, с помощью которого создается таблица *authors:*

SQL:

create table authors

(au\_id char(11) not null,

au\_lname varchar(40) not null,

au\_fname varchar(20) not null,

phone char(12) null,

address varchar(40) null,

city varchar(20) null,

state char(2) null,

zip char(5) null)

### Выбор типа данных

Тип данных столбца определяет, какого рода данные будут содержаться в этом столбце (символы, числа, даты и т.д.).

**Символьные типы данных** (character datatypes)

В некоторых наборах символов также поддерживаются национальные символы фиксированной и переменной длины (*national character, nchar, national character varying, nvarchar*). Ряд систем допускают использование специальных символьных типов данных для записи больших текстовых фрагментов. Обычно они называются типами *long* или *text*. Также в символьных столбцах можно использовать специальные символы, например, дефисы и скобки в номерах телефонов.

**Целые типы данных** (whole-number datatypes)

**Десятичные типы данных** (decimal datatypes). Для десятичных чисел с плавающей точкой используются имена *decimal* или *numeric*. Обычно разрешается определять их точность (общее количество цифр и количество цифр после запятой). Вещественные числа в стандартном формате описываются ключевыми словами *real, double, double precision, float* и *smallfloat.*

**Денежные типы данных** (money datatypes)

**Дата и время** (date and time datatypes)

**Двоичные типы данных** (binary datatypes) используются для хранения двоичных и шестнадцатеричных кодов.

**Последовательные типы данных** (serial datatypes) используются для представления последовательно возрастающих числовых показателей.

#### Выбор длины типа данных

Прежде всего нужно уточнить, как физически хранятся в системе данные разных типов. Нужно узнать, какие из них имеют фиксированную длину. К примеру, под все значения столбца, определенного как *char(10)*, всегда будет отводиться по десять байт дискового пространства. Более длинные значения будут подрезаться, а к более коротким справа будут добавляться пробелы.

#### Выбор точности

Десятичные типы с плавающей точкой позволяют устанавливать требуемую точность. Например, если известно, что десятичное значение состоит строго из шести цифр, причем три из них располагаются после запятой, его можно описать как *decimal (6, 3)*.

### Назначение нулевого статуса

В большинстве систем при создании таблицы можно указать статус каждого столбца (нулевой или ненулевой). При назначении нулевого статуса столбцу пока пользователь не введет конкретные данные, все его значения будут нулевыми. Стоит напомнить, что **под нулем подразумевается неизвестное, недопустимое или неприменимое значение, а не обычный нуль или пробел**.

Назначение ненулевого статуса столбцу означает, что все его значения должны быть определены. Если не задать какое-то значение столбца, система выдаст соответствующее сообщение об ошибке.

При проектировании таблицы нельзя допускать появление нулей в столбцах, значения которых являются критическими для всей БД. К примеру, нули не могут появляться в столбцах первичных ключей, так как в этом случае они не будут однозначно идентифицировать строки. Так, без столбца *au\_id* нельзя было бы определить авторов книг из БД *bookbiz*. Эта БД построена так, что таблицы *authors* и *titles* связываются через объединение по столбцам *au\_id* и *title\_id*.

Кроме того, целесообразно запретить нулевые значения в столбцах *au\_lname* и *au\_fname*, так как достаточно странно вводить адрес и телефон автора, не зная его имени и фамилии.

Однако если известно, что определенные данные в столбце могут быть неизвестны, тогда можно разрешить появление в нем нулевых значений. К примеру, могут быть неизвестны адреса и номера телефонов. В столбце имен также могут быть нулевые значения, так как некоторые авторы используют однословные псевдонимы.

### Процесс создания таблицы

Создание таблицы, подобно созданию БД, представляет собой пошаговый процесс, начинающийся с проектирования и заканчивающийся вводом команд SQL.

Основные действия, которые надо предпринять для создания таблицы:

* Определить типы данных (длину, точность, если необходимо) для каждого столбца.
* Определить, какие столбцы допускают нулевые значения, а какие – нет.
* Решить, какие столбцы должны содержать только уникальные значения.
* Разобраться с парами первичный ключ-внешний ключ. Используя описываемый далее более сложный синтаксис можно определять ограничения для обеспечения ссылочной целостности.
* В случае многопользовательского окружения убедиться в наличии прав на создание таблиц и индексов. При их отсутствии нужно поговорить об этом с системным администратором или владельцем используемой БД.
* Создать таблицу (и в случае необходимости все нужные индексы) с помощью операторов CREATE TABLE CREATE и INDEX.

Ниже в таблице представлена структура таблицы *titles*: имена столбцов, типы данных, нулевые статусы, столбцы с уникальными значениями, первичные и вторичные ключи.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Столбец** | **Тип данных** | **Нуль?** | **Уникальный?** | **Ключи** |
| title\_id | varchar(11) | not null | да-titleidind | Первичный |
| title | varchar(80) | not null | да-titleidind |  |
| type | char(12) | null |  |  |
| pub\_id | char(4) | null |  | Внешний (таблица publishers) |
| price | money | null |  |  |
| advance | money | null |  |  |
| ytd\_sale | int | null |  |  |
| contract | bit | not null |  |  |
| notes | varchar(200) | null |  |  |
| Pubdate | datetime | null |  |  |

#### Определение таблиц в БД bookbiz

Ниже приведена команда **CREATE TABLE** для таблиц *titles*.

create table titles

(title\_id char(6) not null,

title varchar(80) not null,

type char(12) null,

pub\_id char(4) null,

price money null,

advance money null,

ytd\_sales int null,

contract bit not null,

notes varchar(200) null,

pubdate datetime null)

## Создание индексов

В СУРБД индексация является механизмом для повышения производительности. Подобно тому, как предметный указатель помогает находить требуемые страницы в книге, индексирование ускоряет выборку информации из БД. При поиске определенной темы в книге читатель не просматривает для этого подряд все страницы. Аналогично, при поиске нужных данных индексы служат в качестве логических указателей на их физическое местоположение.

Однако между предметным указателем в книге и индексом в БД существует несколько важных отличий. Читатель книги сам решает, обращаться ему к предметному указателю или нет. Пользователь реляционной БД только решает, создавать ему индекс или нет, а способы его использования при выполнении запросов определяет исключительно система. Данные в реляционных системах и индексах могут постоянно меняться. В любой момент данные в таблице могут измениться, при этом должны соответствующим образом поменяться один или несколько индексов. При этом все заботы о поддержке индексов берет на себя система.

Еще одно отличие между предметными указателями книг и индексами БД состоит в том, что очень часто таблица имеет несколько индексов.

Далее приводится синтаксис операторов **CREATE INDEX** и **DROP INDEX**.

### Оператор CREATE INDEX

В большинстве систем имеется команда следующего вида:

CREATE [UNIQUE] INDEX имя\_индекса

ON имя\_таблицы (имя\_столбца)

Имена таблицы и столбца определяют столбец, который необходимо индексировать, и таблицу, которой он принадлежит.

Чтобы проиндексировать таблицу *authors* по столбцу *au\_id,* нужно выполнить следующую команду:

create index auidind

on authors(au\_id)

Лучше всего выполнять индексирование при создании таблицы, однако SQL также позволяет создавать индексы и после заполнения таблиц данными.

Во многих системах поддерживаются **составные индексы** (composite indexes) – индексы, использующие несколько столбцов, и **уникальные индексы** (unique indexes) – предотвращающие дублирование данных. В некоторых системах реализованы **групповые индексы** (clustered indexes) с возможностью не только логической, но и физической сортировки.

#### Составные индексы

Составные индексы используются, когда одновременно лучше осуществлять поиск по двум или нескольким столбцам, что объясняется их логической взаимосвязью. Например, таблица *authors* имеет составной индекс на основе столбцов *au­­­­\_fname* и *au\_lname*.

При создании составного индекса нужно указать все соответствующие столбцы. Команда, создающая составной индекс по таблице *authors*, может иметь следующий вид:

create index aunameind

on authors (au\_lname, au\_fname)

В большинстве диалектов SQL порядок столбцов при создании составного индекса не обязательно должен повторять их порядок в операторе CREATE TABLE. С точки зрения производительности лучше первым указывать столбец, по которому чаще всего выполняется поиск.

#### Уникальные индексы

Уникальность означает, что никакие две строки не могут иметь индексы с одинаковыми значениями. В этом случае при создании индекса и при каждом добавлении информации система будет следить за тем, чтобы значения индексов не повторялись. Уникальные индексы обычно создаются на ключевых столбцах, чтобы усилить их возможности как уникальных идентификаторов строк.

Создавать уникальные индексы имеет смысл тогда, когда это диктуется данными. Например, не нужно создавать уникальный индекс по столбцу *last\_name*, так как даже в таблице всего из нескольких сотен строк наверняка найдется несколько человек с фамилиями *Wong* или *Smith*. С другой стороны, целесообразно создать уникальный индекс по столбцу с номерами карточек социального страхования. В этом случае уникальность является характеристикой данных: эти номера обязательно различаются у разных людей.

Реализации SQL, поддерживающие уникальные индексы, должны иметь для этого специальные механизмы. Обычно система гарантирует уникальность, отбрасывая команды, которые пытаются выполнить следующее:

* Создать уникальный индекс на существующих данных, которые содержат одинаковые значения.
* Изменить данные, на основе которых построен уникальный индекс, таким образом, чтобы в них появились одинаковые значения.

Для создания составных и простых (одностолбцовых) индексов применяется ключевое слово **UNIQUE**.

#### Групповые индексы

Некоторые СУРБД предоставляют пользователям выбор между групповыми и негрупповыми индексами. При создании группового индекса строки таблиц сортируются таким образом, что их физический порядок на устройстве БД совпадает с их логическим (индексным) порядком.

Поскольку групповой индекс управляет физическим положением данных, таблица может иметь только один такой индекс. Групповые индексы обычно создаются на первичных ключах, но целесообразнее использовать для их построения столбцы с наиболее часто запрашиваемой информацией.

При использовании негруппового индекса физический порядок строк не совпадает с их индексным порядком. Все негрупповые индексы таблицы могут обеспечивать доступ к данным в разном порядке.

Поиск данных с использованием групповых индексов почти всегда выполняется быстрее поиска с негрупповыми индексами. Это преимущество особенно ощутимо, когда из БД извлекается целый набор строк с последовательными **ключевыми значениями** (key values). Как только будет найдена строка с первым ключевым значением, дальнейший поиск прекращается, так как строки с последовательными индексными значениями фактически располагаются друг за другом. Однако наличие группового индекса может замедлять выполнение операторов модификации данных, так как системе требуется время на перестройку индекса при изменении ключевых значений.

### Как, что и зачем нужно индексировать

Индексирование ускоряет процесс выборки данных. После создания индекса по столбцу запрос, который до этого требовал для своего выполнения достаточно много времени, может отработать почти мгновенно.

Отказ от индексации всех столбцов объясняется требованием времени и места на устройстве БД при построении и поддержке индексов. Другая причина связана с тем, что вставка, удаление или изменение данных в индексированных столбцах требует несколько большего времени, чем в неиндексированных, так как при изменении ключевых значений системе необходимо дополнительное время на перестройку индекса. Однако это обычно компенсируется за счет увеличения производительности при выборке данных.

В общем случае имеет смысл индексировать только часто используемые в запросах столбцы, в первую очередь ключевые столбцы и столбцы, используемые для объединения и сортировки. Предлагаем несколько более точных рекомендаций. Обычно должны индексироваться:

* Столбцы с первичными ключами, особенно если они часто используются в операциях объединения с другими таблицами. Уникальные индексы на первичных ключах предотвращают появление в них одинаковых значений и гарантируют, что каждое значение в столбце первичного ключа будет однозначно идентифицировать строку.
* Столбцы, по которым часто выполняется сортировка.
* Столбцы, постоянно используемые в операциях объединения, так как в этом случае объединение будет выполняться быстрее.
* Столбцы, по группам значений которых часто выполняется поиск, особенно если система поддерживает групповые индексы. Как только будет найдена строка с первым искомым значением, дальнейший поиск прекращается, так как строки с последовательными значениями физически располагаются друг за другом. При поиске единственного значения групповые индексы такими преимуществами не обладают.

Существует ряд случаев, в которых применение индексов нецелесообразно.

* Индексирование столбцов, которые редко используются в запросах, не приведет к повышению производительности, так как их значения очень редко или вообще не используются при поиске и выборке строк из БД. Однако и в этом случае индексирование можно применять для обеспечения уникальности значений столбца.
* Не имеет смысла индексировать столбцы только с двумя-тремя значениями (например, с описанием пола).
* Индексирование не приводит к ощутимому выигрышу на небольших таблицах с несколькими строками. В этом случае для поиска информации обычно используется простое **сканирование таблицы** (table scan) – поочередный просмотр строк, а не индекс. Время такого сканирования пропорционально количеству строк в таблице.

## Создание таблиц с помощью ограничений SQL-92

Большинство коммерческих систем поддерживает предложения (ограничения) **PRIMARY KEY, UNIQUE, DEFAULT, CHECK, REFERENCES** и **FOREIGN KEY** в команде **CREATE TABLE**. Эти элементы обеспечивают защиту целостности данных.

* **PRIMARY KEY** помечает столбец (в котором не могут присутствовать нулевые значения) в качестве первичного ключа таблицы. Каждое вводимое в этот столбец значение должно быть уникальным, ввод одинаковых значений не допускается. Если первичный ключ включает несколько столбцов, **PRIMARY KEY** определяется на уровне таблицы.
* **UNIQUE** также гарантирует отличие между всеми значениями в столбце, но допускает в нем одно нулевое значение.
* **DEFAULT** определяет значение, которое автоматически вставляется системой, если пользователь не введет требуемые данные. Например, можно определить значение по умолчанию для столбца *type* в таблице *titles*. Если книга еще не классифицирована, то система могла бы автоматически вставлять в этот столбец любое назначенное вами значение, например, слово unclassified (не классифицирована). Если значение по умолчанию не определено, но столбец *type* допускает нулевые значения, система автоматически вставит вместо неизвестных данных значение NULL.
* **CHECK** определяет, какие данные могут вводиться в определенный столбец, т.е. позволяет определить область значений столбца. К примеру, вы можете захотеть, чтобы в столбец *title\_id* всегда вводилось значение, состоящее из букв и четырех цифр, а в столбец *type* – один из шести допустимых терминов. Если пользователь попытается нарушить это ограничение, соответствующая команда модификации данных выполняться не будет. Эти ограничения иногда называются **правилами** (rules), поскольку позволяют проверить принадлежность вводимых данных области значений данного столбца. Правила, использующие несколько столбцов, определяются на уровне таблицы.
* **REFERENCES** и **FOREIGN KEY** связывают вместе первичные и внешние ключи. При введении значения в столбец внешнего ключа, определенного с помощью предложения **REFERENCES**, этот столбец и столбец, на который он ссылается, должны существовать в таблице, иначе команда модификации данных будет отвергнута.

Перед тем как определять ограничения и значения по умолчанию, необходимо четко сформулировать свои требования. В таблице представлены значения по умолчанию и ограничения, используемые в таблице *titles*.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Столбец** | **Тип данных** | **Нуль?** | **Ключ** | **Значение по умолчанию** | **Ограничения** | **Ссылки** |
| title\_id | char(6) | not null | первичный,  уникальный |  | 2 буквы и 4 цифры |  |
| title | varchar(80) | not null | уникальный |  |  |  |
| type | char(12) |  |  | не классифицирована | business, mod\_cook, trad\_cook, psychology, popular\_comp, unclassified |  |
| pub\_id | char(4) | null |  |  |  | publishers, pub\_id |
| price | money | null |  |  |  |  |
| advance | money | null |  |  |  |  |
| ytd\_sales | int | null |  |  |  |  |
| contract | bit | not null |  |  |  |  |
| notes | varchar(20) | null |  |  |  |  |
| pubdate | datetime | null |  | текущая дата |  |  |

Синтаксис этих предложений может быть различным в разных системах, так что нужно уточнить его из руководства к системе.

Кроме того, эти предложения можно узнать по ключевым словам **DEFAULT, CHECK, PRIMARY KEY, UNIQUE, FOREIGN KEY** и **REFERENCES**. Помимо этого, может присутствовать слово **CONSTRAINT** и имя ограничения. Ограничения на таблицу лучше всего помещать после списка всех ее элементов (хотя это и необязательно). Столбцы могут иметь более одного ограничения, причем ограничения не разделяются запятыми:

CREATE TABLE имя\_таблицы

(имя\_столбца тип [NULL | NOT NULL] [DEFAULT значение\_по\_умолчанию]

[ограничение\_на\_столбец]…

[ , имя\_столбца тип [NULL | NOT NULL] [DEFAULT значение\_по\_умолчанию]]

[ограничение\_на\_столбец]…]…

[ограничение\_на\_таблицу]…)

Ниже представлена SQL-версия таблицы *titles* для SQL Anywhere (под названием *titlescnstr*), использующая значения по умолчанию и ограничения.

(Предложения **LIKE** и **IN** рассматриваются в следующей главе. Для столбца *pubdate* SQL Anywhere использует значение по умолчанию CURRENT DATE, соответствующее текущей дате).

create table titlescnstr

(title\_id char(6) not null

primary key

check (title\_id like '[A-Z] [A-Z] [0-9] [0-9] [0-9] [0-9] '),

title varchar(80) not null

unique,

type char(12)

default 'unclassified' null

check (type in ('business', 'mod\_cook', 'trad\_cook',

'psychology', 'popular\_comp', 'unclassified')),

pub\_id char(4) null

references publishers (pub\_id),

price money null,

advance money null,

ytd\_sales int null,

contract bit not null,

notes varchar(20) null,

pubdate datetime null

default current date)

Необходимо уточнить соответствующие детали в своем руководстве по системе. Ограничения на таблицу используют более одного столбца. В таблице указаны ограничения для таблицы *titleauthors*.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Столбец** | **Тип данных** | **Нуль?** | **Ключ** | **Значение по умолчанию** | **Ограничения** | **Ссылки** |
| au\_id | char(11) | not null | первичный ключ |  |  | authors.au\_id |
| title\_id | char(6) | not null | первичный ключ |  |  | titles.title\_id |
| au\_ord | tinyint | not null |  |  |  |  |
| royaltyshare | float | not null |  |  |  |  |

Ниже показано, как в SQL Anywhere реализуются ссылки на примере таблицы *titleauthors (titleauthorscnstr)*. Ограничение на первичный ключ является ограничением на всю таблицу, так как использует более одного столбца.

create table titleauthorscnstr

(au\_id char(11) not null references authors (au\_id),

title\_id char(6) not null references titles (title\_id),

au\_ord tinyint not null,

royaltyshare float not null,

primary key (au\_id, title\_id))

Ограничения с проверкой также могут применяться ко всей таблице, если они используют более одного столбца (столбцы должны принадлежать одной таблице). Например, в таблице *salesdetails* есть столбцы для описания количества заказанных и отправленных книг. В этом случае можно использовать табличное ограничение, проверяющее, что количество отправленных книг не превышает количество заказанных.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Столбец** | **Тип данных** | **Нуль?** | **Ключ** | **Значение по умолчанию** | **Ограничения** | **Ссылки** |
| sonum | int | not null | первичный ключ |  |  | sales.sonum |
| title\_id | char(6) | not null | первичный ключ |  |  | titles.title\_id |
| qty\_shipped | smallint | null |  |  | qty\_shipped not > than qty\_ordered |  |
| qty\_ordered | smallint | not null |  |  | qty\_shipped not > than qty\_ordered |  |
| date\_shipped | datetime | null |  |  |  |  |

create table salesdetailscnstr

(sonum int not null references sales (sonum),

qty\_ordered smallint not null,

qty\_shipped smallint null,

title\_id char(6) not null references titles

(title\_id),

date\_shipped datetime null,

check (qty\_shipped <= qty\_ordered),

primary key (sonum, title\_id))

Чтобы предложение **REFERENCES** работало надлежащим образом, оператор SQL Anywhere **CREATE TABLE** требует, чтобы столбец *sonum* в таблице *sales* являлся либо первичным ключом, либо имел уникальный индекс.

В ряде систем имеются специальные команды (не связанные с оператором **CREATE TABLE**) для обработки значений по умолчанию, правил и ограничений на целостность.

## Изменение и удаление БД и их объектов

БД и их объекты создаются с помощью той или иной модификации команды **CREATE**. Для удаления самой БД и объектов БД в большинстве систем предусмотрена команда **DROP**.

### Изменение БД

В некоторые версии SQL включена команда, позволяющая изменять размер БД (обычно в большую сторону). Возможность выделения дополнительного пространства на устройстве БД особенно важна для растущих со временем приложений. Не менее важна (но реже реализуема) возможность просмотра БД на предмет наличия неиспользуемого пространства на устройстве БД.

### Изменение определений таблицы

Спроектировав и создав БД, через определенное время можно обнаружить, что она недостаточно хороша, или что просто изменились некоторые требования. В некоторых системах с помощью команды **ALTER TABLE** можно изменять структуру таблицы (даже если она заполнена данными). В ней могут использоваться ключевые слова для добавления и удаления столбцов, изменения их имени, типа, длины, нулевого статуса и ограничений.

Например, чтобы добавить столбец к таблице *authors*, можно попробовать ввести следующую команду:

alter table authors

add birth\_date datetime null

Столбцы, добавленные к таблице с помощью оператора **ALTER TABLE**, как правило, должны допускать нулевые значения. Это связано с тем, что если новый столбец добавляется к уже заполненной данными таблице, то в нем должны содержаться некоторые значения, и самым естественным выбором при этом являются "нулевые" (неизвестные) значения.

Например, для изменения типа столбца таблицы можно воспользоваться следующей командой:

alter table pers\_bankname

alter column creditrate smallint null

### Удаление БД

Синтаксис команды **DROP DATABASE** обычно имеет следующий вид:

DROP DATABASE имя\_базы\_данных

Это очень опасная команда, так как при ее выполнении уничтожается все содержимое БД.

### Удаление таблиц

В большинстве диалектов SQL удаление таблицы выполняется следующим образом:

DROP TABLE имя\_таблицы

При выполнении этой команды из БД удаляется заданная таблица со всем ее содержимым. Если же требуется сохранить структуру таблицы, но удалить из нее все данные, можно воспользоваться командой **DELETE**.

Для удаления столбца необходимо ввести следующую команду:

alter table personnel

drop column creditcard

### Удаление индекса

Удаление индекса может потребоваться в двух ситуациях:

* кто-то создал индекс по столбцу, но в большинстве запросов он не используется;
* необходимо серьезным образом модифицировать ключевые значения (поскольку при каждом таком изменении система будет перестраивать индекс, имеет смысл сначала удалить его, а после выполнения модификаций создать индекс заново).

Хотя повторное создание индекса потребует некоторого времени, это удобнее, нежели наблюдать, как система "задумывается" при выполнении каждого оператора модификации данных.

В большинстве систем команда удаления индекса имеет следующий вид:

DROP INDEX имя\_таблицы.имя\_индекса

При выполнении этой команды система удаляет из БД заданный индекс. Команда для удаления индекса *auidind* в таблице *authors* имеет следующий вид:

SQL:

drop index authors.auidind

## Добавление, удаление и изменение данных

После проектирования и создания БД (таблиц, и, возможно, индексов) для начала полноценной работы в нее нужно поместить данные, которые впоследствии можно будет в случае необходимости добавлять, изменять и удалять. Имеющуюся структуру необходимо наполнить содержанием.

В SQL для изменения данных используются три основные команды (их часто называют операторами модификации данных):

* Оператор **INSERT** добавляет новые строки в БД.
* Оператор **UPDATE** изменяет существующие в БД строки.
* Оператор **DELETE** удаляет строки из БД.

В этом разделе описываются команды SQL по модификации данных.

Другой метод добавления данных в таблицу подразумевает их загрузку с помощью специальной команды вставки из файла ОС. Этот метод особенно подходит для переноса данных из одной СУБД в другую.

Реляционные системы с графическим интерфейсом пользователя предоставляют для ввода данных специальные **формы** (form), напоминающие обычный бумажный бланк, в строки которого вводятся необходимые данные. Формы обычно удобнее использовать, нежели операторы модификации данных, так как они позволяют автоматизировать и упростить работу. Однако все действия по модификации в СРБД выполняются на основе команд SQL, так что изучать их надо даже независимо от необходимости использования в будущем.

Выполнять операторы модификации данных обычно разрешается не всем. Владелец БД или владельцы отдельных объектов БД с помощью операторов **GRANT** и **REVOKE** могут разрешить отдельным пользователям выполнять определенные команды модификации.

С помощью каждого оператора модификации за один раз можно изменять данные только в одной таблице. Однако в ряде систем эти изменения могут касаться данных из других таблиц и даже других БД. Используя оператор **SELECT** в команде модификации данных, можно переместить значения из одной таблицы в другую.

### Добавление новой строки

Оператор **INSERT** позволяет добавлять строки в БД одним из двух способов: с помощью ключевого слова **VALUES** или с помощью оператора **SELECT**. Опишем сначала правила использования ключевого слова **VALUES**.

Ключевое слово **VALUES** определяет значения некоторых или всех данных в столбцах новой строки. Общая форма оператора **INSERT**, использующего ключевое слово **VALUES**:

INSERT INTO имя\_таблицы [ ( столбец1 [ , столбец2]…) ]

VALUES ( константа1 [ , константа2]…)

Следующий оператор **INSERT** добавляет новую строку в таблицу *publishers* и задает значения для каждого столбца:

insert into publishers

values('1622', 'Jardin, Inc.', '5 5th Ave.', 'Camden', 'NJ')

Нужно обратить внимание, что значения вводятся в том порядке, в котором определялись столбцы в соответствующем операторе **CREATE TABLE** (другими словами, сначала идентификационный номер, затем имя, адрес, город и, наконец, штат). Данные после ключевого слова **VALUES** заключаются в круглые скобки. В большинстве систем значения вводятся в двойных или одинарных кавычках и разделяются запятыми.

Для каждой добавляемой строки используется отдельный оператор **INSERT**.

#### Вставка данных в несколько столбцов

Если данные добавляются не во все столбцы таблицы, их нужно дополнительно определить. Во избежание сбоев, для столбцов значения которых не вводятся, должны быть определены значения по умолчанию либо они должны допускать нулевые значения. Например, для добавления данных только в два столбца (*pub\_id* и *pub\_name*) используется следующая команда:

insert into publishers(pub\_id, pub\_name)

values('1756', 'HealthText')

Порядок перечисления столбцов в операторе **INSERT** может быть любым, но при этом он должен соответствовать порядку перечисления значений. Например, тот же результат достигается при выполнении следующего оператора (в нем переставлены столбцы *pub\_name* и *pub\_id*):

insert into publishers(pub\_name, pub\_id)

values('HealthText', '1756')

Оба этих оператора помещают значение "1756" в столбец идентификационных номеров и "HealthText" в столбец имен издателей. А что будет со столбцами *address, city* и *state?* Добавленную в таблицу *publishers* строку можно просмотреть с помощью оператора **SELECT**:

select pub\_id, pub\_name, address, city, state

from publishers

where pub\_name = 'Healthtext'

Результат:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| pub\_id | pub\_name | address | city | state |
| 1756 | HealthText | NULL | NULL | NULL |

Так как в операторе **INSERT** не определены значения для столбцов *address, city* и *state*, а таблица *publishers* допускает использование для них нулевых значений, они будут обнулены.

Если в операторе **CREATE TABLE** для столбцов *city* и *state* определен ненулевой статус, предыдущий оператор **INSERT** не сработает, так как для таких столбцов должны вводиться все значения.

### Использование оператора SELECT в команде INSERT

Для получения данных из одной или нескольких таблиц в команде **INSERT** можно использовать оператор **SELECT**. Упрощенный синтаксис команды **INSERT**, использующей оператор **SELECT**, имеет следующий вид:

INSERT INTO имя\_таблицы [ (вставляемый\_список\_столбцов) ]

SELECT список\_столбцов

FROM список\_таблиц

WHERE условия

Оператор **SELECT** в команде **INSERT** позволяет взять данные из нескольких или всех столбцов одной таблицы и вставить их в другую таблицу. Если вставлять значения только для части столбцов, определить значения для других можно будет позднее с помощью оператора **UPDATE**.

Если вставлять строки из одной таблицы в другую, эти таблицы должны иметь совместимую структуру, то есть соответствующие столбцы должны иметь одинаковый тип, или же система должна уметь автоматически выполнять нужное преобразование.

Если столбцы в обеих таблицах совместимы по типам и определены в одинаковом порядке в соответствующих операторах **CREATE TABLE**, перечислять их в команде **INSERT** необязательно. Предположим, что в таблице *newauthors* содержатся строки с информацией об авторах в том же формате, что и в таблице *authors*. Для добавления всех строк из таблицы *newauthors* в таблицу *authors* можно воспользоваться одной из следующих команд:

insert into authors

select au\_id, au\_lname, au\_fname, phone, address, city, state, zip

from newauthors

insert into authors

select \*

from newauthors

Если столбцы в двух таблицах (таблица, в которую вставляются данные, и таблица, из которой берутся данные) определены в разном порядке в соответствующих операторах **CREATE TABLE**, для установления соответствия между ними можно воспользоваться предложениями **INSERT** или **SELECT**.

Например, предположим, что в операторе **CREATE TABLE** для таблицы *authors* столбцы определены в следующем порядке – *au\_id, au\_fname,* *au\_lname* и *address*, а для таблицы *newauthors* – *au\_id, address, au\_fname* и *au\_lname*. Тогда установить соответствие между ними можно с помощью оператора **INSERT**. Для этого столбцы таблицы *authors* нужно перечислить в предложении **INSERT**:

insert into authors(au\_id, address, au\_lname, au\_fname)

select \*

from newauthors

Такой же результат можно получить, перечислив в нужном порядке столбцы таблицы *newauthors* в предложении **SELECT**:

insert into authors

select au\_id, au\_fname, au\_lname, address

from newauthors

Если последовательности столбцов в двух таблицах не согласованы, операция **INSERT** либо не будет выполняться вообще, либо выполнится не полностью. При этом данные могут быть размещены в неверных столбцах.

#### Выражения

Одним из преимуществ использования оператора **SELECT** в команде **INSERT** является возможность включения в него различных **выражений** (expression) – строк символов, математических формул и функций, позволяющих манипулировать вставляемыми данными.

Ниже приводится пример предложения **SELECT**, в котором над столбцом выполняются математические действия. Предположим, что один из филиалов описываемой нами компании перекупил серию книг у другой издательской компании. Причем, по счастливой случайности, для описания книг эта компания использовала таблицу с той же структурой, что и таблица *titles*. Эти книги описаны в таблице *Books* и их нужно поместить в таблицу *tiltes*. Однако при покупке стоимость этих книг была увеличена на 50 процентов. Оператор, увеличивающий значения стоимости книг и вставляющий строки из таблицы *Books* в таблицу *titles*, имеет следующий вид:

insert into titles

select title\_id, title, type, pub\_id, price \* 1.5,

advance, royalty, ytd\_sales, contract, notes, pubdate

from Books

***Вставка данных в несколько столбцов.***

С помощью оператора **SELECT** можно добавлять данные как во все сразу, так и в отдельные столбцы, по аналогии с предложением **VALUE**. Для этого нужно просто задать имена столбцов, в которые вы хотите добавить данные в предложении **INSERT**.

Например, если в таблице *titles* имеются книги, на которые еще не заключены контракты, и которые, следовательно, не представлены соответствующими строками в таблице *titleauthors*, тогда для выборки идентификационных номеров из таблицы *titles* и вставки их в таблицу *titleauthors* (пока просто для резервирования места) можно воспользоваться следующим оператором:

insert into titleauthors (title\_id)

select title\_id

from titles

where contract = 0

Однако в этот оператор закралась ошибка, так как необходимо ввести значение в столбец *au\_id* таблицы *titleauthors* (согласно определению этой таблицы, в столбце *au\_id* не допускаются нулевые значения и для него не определены значения по умолчанию). Поэтому в столбец *au\_id* в виде константы надо поместить **фиктивное значение** (dummy value), например, хххххх:

insert into titleauthors(title\_id, au\_id)

select title\_id, 'xxxxxx'

from titles

where contract = 0

В результате выполнения этой команды таблица *titleauthors* будет содержать две новые строки с реальными значениями в столбце *title\_id*, фиктивными значениями в столбце *au\_id* и нулевыми значениями в двух других столбцах. Однако это не сработает, если по столбцу построен уникальный индекс или на него наложены ограничения **UNIQUE** или **PRIMARY KEY**.

В большинстве версий SQL не допускается использование одной и той же таблицы для выборки и вставки данных:

insert into test

select \*

from test

Sybase SQL Server – одна из немногих систем, допускающих подобный синтаксис.

## Изменение существующих данных

В то время как оператор **INSERT** добавляет в таблицу новые строки, оператор **UPDATE** предназначен для изменения существующих в таблице данных.

В операторе **UPDATE** нужно указать изменяемые строки и их новые значения. Новые данные могут быть константами или выражениями, или могут быть получены из других таблиц.

Ниже приведен упрощенный синтаксис команды **UPDATE**, изменяющей выбранные строки:

UPDATE имя\_таблицы

SET имя\_столбца = выражение

[WHERE условие]

### Оператор UPDATE

Ключевое слово **UPDATE** предшествует названию таблицы или курсора (виртуальной таблицы). Как и в случае с другими операторами модификации данных, в каждом операторе **UPDATЕ** можно изменять данные только одной таблицы.

Если при выполнении оператора **UPDATE** нарушаются ограничения на целостность (например, добавляемое значение имеет неверный тип), система запрещает обновление данных и обычно выдает сообщение об ошибке.

### Предложение SET

В предложении **SET** определяются столбцы и указываются их новые значения. В предложении **WHERE** можно указать изменяемые строки. Если предложение **WHERE** не используется, то при выполнении команды **UPDATE** будут изменены значения во всех строках столбцов, указанных в предложении **SET**.

Например, таблица *publishers* имеет следующий вид:

select \*

from publishers

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **pub\_id** | **pub\_name** | **address** | **city** | **state** |
| 0736 | New Age Books | 1 1st St | Boston | MA |
| 0877 | Binnet & Hardley | 2 2nd Ave. | Washington | DC |
| 1389 | Algodata Infosystems | 3 3rd Dr. | Berkeley | CA |
| 0010 | Pragmatics | 4 4th Ln. | Chicago | IL |
| 1756 | HealtText | NULL | NULL | NU |

Если, например, все филиалы разместят свои центральные офисы в Атланте, штат Джорджия, в таблицу потребуется внести изменения:

update publishers

set city = 'Atlanta', state = 'GA'

Кроме того, при обновлении данных со значениями столбцов можно выполнять математические преобразования. Чтобы увеличить в два раза все цены в таблице *titles*, можно воспользоваться оператором:

update titles

set price = price \* 2

Поскольку в этом операторе не используется предложение **WHERE**, будут изменены цены во всех строках таблицы.

### Предложение WHERE

Предложение **WHERE** в операторе **UPDATE** определяет изменяемые строки. Например, если вдруг северная Калифорния станет отдельным штатом с названием Pacifica (сокращенно PC), а жители Окланда решат переименовать свой город в Big Bad Bay City, то в таблицу *authors* потребуется внести следующие изменения:

update authors

set state = 'PG', city = 'Big Bad Bay City'

where state = 'CA' and city = 'Oakland'

Кроме того, потребуется выполнить дополнительные операторы **UPDATE** для авторов из других городов северной Калифорнии.

Предложение **WHERE** в операторе **UPDATE** может включать в себя подзапросы к одной или нескольким другим таблицам.

## Удаление данных: Команда DELETE

Не менее важной, чем добавление и изменение строк, является возможность их удаления. Подобно **INSERT** и **UPDATE**, команда **DELETE** позволяет манипулировать одной или несколькими строками. Так же, как и в случае с другими операторами модификации данных, при удалении строк можно пользоваться информацией из других таблиц.

Оператор **DELETE** имеет следующий синтаксис:

DELETE FROM имя\_таблицы

WHERE условие

В предложении **WHERE** определяются подлежащие удалению строки. Для удаления строки из таблицы *publishers* (издатель с идентификационном номером 1622) можно использовать следующую команду:

delete from publishers

where pub\_id = '1622'

После удаления строки, описывающей этого издателя, с помощью объединения таблиц *publishers* и *titles* по столбцу с идентификационными номерами уже не удастся найти названия изданных им книг.

В случае отсутствия предложения **WHERE** в операторе **DELETE** будут удалены все строки таблицы.

# Выборка информации из БД

## Синтаксис оператора SELECT

Все дальнейшие усложнения начинаются со следующей конструкции оператора **SELECT**:

SELECT список\_столбцов

FROM таблица[ы]

[WHERE условия]

select address

from personnel

where name = 'Richard Roe'

|  |  |
| --- | --- |
| **Name** | **Address** |
| Jane Doe | 127 Elm St. |
| Richard Roe | 10 Trenholm Place |
| Edgar Poe | 1533 User House |

Полный синтаксис оператора **SELECT** имеет следующий вид:

SELECT [ALL | DISTINCT] список\_выбора

FROM {имя\_таблицы | имя\_курсора}

[, {имя\_таблицы | имя\_курсора} ]…

[WHERE условия]

[GROUP BY имя\_столбца [ , имя\_столбца]…]

[HAVING условия]

[ORDER BY {имя\_столбца | список\_выбора} [ASC | DESC]

[ , {column\_name | список\_выбора} [ASC | DESC] ]… ]

Хотя SQL и является языком с необусловленной формой, порядок предложений в операторе **SELECT** должен строго соблюдаться (например, предложение **GROUP BY** должно предшествовать предложению **ORDER BY**). В противном случае это приведет к появлению ошибки синтаксиса.

Помимо этого, нужно однозначно описывать все объекты БД. Например, если в БД содержится несколько столбцов с именем *notes*, нужно точно указать, какой из них имеется ввиду. Для этого в соответствующий запрос нужно включить имя базы данных, таблицы или курсора и имя владельца, например:

база\_данных.владелец.имя\_таблицы.notes

база\_данных.владелец.имя\_курсора.notes

Все примеры запросов в этой главе используют по одной таблице, поэтому такие уточнения не потребуются. О них также не упоминается в большинстве книг, статей и руководств по SQL, так как более краткая форма упрощает понимание операторов **SELECT**. Тем не менее необходимо включать их в запросы при необходимости.

## Выбор столбцов: Список выбора

Любой оператор **SELECT** начинается с ключевого слова **SELECT**. Ключевые слова **ALL** и **DISTINCT**, которые определяют, будут ли включаться в результат повторяющиеся строки, являются необязательными и описываются в следующей главе.

В **списке выбора** (select list) определяется столбец или столбцы, включаемые в результат. Он может состоять из имен одного или нескольких столбцов, или включать единственную звездочку (\*), определяющую все столбцы. Можно также использовать выражения: константы, имена столбцов, функций и их комбинации с арифметическими операторами и скобками. Вот несколько примеров выражений:

ytd\_sales \* price

price \* 1.2

(12000 – 500) / 13

avg (advance)

Каждый элемент в списке выбора отделяется от другого запятой.

### Выбор всех столбцов: SELECT \*

Знак звездочки (\*) означает выбор *всех имен столбцов во всех таблицах* из списка таблиц.

Для выбора всех столбцов таблиц используется следующий оператор:

SELECT \*

FROM список\_таблиц

Так как оператор **SELECT** \* находит все текущие столбцы таблицы, при изменении структуры таблицы (добавление, удаление или переименование столбцов) соответственно изменяются и его результаты. Оператор **SELECT** \* наиболее подходит для таблиц с несколькими столбцами, так как одновременное отображение данных из большого количества столбцов обычно неудобно. Также этот оператор будет полезен в случае желания получить общие представления о структуре таблицы (о столбцах и их порядке).

В некоторых системах в списке выбора могут одновременно присутствовать и звездочка, и имена отдельных столбцов. Это наиболее удобно в многотабличных запросах, когда вместе со звездочкой используется имя таблицы. Следующий оператор извлекает информацию из таблицы *publishers* для каждого значения *title\_id* из таблицы *titles*. При этом будут извлекаться все столбцы из таблицы *publishers* (так как в списке выбора присутствует *publishers.\**) и только один столбец из таблицы *titles* (так как в списке выбора присутствует *title\_id*).

select title\_id, publishers.\*

from titles, publishers

where title\_id = publishers.pub\_id

Без использования звездочки этот запрос имел бы следующий вид:

select title\_id, publishers.pub\_id, publishers.pub\_name, publishers.address,

publishers.city, publishers.state

from titles, publishers

where title\_id = publishers.pub\_id

### Выбор отдельных столбцов

Для выбора подмножества столбцов таблицы нужно просто перечислить их в списке выбора, как это делалось в предыдущем примере:

SELECT имя\_столбца [ , имя\_столбца]…

FROM имя\_таблицы

### Выражения: больше, чем просто имена столбцов

До сих пор с помощью операторов **SELECT** отображались только содержащиеся в таблицах данные. Это полезно, но не всегда достаточно. SQL позволяет манипулировать результатами, делая их вполне понятными. В списке выбора можно использовать символьные строки, математические действия и функции, реализованные в системе.

***Переименование столбцов и задание имен выражениям.*** При выводе результатов запроса каждый столбец по умолчанию получает заголовок, совпадающий с его именем в БД. Столбцы в БД обычно имеют сокращенные названия (чтобы их легче было набирать) и могут быть непонятны пользователям, незнакомым с используемым в БД жаргоном.

Для упрощения чтения и понимания результатов запроса можно переопределить заголовки столбцов. Чтобы получить необходимые имена заголовков, просто нужно ввести имя\_столбца, имя\_заголовка или имя\_столбца as имя\_заголовка в списке выбора вместо обычных имен столбцов. (В некоторых системах используется другая форма записи в виде заголовок\_столбца = имя\_столбца).

Например, для изменения заголовка *pub\_name* на *Publisher* следует попробовать выполнить один из следующих операторов:

select pub\_name Publisher, pub\_id

from publishers

select pub\_name as Publisher, pub\_id

from publishers

В результате выполнения этого запроса изменится заголовок столбца:

|  |  |
| --- | --- |
| **Publisher** | **pub\_id** |
| New Age Books | 0736 |
| Binnet & Hardley | 0877 |
| Algodata Infosystems | 1389 |

Размер заголовка не ограничивается размером данных соответствующего столбца. Например, столбец *pub\_id* может иметь заголовок более чем из четырех символов. Вот что получится, если изменить заголовок этого столбца на *Identification#*.

select pub\_name as Publisher, pub\_id as Identification#

from publishers

Результат:

|  |  |
| --- | --- |
| **Publisher** | **Identification#** |
| New Age Books | 0736 |
| Binnet & Hardley | 0877 |
| Algodata Infosystems | 1389 |

Как правило, в заголовках нельзя использовать кавычки и пробелы. Заголовки можно создавать и для столбцов, использующих различные вычисления и выражения, например, *New\_price, Double\_Advance* и т.д.:

select title, advance\*2 as Doouble\_Advance

from titles

***Символьные строки в результатах запросов.*** Иногда, чтобы сделать более понятными результаты запроса, имеет смысл добавить к ним небольшие пояснения. Здесь на помощь приходят символьные строки.

Например, перед именами издателей можно добавить текст типа "The publisher's name is" (Имя издателя). Для этого данную строку нужно вставить в список выбора, взять в двойные или одинарные кавычки, чтобы система не посчитала ее названием столбца, и отделить от других элементов списка выбора запятыми.

Если в строке имеется апостроф, нужно строго следовать правилам конкретной системы. Например, чтобы апостроф не рассматривался системой как закрывающаяся кавычка, в нашем случае используется вторая одинарная кавычка.

select 'The publisher' 's name is', pub\_name as Publisher

from publishers

Результат:

|  |  |
| --- | --- |
|  | **Publisher** |
| The publisher's name is | New Age Books |
| The publisher's name is | Binnet & Hardley |
| The publisher's name is | Algodata Infosystems |

В результате выполнения этого запроса создается новый столбец, однако отображаемые на экране изменения не влияют на физическую структуру БД.

Кроме того, каждое слово символьной строки можно превратить в отдельное поле:

select 'The', 'publisher', 'name', 'is', pub\_name

from publishers

Результат:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | **pub\_name** |
| The | publisher's | name | is | New Age Books |
| The | publisher's | name | is | Binnet & Hardley |
| The | publisher's | name | is | Algodata Infosystems |

С помощью такого подхода можно комбинировать столбцы и текст, например:

select 'The name for publisher #', pub\_id, 'is', pub\_name

from publishers

Результат:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | **pub\_id** |  | **pub\_name** |
| The | name | for | publisher# | 0736 | is | New Age Books |
| The | name | for | publisher# | 0877 | Is | Binnet & Hardley |
| The | name | for | publisher# | 1389 | Is | Algodata Infosystems |

***Вычисления с использованием констант.*** В списке выбора с числовыми данными и константами можно выполнять арифметические действия.

Ниже приводится список допустимых арифметических операторов:

***Символ*** ***Операция***

+ сложение

- вычитание

/ деление

\* умножение

Арифметические операторы – сложение, вычитание, деление и умножение – могут применяться к любым числовым столбцам. В некоторых системах добавляется еще одна операция – **остаток от деления** (modulo), представляемая символом %. Результатом выполнения этой операции является остаток от деления двух целых чисел. Например, 21%9 = 3. Если в системе реализованы функции обработки данных, то некоторые арифметические операции могут выполняться и на нецифровых столбцах.

В списке выбора эти операции могут использоваться в любых комбинациях с именами столбцов и числовыми константами. Например, чтобы отобразить в таблице *titles* повышение стоимости всех книг на 100%, можно выполнить следующий запрос:

select title\_id, ytd\_sales, ytd\_sales \* 2

from titles

В результате получится следующая таблица:

|  |  |  |
| --- | --- | --- |
| **title\_id** | **ytd\_sales** |  |
| PC8888 | 4095 | 8190 |
| BU1032 | 4095 | 8190 |
| PS7777 | 3336 | 6672 |

Следует обратить внимание на нулевые значения в *ytd\_sales* и производном столбце. Любые арифметические операции над нулевыми значениями в результате дают NULL. Производному столбцу можно задать заголовок (например, *Projected\_Sales*):

select title\_id, ytd\_sales, ytd\_sales \* 2 as Projected\_Sales

from titles

Иногда, как это было в предыдущем примере, в результат включаются данные как исходного, так и производного столбца. Однако включать в список выбора столбец, на основе которого выполнялись вычисления, необязательно. Чтобы увидеть только вычисленные значения, можно выполнить следующую команду:

select title\_id, ytd\_sales \* 2

from titles

***Вычисления с использованием имен столбцов.*** Арифметические операции также можно выполнять на данными двух и более столбцов. Например:

select title\_id, ytd\_sales \* price

from titles

|  |  |
| --- | --- |
| **title\_id** |  |
| PC8888 | 81900.0000 |
| BU1032 | 81859.0500 |
| PS7777 | 26654.6400 |

Наконец, можно получать требуемые значения на основе столбцов из разных таблиц.

***Порядок выполнения арифметических операторов.*** Если в выражении имеется несколько арифметических операторов, они будут выполняться в порядке, установленном в системе. В соответствии с общепринятыми правилами сначала выполняется умножение и деление, затем вычитание и сложение. Если в выражении присутствует несколько операторов с одинаковым приоритетом, они выполняются справа налево. Максимальный приоритет имеют выражения, взятые в скобки.

Рассмотрим пример. В следующем операторе **SELECT** сначала вычисляется произведение *ydt\_sales* и *price*, так как операция умножения имеет более высокий приоритет, чем операция вычитания.

select title\_id, ytd\_sales \* price - advance

from titles

Во избежание недоразумений рекомендуется использование скобок. Например:

select title\_id, (ytd\_sales \* price) - advance

from titles

Скобки также используются для изменения порядка выполнения операций, поскольку заключенные в них операторы выполняются в первую очередь. В случае вложенных скобок (одна пара скобок внутри другой пары) первыми выполняются операторы, имеющие максимальную глубину вложения. Например, результаты предыдущего запроса изменятся, если переставить скобки таким образом, чтобы сначала выполнялась операция вычитания:

select title\_id, ytd\_sales \* (price - advance)

from titles

## Указание таблиц: список таблиц

**В списке таблиц** (table list) указываются имена таблиц и курсоров, содержащих столбцы из списка выбора и предложения **WHERE**. Имена таблиц в списке разделяются запятыми. Ниже приведен общий вид предложения **FROM**:

SELECT список\_выбора

FROM [квалификатор] {имя\_таблицы | имя\_курсора}

[ , [квалификатор] {имя\_таблицы | имя\_курсора}]…

В списке таблиц также можно указать соответствующие таблицам БД и их владельцев. Необходимость в этом может возникнуть, если в запросе используются одноименные таблицы из разных БД.

В большинстве диалектов SQL для упрощения набора таблицам позволяется задавать **псевдонимы** (aliases). Псевдоним указывается после имени таблицы в списке таблиц:

select p.pub\_id, p.pub\_name

from publishers p

Буква *p* перед именами столбцов заменяет полное имя таблицы (*publishers*). Этот запрос эквивалентен следующему:

select publishers.pub\_id, publishers.pub\_name

from publishers

Так как в каждом запросе используется только одна таблица, и при использовании столбца *pub\_id* не возникает никакой неопределенности, имя таблицы можно опустить. Псевдонимы действительно полезны в многотабличных запросах, когда нужно различать одноименные столбцы из разных таблиц.

## Выбор строк: предложение WHERE

Предложение **WHERE** является частью оператора **SELECT** и позволяет определить условия для выборки строк. Общий формат оператора имеет следующий вид:

SELECT список\_выбора

FROM список\_таблиц

WHERE условия

При выполнении оператора **SELECT** с предложением **WHERE** выбираются все строки, удовлетворяющие наложенному условию.

В SQL имеется целый ряд операторов и ключевых слов для задания условий.

* Операторы сравнения (=, <, >, и т.д.)  
  where advance \* 2 > ytd\_sales \* price
* Комбинации условий и логическое отрицание (AND, OR, NOT)  
  where advance < 5000 or ytd\_sales > 2000
* Диапазоны (BETWEEN и NOT BETWEEN)  
  where ytd\_sales between 4095 and 12000
* Списки (IN, NOT IN)  
  where state in ('CA', 'WH', 'MD')
* Неизвестные значения (IS NULL и IS NOT NULL)  
  where advance is null
* Соответствия символов (LIKE и NOT LIKE)  
  where phone not like '415%'

Все эти ключевые слова будут описаны далее.

Кроме того, в предложение **WHERE** могут включаться условия на объединение и подзапросы.

### Операторы сравнения

Часто в приложениях возникает необходимость провести сравнительный анализ некоторых значений из БД: выяснить, какое из них больше или меньше (в числовом или алфавитном порядке). Для этих целей в SQL предусмотрен целый ряд операторов сравнения:

**Оператор Значение**

= Равно

> Больше

< Меньше

>= Больше или равно

<= Меньше или равно

!= Не равно

<> Не равно

Эти операторы используются следующим образом:

WHERE выражение оператор\_сравнения выражение

В качестве выражений могут использоваться константы, имена столбцов, функции, подзапросы или их комбинации, связанные с арифметическими операторами.

Обычно операторы сравнения применяются к числовым значениям. В SQL они могут также применяться к данным с типами *char* и varchar и к датам. При использовании символьных значений и дат в SQL их нужно заключать в кавычки.

Порядок следования строчных, прописных и специальных символов определяется используемой кодовой таблицей символов и может отличаться в разных системах. В большинстве систем при сравнении игнорируются все концевые пробелы. Например, значения "Dirk" и "Dirk " считаются одинаковыми. На примере следующих операторов **SELECT** можно увидеть, как работают операторы сравнения. Первый запрос используется для поиска книг стоимостью выше $15:

select title, price

from titles

where price > $15.00

Результат:

|  |  |
| --- | --- |
| **title** | **price** |
| Secrets of Silicon Valley | 20 |
| The Busy Executive's Database Guide | 19.99 |
| Prolonged Data Deprivation: Four Case Studies | 19.99 |
| Silicon Valley Gastronomic Treats | 19.99 |

В результате выполнения второго запроса находятся авторы с фамилиями, идущими в алфавитном порядке после фамилии McBadden:

select au\_fname, au\_lname

from authors

where au\_lname > 'McBadden'

Результат:

|  |  |
| --- | --- |
| **au\_lname** | **au\_fname** |
| Ringer | Albert |
| Ringer | Anne |
| Panteley | Sylvia |
| Stringer | Dirk |
| Straight | Dick |

Полученные результаты могут отличаться от приведенных выше, в зависимости от метода сортировки, используемого в системе.

Следующий запрос используется для вывода гипотетической информации – удвоенной стоимости книг, затраты на которые превысили $10000 вместе с их идентификационными номерами:

select title\_id, price \* 2

from titles

where advance > $10000

Результат:

|  |  |
| --- | --- |
| **title\_id** |  |
| BU2075 | 5.98 |
| MC3021 | 5.98 |

Вот пример запроса, использующего оператор сравнения "не равен" для поиска телефонных номеров авторов, живущих за пределами Калифорнии (в разных версиях SQL для записи этого оператора могут применяться знаки != или <>):

select au\_id, phone

from authors

where state != 'CA'

Результат:

|  |  |
| --- | --- |
| **au\_id** | **phone** |
| 998-72-3567 | 801 826-0752 |
| 899-46-2035 | 801 826-0752 |
| 722-51-5454 | 219 547-9982 |

### Совместное использование условных и логических операторов

Если в предложение **WHERE** нужно поместить несколько условий, то для их соединения можно использовать **логические операторы** (logical operators) AND, OR и NOT, называемые также **булевыми операторами**.

Оператор AND объединяет два и более условий и возвращает истинное значение только при выполнении всех условий. Например, следующий запрос находит только авторов с фамилией Ringer и именем Anne. Albert Ringer в этот список не попадет.

select \*

from authors

where au\_lname = 'Ringer' AND au\_fname = 'Anne'

В следующем примере находятся все книги по бизнесу стоимостью выше $10 и затратами ниже $20000:

select title, type, price, advance

from titles

where type = 'business'

and price > $10.00

and advance < $20000

Результат:

|  |  |  |  |
| --- | --- | --- | --- |
| **title** | **type** | **price** | **advance** |
| The Busy Executive's Database Guide | business | 19.99 | 5000 |
| Cooking with Computers | business | 11.95 | 5000 |
| Straight Talk About Computers | business | 19.99 | 5000 |

Оператор OR также связывает два или больше условий, но возвращает истинный результат при выполнении хотя бы одного условия. Следующий запрос предназначен для поиска строк, содержащих в столбце *au\_fname* значения Anne или Ann:

select au\_id, au\_lname, au\_fname

from authors

where au\_fname = 'Anne'

OR au\_fname = 'Ann'

Результат:

|  |  |  |
| --- | --- | --- |
| **au\_id** | **au\_lname** | **au\_name** |
| 899-46-2035 | Ringer | Anne |
| 427-17-2319 | Dull | Ann |

В результате выполнения следующего запроса находятся все книги стоимостью выше $20 и затратами меньше $5000.

select title, type, price, advance

from titles

where price > $20.00

OR advance < $5000

Результат:

|  |  |  |  |
| --- | --- | --- | --- |
| **title** | **type** | **price** | **advance** |
| Emotional Security: A New Algorithm | psychology | 7.99 | 4000 |
| Prolonged Data Deprivation | psychology | 19.99 | 2000 |
| Silicon Valley Gastronomic Treats | mod\_cook | 19.99 | 0 |

Следующий пример отражает потенциальные проблемы, которые могут возникнуть при использовании оператора OR. Предположим, что нужно найти все книги по бизнесу, а также все книги стоимостью $10 и все книги с затратами меньше $20000. В обычном языке для описания этого набора используется связка AND (И), тогда как в SQL нужно использовать оператор OR, потому что нужно найти книги из всех трех категорий, а не только книги, удовлетворяющие всем этим условиям одновременно. Ниже приведен соответствующий оператор SQL:

select title, type, price advance

from titles

where type = 'business'

or price > $10.00

or advance < $20000

Результат:

|  |  |  |  |
| --- | --- | --- | --- |
| **title** | **type** | **price** | **advance** |
| Secrets of Silicon Valley | popular\_comp | 20 | 8000 |
| The Busy Executive's Database Guide | business | 19.99 | 5000 |
| Emotional Security: A New Algorithm | psychology | 7.99 | 4000 |
| Prolonged Data Deprivation | psychology | 19.99 | 2000 |

Логический оператор NOT используется для построения отрицаний и помещается перед самим выражением. Следующие два запроса эквивалентны:

select au\_lname, au\_fname, state

from authors

where NOT state = 'CA'

select au\_lname, au\_fname, state

from authors

where state != 'CA'

В результате получится следующий список:

|  |  |  |
| --- | --- | --- |
| **au\_lname** | **au\_fname** | **state** |
| Ringer | Albert | UT |
| Ringer | Anne | UT |
| DeFrance | Michel | IN |
| Panteley | Sylvia | MD |

#### Порядок выполнения логических операторов

Аналогично операторам сравнения, логические операторы выполняются в строго определенной последовательности. Если в выражении присутствуют оба типа операторов, первыми выполняются арифметические операторы. Если в выражении используется несколько логических операторов, они выполняются в следующей последовательности: сначала NOT, затем AND и наконец OR.

Скобки – Умножение/Деление – Вычитание/Сложение – NOT – AND – OR

Рассмотрим несколько примеров. Следующий запрос используется для поиска всех книг по бизнесу в таблице *titles*, независимо от величины затрат на них, а также книги по психологии с затратами больше $5500. Ограничение на затраты применяется только к книгам по психологии, так как оператор AND выполняется перед оператором OR.

select title\_id, type, advance

from titles

where type = 'business'

or type = 'psychology'

and advance > $5500

Результат:

|  |  |  |
| --- | --- | --- |
| **title\_id** | **type** | **advance** |
| BU1032 | business | 5000 |
| BU1111 | business | 5000 |
| BU2075 | business | 10125 |
| PS2106 | psychology | 6000 |

Поскольку логические операторы выполняются в определенной последовательности, в результирующий список будут включены три книги по бизнесу с расходами меньше $5500. Другими словами, сначала будет найдены все книги по психологии с расходами больше $5500, а затем все без ограничения книги по бизнесу.

Чтобы первым выполнялся оператор OR, в запросе нужно использовать скобки. В результате выполнения следующего запроса будут найдены все книги по бизнесу и психологии с затратами выше $5500:

select title\_id, type, advance

from titles

where (type = 'business'

OR type = 'psychology')

AND advance > $5500

Результат:

|  |  |  |
| --- | --- | --- |
| **title\_id** | **type** | **advance** |
| BU2075 | business | 10125 |
| PS2106 | psychology | 6000 |
| PS1372 | psychology | 7000 |

Благодаря скобкам сначала осуществляется поиск всех книг по бизнесу и психологии, а затем из них выбираются книги с затратами больше $5500.

Ниже приводится пример запроса, в котором одновременно используются арифметические, логические и операторы сравнения. Этот запрос используется для поиска книг, затраты на которые не окупились. Другими словами, находятся все книги, доход от продажи которых (т.е. произведение *ytd\_sales* на *price*) меньше удвоенной суммы, выплаченной авторам. Кроме того, в нашем случае за точку отсчета берется 15.10.1985 с целью фиксации достаточного срока, в течение которого должен был разойтись тираж. Это условие присоединяется к запросу с помощью оператора AND и в соответствии с правилами вычисляется после арифметических операций.

select title\_id, type, price, advance, ytd\_sales

from titles

where price \* ytd\_sales < 2 \* advance

AND pubdate < '10/15/85'

Результат:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **title\_id** | **type** | **price** | **advance** | **ytd\_sales** |
| PS2106 | psychology | 7 | 6000 | 111 |

### Диапазоны (BETWEEN и NOT BETWEEN)

Другим средством реализации условий являются диапазоны. Определить диапазон можно двумя способами:

* с помощью операторов сравнения > и <;
* с помощью ключевого слова **BETWEEN**.

Ключевое слово **BETWEEN** можно использовать для задания **включающего диапазона** (inclusive range), если в искомые значения должны включаться и границы диапазона. Например, для поиска всех книг с количеством проданных экземпляров между 4095 и 12000 (включительно) можно использовать следующий запрос:

select title\_id, ytd\_sales

from titles

where ytd\_sales BETWEEN 4095 AND 12000

Результат:

|  |  |
| --- | --- |
| **title\_id** | **ytd\_sales** |
| PC8888 | 4095 |
| BU1032 | 4095 |
| TC7777 | 4095 |
| PC1035 | 8780 |
| BU7832 | 4095 |

Нужно обратить внимание, что в результат включаются книги с 4095 проданными экземплярами. Также туда были бы включены и книги с 12000 проданными экземплярами, если бы в таблице имелись соответствующие данные. В этом смысле диапазон **BETWEEN** отличается от диапазона больше-меньше (> <). При выполнении предыдущего запроса с использованием диапазона больше-меньше получаются другие результаты, так как при поиске не учитываются границы диапазона:

select title\_id, ytd\_sales

from titles

where ytd sales > 4095 AND ytd\_sales < 12000

Результат:

|  |  |
| --- | --- |
| **title\_id** | **ytd\_sales** |
| PC1035 | 8780 |

При использовании конструкции **NOT BETWEEN** находятся все строки, не входящие по своим значениям в указанный диапазон. Так, для поиска всех книг с количеством проданных экземпляров вне диапазона от 4095 до 12000 можно использовать следующий запрос:

select title\_id, ytd\_sales

from titles

where ytd\_sales NOT BETWEEN 4095 AND 12000

|  |  |
| --- | --- |
| **title\_id** | **ytd\_sales** |
| PS7777 | 3336 |
| PS3333 | 4072 |
| BU1111 | 3876 |
| MC2222 | 2032 |
| TC4203 | 15096 |

Тот же результат можно получить с помощью операторов сравнения, однако нужно обратить внимание, что в следующем запросе вместо оператора AND используется оператор OR:

select title\_id, ytd\_sales

from titles

where ytd\_sales < 4095 or ytd\_sales > 12000

Результат:

|  |  |
| --- | --- |
| **title\_id** | **ytd\_sales** |
| PS7777 | 3336 |
| PS3333 | 4072 |
| BU1111 | 3876 |
| MC2222 | 2032 |
| TC4203 | 15096 |

Это еще один пример, когда могут возникнуть определенные затруднения. На обычном языке суть запроса формируется следующим образом: нужно найти все книги с количеством проданных экземпляров до 4095 и все книги с количеством проданных экземпляров больше 12000. Хотя мы говорим "и", в запросе нужно использовать оператор OR (или). Если вместо него применить оператор AND, то в результате не будет найдена ни одна книга, так как не может быть продано экземпляров одной книги меньше 4095 и одновременно больше 12000.

### Списки (IN и NOT IN)

Ключевое слово **IN** позволяет выбрать значения, совпадающие со значениями из заданного списка. Например, без использования **IN** для получения списка для получения списка авторов, проживающих в Калифорнии, Индиане и Мериленде, нужно было бы выполнить следующий запрос:

select au\_lname, state

from authors

where state = 'CA' OR state = 'IN' OR state = 'MD'

Этот же результат можно получить с использованием ключевого слова **IN**. Значения, идущие после **IN**, должны браться в кавычки, разделяться запятыми и заключаться в скобки.

select au\_lname, state

from authors

where state IN('CA', 'IN', 'MD')

Оба предыдущих запроса дают одинаковый результат:

|  |  |
| --- | --- |
| **au\_lname** | **state** |
| Bennet | CA |
| Green | CA |
| Carson | IN |
| Panteley | MD |

На следующем примере можно понять, как в подзапросах можно использовать ключевое слово **IN**. Предположим, нужно найти имена авторов, которые получают меньше 50% от суммарного гонорара за книги, написанные ими в соавторстве. Имена авторов содержатся в таблице *authors*, информация о распределении гонораров хранится в таблице *titleauthors*. Ключевое слово **IN** позволяет как бы объединить обе таблицы (без получения общей таблицы) и извлечь при этом нужную информацию.

select au\_lname, au\_fname

from authors

where au\_id IN

(select au\_id

from titleauthors

where royaltyshare < .50)

Результат:

|  |  |
| --- | --- |
| **au\_lname** | **au\_fname** |
| Green | Marjorie |
| Ringer | Anne |
| MacFeather | Stearns |
| Yokomoto | Akiko |
| O'Leary | Michael |

Таким образом, шесть авторов хотя бы за одну книгу получили меньше 50% от общего гонорара.

Конструкция **NOT IN** позволяет найти авторов, не удовлетворяющих условиям, перечисленным в списке. С помощью следующего запроса можно найти всех авторов, которые хотя бы за одну книгу получили больше половины от общего гонорара.

select au\_lname, au\_fname

from authors

where au\_id NOT IN

(select au\_id

from titleauthors

where royaltyshare < .50)

Результат:

|  |  |
| --- | --- |
| **au\_lname** | **au\_fname** |
| Bennet | Abraham |
| Carson | Cheryl |
| Ringer | Albert |
| DeFrance | Michel |
| Panteley | Sylvia |

### Выборка нулевых значений

Значение NULL используется для представления неизвестной информации. И это не то же самое, что нуль или пустая позиция.

Чтобы прояснить эти различия, рассмотрим следующий листинг, в котором представлены названия и расходы по книгам, принадлежащим одному издателю.

Результат:

|  |  |
| --- | --- |
| **title** | **advance** |
| Silicon Valley Gastronomic Treats | 0 |
| Sushi, Anyone? | 8000 |
| Fifty Years in Buckingham Palace | 4000 |
| The Psychology of Computer Cooking | NULL |
| The Gourmet Microwave | 15000 |

Расходы на книгу The Psychology of Computer Cooking выражается нулевым значением (NULL). Возможно, автор и издатель еще не полностью договорились о характере своих отношений либо эти данные просто забыли ввести в БД. Пока сумма расходов не определена, она будет выражаться значением NULL.

Что произойдет, если операция сравнения будет применена к нулевому значению? Так как нуль (NULL) представляет неизвестное значение, его нельзя сравнить с любым другим значением, даже с другим нулевым значением. Например, в результаты запроса, который ищет все идентификаторы книг с расходами ниже $5000, не будет включена книга MC3026, The Psychology of Computer Cooking.

select title\_id, advance

from titles

where advance < $5000

Результат:

|  |  |
| --- | --- |
| **title\_id** | **advanc**e |
| PS7777 | 4000 |
| PS3333 | 2000 |
| MC2222 | 0 |
| TC4203 | 4000 |
| PS2091 | 2275 |

Эта книга не будет найдена и при поиске книг с расходами выше $5000:

select title\_id, advance

from titles

where advance > $5000

Результат:

|  |  |
| --- | --- |
| **title\_id** | **advanc**e |
| PC8888 | 8000 |
| TC7777 | 8000 |
| PC1035 | 7000 |
| BU2075 | 10125 |
| PS2106 | 6000 |

Значение NULL не меньше, не больше и не равно $5000, потому что является неизвестным. С помощью специального условия из таблицы всегда можно извлечь строки с нулевыми значениями:

WHERE имя\_столбца IS [NOT] NULL

Чтобы найти все книги с нулевыми запросами, можно использовать следующий запрос:

select title\_id, advance

from titles

where advance is NULL

Результат:

|  |  |
| --- | --- |
| **title\_id** | **advanc**e |
| MC3026 | NULL |
| PC9999 | NULL |

Это условие можно использовать совместно с другими операторами. Например, в результате выполнения следующего запроса находятся все книги, расходы на которые были либо нулевыми (NULL), либо составляли меньше $5000:

select title\_id, advance

from titles

where advance < $5000

or advance is NULL

Результат:

|  |  |
| --- | --- |
| **title\_id** | **advanc**e |
| PS7777 | 4000 |
| PS3333 | 2000 |
| MC2222 | 0 |
| TC4203 | 4000 |
| MC3026 | NULL |
| PS2091 | 2275 |
| PC9999 | NULL |

### Поиск по подстрокам: предложение LIKE

Некоторые задачи нельзя решить только с помощью операторов сравнения. Вот несколько примеров:

* "Его фамилия начинается с Мс или Мас, а дальше я не помню".
* "Мне нужен список телефонов, начинающихся с 415".
* "Я забыл название книги, но помню, что в ее описании упоминалось слово *exercise*".
* "Я не помню, как точно пишется его фамилия – Carson или Karsen".
* "Его имя Dirk или Dick. Четыре буквы, начинается с D и заканчивается на k".

В каждом из этих случаев известен некоторый образец, где-то встречающийся в столбце, и с его помощью нужно извлечь всю или часть соответствующей ему строки. Для решения этой задачи и предназначено ключевое слово **LIKE**. Его можно применять к символьным полям, а в некоторых системах и к полям даты. Однако ключевое слово **LIKE** не работает с числовыми полями (целыми, денежными, десятичными и плавающими). Его синтаксис имеет следующий вид:

WHERE имя\_столбца [NOT] LIKE 'образец' [ESCAPE ключевой\_символ]

Образец должен заключаться в кавычки и может содержать один или несколько **шаблонов** (wildcards) – символов, замещающих в образце пропущенные буквы или строки. Ключевое слово **ESCAPE** используется в том случае, если в образце содержится один из шаблонов, но его нужно рассматривать как простой литерал.

ANSI SQL предусматривает использование двух символов шаблона совместно с ключевым словом **LIKE** – знак процента (%) и подчеркивание (\_).

***Шаблон Значение***

% Любая строка с любым количеством символов

\_ Любой одиночный символ

В разных системах могут использоваться разные символы шаблонов.

Теперь вернемся к сформулированным выше задачам и приведем соответствующие запросы. Сначала осуществим поиск по шотландским фамилиям:

select au\_lname, city

from authors

where au\_lname LIKE 'Mc%' OR au\_lname like 'Mac%'

Результат:

|  |  |
| --- | --- |
| **au\_lname** | **city** |
| MacFeather | Oakland |
| McBadden | Vacaville |

В соответствии с образцами, приведенными после ключевого слова **LIKE**, система найдет все фамилии, начинающиеся с Mc и Mac. Нужно обратить внимание на то, что шаблон находится внутри кавычек.

С помощью следующего запроса можно получить список всех телефонов, начинающихся с 415:

select au\_lname, phone

from authors

where phone LIKE '415%'

Результат:

|  |  |
| --- | --- |
| **au\_lname** | **phone** |
| Bennet | 415 658-9932 |
| Green | 415 986-7020 |
| Carson | 415 548-7723 |
| Stringer | 415 843-2991 |

Найти книгу по слову в ее описании несколько сложнее. Неизвестно, где в описании находится слово *exercise* – в начале или в конце. Кроме того, неизвестно, с какой буквы оно начинается – строчной или прописной. Все эти возможности можно предусмотреть, использовав шаблоны в начале и в конце образца.

select title\_id, notes

from titles

where notes like '%xercise%'

Результат:

|  |  |
| --- | --- |
| **title\_id** | **notes** |
| PS2106 | New exercise, meditation, and nutritional techniques that can reduce the shock of daily interactions. |

Если известно точное количество пропущенных символов, можно использовать односимвольный шаблон (\_). В случае с фамилиями, первая буква либо К, либо С, а предпоследняя – е или о.

select au\_lname, city

from authors

where au\_lname LIKE '\_ars\_n'

Результат:

|  |  |
| --- | --- |
| **au\_lname** | **city** |
| Carson | Berkeley |
| Karsen | Oakland |

Вместе с шаблонами можно использовать и конструкцию **NOT LIKE**. Чтобы найти в таблице *authors* все номера телефонов, которые не начинаются с 415, можно воспользоваться одним из следующих запросов (они эквивалентны):

select phone

from authors

where phone NOT LIKE '415%'

select phone

from authors

where NOT phone LIKE '415%'

Символы шаблонов обычно используются вместе с ключевым словом **LIKE**. Без него они интерпретируются как обычные символы. В результате выполнения приведенного ниже запроса будут найдены телефоны, состоящие только из четырех символов 415%.

select phone

from authors

where phone = '415%'

А что, если нужно найти значение, которое само содержит один из символов шаблона? Например, в таблице *titles* в столбце *notes* одной из строк используется знак процента. Для поиска по знаку процента нужно использовать ключевое слово **ESCAPE**, позволяющее трактовать его как обычный символ. Шаблон, следующий непосредственно после ключевого символа, рассматривается в качестве простого литерала. Все последующие символы шаблонов имеют свое обычное значение. С помощью следующего выражения **LIKE** в столбце *notes* находятся строки, содержащие знак процента. Так как он, скорее всего, не является первым или последним символом строки, в запросе используются шаблоны в начале и в конце образца, а также шаблон в середине, непосредственно после ключевого символа.

select title\_id, notes

from titles

where notes like '%@%%' escape '@'

Результат:

|  |  |
| --- | --- |
| **title\_id** | **notes** |
| TC7777 | Detailed instructions on improving your position in life by learning how to make authentic Japanese sushi in your spare time. 5-10% increase in number of friends per recipe reported from beta test. |

Вот несколько примеров использования ключевого слова **LIKE**.

***Символ Значение***

LIKE '27%' 27, за которым следует любая строка символов

LIKE '27@%' 27%

LIKE '\_n' an, in, on и так далее

LIKE '@\_n' \_n

# Сортировка данных и другие методы выбора

## Сортировка результатов запроса: Предложение ORDER BY

Предложение **ORDER BY** позволяет улучшить представление получаемых результатов. С его помощью можно сортировать результаты по любому столбцу или выражению, указанному в списке выбора. Данные могут сортироваться как по убыванию, так и по возрастанию.

Обычно набор символов и порядок сортировки выбираются при установке БД. SQL предоставляет ряд таких команд для выбора набора символов и порядка сортировки: **CREATE CHARACTER SET, DROP CHARACTER SET, CREATE COLLATION, DROP COLLATION, COLLATE** и **COLLATION FROM**. Однако они редко реализуются в коммерческих системах.

### Как выполняется сортировка

Предположим, что нужно получить список цен, идентификационных номеров и номеров издателей из таблицы *titles.* Для этого можно выполнить следующий запрос:

select price, title\_id, pub\_id

from titles

|  |  |  |
| --- | --- | --- |
| **price** | **title\_id** | **pub\_id** |
| 20 | PC8888 | 1389 |
| 19.99 | BU1032 | 1389 |
| 7.99 | PS7777 | 0736 |
| 19.99 | PS3333 | 0736 |
| 11.95 | BU1111 | 1389 |
| 19.99 | MC2222 | 0877 |
| 14.99 | TC7777 | 0877 |
| 11.95 | TC4203 | 0877 |
| 22.95 | PC1035 | 1389 |
| NULL | MC3026 | 0877 |

В этом списке представлена вся запрошенная информация, но разбираться в ней неудобно, так как результат неупорядочен. Чтобы сделать эту информацию более полезной, ее надо отсортировать по цене. Это делается следующим образом:

select price, title\_id, pub\_id

from titles

order by price

|  |  |  |
| --- | --- | --- |
| **price** | **title\_id** | **pub\_id** |
| NULL | MC3026 | 0877 |
| 7.99 | PS7777 | 0736 |
| 11.95 | BU1111 | 1389 |
| 11.95 | TC4203 | 0877 |
| 14.99 | TC7777 | 0877 |
| 19.99 | MC2222 | 0877 |
| 19.99 | BU1032 | 1389 |
| 19.99 | PS3333 | 0736 |
| 20 | PC8888 | 1389 |
| 22.95 | PC1035 | 1389 |

Теперь строки будут отсортированы по цене.

### Синтаксис предложения ORDER BY

Общая форма предложения **ORDER BY** в операторе **SELECT** имеет следующий вид:

SELECT список\_выбора

FROM список\_таблиц

[WHERE условия]

[ORDER BY {выражение [ASC | DESC] | позиция [ASC | DESC] }

[ , {выражение [ASC | DESC] | позиция [ASC | DESC] } ]… ]

В большинстве систем требуется, чтобы каждый элемент, по которому выполняется сортировка (столбец или выражение), присутствовал в списке выбора. Для выражений имеется три возможности: можно использовать целое число, описывающее позицию выражения в списке выбора, использовать заголовок столбца, определенный в списке выбора (price \* ytd\_sales as income), или использовать выражение целиком.

### Сортировка внутри сортировки

После получения результата сортировки по ценам неплохо было бы, чтобы книги в каждой ценовой категории, выпущенные одним издателем, шли в списке рядом. Для этого в список **ORDER BY** нужно добавить столбец *pub\_id*:

select price, title\_id, pub\_id

from titles

order by price, pub\_id

|  |  |  |
| --- | --- | --- |
| **price** | **title\_id** | **pub\_id** |
| NULL | MC3026 | 0877 |
| 7.99 | PS7777 | 0736 |
| 11.95 | TC4203 | 0877 |
| 11.95 | BU1111 | 1389 |
| 14.99 | TC7777 | 0877 |
| 19.99 | PS3333 | 0736 |
| 19.99 | MC2222 | 0877 |
| 19.99 | BU1032 | 1389 |
| 20 | PC8888 | 1389 |
| 22.95 | PC1035 | 1389 |

При использовании более одного столбца в предложении **ORDER BY** выполняется так называемая **вложенная** (nested) сортировка, то есть сначала выполняется сортировка по цене, а затем по значениям *pub\_id* в каждой ценовой категории.

Количество уровней сортировки может быть любым. Во многих системах требуется, чтобы элемент, по которому ведется сортировка, присутствовал в списке выбора, однако порядок их перечисления в предложении **ORDER BY** не обязан совпадать с порядком перечисления столбцов и выражений в операторе **SELECT**. Если в предыдущем примере поменять порядок сортировки так, чтобы результаты сначала упорядочивались по столбцу *pub\_id*, а затем по столбцу *price*, результат изменится следующим образом:

select price, title\_id, pub\_id

from titles

order by pub\_id, price

Результат:

|  |  |  |
| --- | --- | --- |
| **price** | **title\_id** | **pub\_id** |
| 7.99 | PS7777 | 0736 |
| 19.99 | PS3333 | 0736 |
| NULL | MC3026 | 0877 |
| 11.95 | TC4203 | 0877 |
| 14.99 | TC7777 | 0877 |
| 19.99 | MC2222 | 0877 |
| 11.95 | BU1111 | 1389 |
| 19.99 | BU1032 | 1389 |
| 20 | PC8888 | 1389 |
| 22.95 | PC1035 | 1389 |

Порядок столбцов (*price, title\_id, pub\_id*) по сравнению с предыдущим примером не изменился, однако строки расположились по-другому: сначала идут строки с идентификационным номером издателя – 0736, затем – с номером 0877 и наконец – с номером 1389.

### Сортировка по возрастанию и убыванию

С помощью ключевых слов **ASC** (по возрастанию) и **DESC** (по убыванию) можно изменить порядок сортировки в каждом отдельном случае. По умолчанию данные сортируются в порядке возрастания. Для изменения порядка сортировки нужно использовать ключевое слово **DESC**.

С помощью следующего запроса можно вывести цены в убывающем порядке:

select price, title\_id, pub\_id

from titles

order by price DESC, pub\_id

|  |  |  |
| --- | --- | --- |
| **price** | **title\_id** | **pub\_id** |
| 22.95 | PC1035 | 1389 |
| 20 | PC8888 | 1389 |
| 19.99 | PS3333 | 0736 |
| 19.99 | MC2222 | 0877 |
| 19.99 | BU1032 | 1389 |
| 14.99 | TC7777 | 0877 |
| 11.95 | TC4203 | 0877 |
| 11.95 | BU1111 | 1389 |
| 7.99 | PS7777 | 0736 |
| NULL | MC3026 | 0877 |

Нужно обратить внимание, что в рамках каждой ценовой категории идентификаторы издателей по-прежнему сортируются в порядке возрастания. При этом нулевые (NULL) значения могут располагаться либо в начале, либо в конце списка. Чтобы изменить порядок сортировки в столбце *pub\_id*, нужно выполнить следующий запрос:

select price, title\_id, pub\_id

from titles

order by price DESC, pub\_id DESC

|  |  |  |
| --- | --- | --- |
| **price** | **title\_id** | **pub\_id** |
| 22.95 | PC1035 | 1389 |
| 20 | PC8888 | 1389 |
| 19.99 | PS3333 | 0736 |
| 19.99 | BU1032 | 1389 |
| 19.99 | MC2222 | 0877 |
| 14.99 | TC7777 | 0877 |
| 11.95 | BU1111 | 1389 |
| 11.95 | TC4203 | 0877 |
| 7.99 | PS7777 | 0736 |
| NULL | MC3026 | 0877 |

### А как насчет выражений?

Что делать, если нужно отсортировать результат по значению выражения из списка выбора? SQL позволяет использовать для этих целей позицию выражения в списке выбора (представленную целым числом) или заголовки (также называемые псевдонимами или метками). Некоторые системы позволяют использовать само выражение. Ниже приведен пример, в котором могут возникнуть проблемы при сортировке по выражению price \* ytd\_sales из списка выбора:

select pub\_id, price \* ytd\_sales, price, title\_id

from titles

|  |  |  |  |
| --- | --- | --- | --- |
| **pub\_id** |  | **price** | **title\_id** |
| 1389 | 81900 | 20 | PC8888 |
| 1389 | 81859.05 | 19.99 | BU1032 |
| 0736 | 26654.64 | 7.99 | PS7777 |
| 0736 | 81399.28 | 19.99 | PS3333 |
| 1389 | 46318.2 | 11.95 | BU1111 |
| 0877 | 40619.68 | 19.99 | MC2222 |
| 0877 | 61384.05 | 14.99 | TC7777 |
| 0877 | 180397.2 | 11.95 | TC4203 |
| 1389 | 201501 | 22.95 | PC1035 |

#### Сортировка по позиции

Предположим, что сначала результаты надо отсортировать по издателям, а затем по значению price \* ytd\_sales. Поскольку это значение является выражением, для него нельзя использовать обычное имя столбца. Вместо него нужно использовать число 2, так как выражение является вторым элементом списка выбора. (Нумерация начинается с 1 и ведется слева направо. Числа со знаками (например, -2, +4 и т.д.) не допускаются и в любом случае не имеют никакого смысла).

select pub\_id, price \* ytd\_sales, price, title\_id

from titles

order by pub\_id, 2

Результат:

|  |  |  |  |
| --- | --- | --- | --- |
| **pub\_id** |  | **price** | **title\_id** |
| 0736 | 777 | 7 | PS2106 |
| 0736 | 8096.25 | 21.59 | PS1372 |
| 0736 | 22392.75 | 10.95 | PS2091 |
| 0736 | 26654.64 | 7.99 | PS7777 |
| 0736 | 55978.78 | 2.99 | BU2075 |
| 0736 | 81399.28 | 19.99 | PS3333 |
| 0877 | NULL | NULL | MC3026 |
| 0877 | 7856.25 | 20.95 | TC3218 |
| 0877 | 40619.68 | 19.99 | MC2222 |

Цифры можно использовать как для выражений, так и для обычных столбцов. Ключевые слова **ASC** и **DESC** аналогично применяются и к цифрам, и к заголовкам столбцов. В следующем примере данные сортируются в порядке возрастания по столбцу *pub\_id*, а затем в порядке убывания по столбцу *price*.

select pub\_id, price \* ytd\_sales, price, title\_id

from titles

order by pub\_id, 3 DESC

Результат:

|  |  |  |  |
| --- | --- | --- | --- |
| **pub\_id** |  | **price** | **title\_id** |
| 0736 | 8096.25 | 21.59 | PS1372 |
| 0736 | 81399.28 | 19.99 | PS3333 |
| 0736 | 22392.75 | 10.95 | PS2091 |
| 0736 | 26654.64 | 7.99 | PS7777 |
| 0736 | 777 | 7 | PS2106 |
| 0736 | 55978.78 | 2.99 | BU2075 |
| 0877 | 7856.25 | 20.95 | TC3218 |
| 0877 | 40619.68 | 19.99 | MC2222 |
| 0877 | 61384.05 | 14.99 | TC7777 |

При использовании номеров в предложении **ORDER BY** нужно внимательно следить за изменениями в списке выбора. При добавлении или удалении столбцов из списка выбора результаты таких запросов могут измениться до неузнаваемости.

#### Сортировка по заголовку выражения

Если в списке выбора выражение определено с заголовком, по нему можно выполнять сортировку. Фактически, благодаря этой возможности можно вообще отказаться от сортировки по позиции. В следующем примере, который, как и предыдущий, приводит к аналогичным результатам, выражению price \* ytd\_sales назначен заголовок *income*. Именно этот заголовок и используется в предложении **ORDER BY** вместо номера позиции в списке выбора.

select pub\_id, price \* ytd\_sales as income, price, title\_id

from titles

order by pub\_id, income DESC

#### Сортировка по выражению

В некоторых системах допускается выполнение сортировки по самому выражению из списка выбора, без использования его номера позиции или заголовка.

select pub\_id, price \* ytd\_sales, price, title\_id

from titles

order by pub\_id, price \* ytd\_sales desc

### Как сортировать нулевые значения

Не все системы одинаково упорядочивают нулевые значения. Все зависит от конкретной реализации.

## Устранение повторяющихся строк: Предложения DISTINCT и ALL

С помощью ключевых слов **DISTINCT** и **ALL** в списке выбора можно определить, что делать с повторяющимися строками результата. **ALL** возвращает все строки, удовлетворяющие условиям запроса. **DISTINCT** возвращает только неповторяющиеся строки.

Например, при поиске в таблице *titleauthors* всех идентификаторов авторов с использованием ключевого слова **ALL** получается следующий результат:

select all au\_id

from titleauthors

Результат:

|  |
| --- |
| **au\_id** |
| 409-56-7008 |
| 486-29-1786 |
| 486-29-1786 |
| 712-45-1867 |
| 172-32-1176 |
| 213-46-8915 |

Внимательно посмотрев на результаты запроса, можно обнаружить в нем повторяющиеся строки. С помощью ключевого слова **DISTINCT** их можно устранить.

select distinct au\_id

from titleauthors

|  |
| --- |
| **au\_id** |
| 172-32-1176 |
| 213-46-8915 |
| 238-95-7766 |
| 267-41-2394 |
| 274-80-9391 |
| 409-56-7008 |

Таким образом из результатов предыдущего запроса будут удалены повторяющиеся строки.

### Синтаксис предложения DISTINCT

Основная форма предложения **DISTINCT** имеет следующий вид:

SELECT [DISTINCT | ALL] список\_выбора

Ключевые слова **DISTINCT** и **ALL** можно использовать в запросе только один раз и они должны стоять в начале списка выбора. В следующем примере умышленно совершена синтаксическая ошибка:

SQL (not right!):

select state, distinct city

from authors

Другими словами, нельзя выбрать все штаты и при этом только неповторяющиеся города.

### Почувствуйте разницу

Если в списке выбора находится несколько элементов, при использовании ключевого слова **DISTINCT** выбираются только строки с уникальными комбинациями значений этих элементов.

Рассмотрим следующий пример. Для начала получим список всех значений столбцов *pub\_id* и *types* из таблицы *titles*.

select pub\_id, type

from title

order by pub\_id

|  |  |
| --- | --- |
| **pub\_id** | **type** |
| 0736 | psychology |
| 0736 | psychology |
| 0736 | psychology |
| 0877 | NULL |
| 0877 | mod\_cook |
| 1389 | popular\_comp |
| 1389 | business |

Результат состоит из семи строк, включая повторяющиеся. (Предложение **ORDER BY** используется исключительно для упрощения вида результата.) Если попробовать извлечь различающиеся номера издателей, то получатся только три строки:

select distinct pub\_id

from titles

order by pub\_id

Результат:

|  |
| --- |
| **pub\_id** |
| 0736 |
| 0877 |
| 1389 |

При выборе только различающихся типов получатся только шесть следующих строк:

select distinct type

from titles

order by type

Результат:

|  |
| --- |
| **type** |
| NULL |
| business |
| mod\_cook |
| popular\_comp |
| psychology |
| trad\_cook |

В случае необходимости выбора различающихся комбинаций значений "издатель-тип" получаются семь строк:

select distinct pub\_id, type

from titles

order by pub\_id

|  |  |
| --- | --- |
| **pub\_id** | **type** |
| 0736 | business |
| 0736 | psychology |
| 0877 | NULL |
| 0877 | mod\_cook |
| 0877 | trad\_cook |
| 1389 | business |
| 1389 | popular\_comp |

Теперь в этом списке представлены только уникальные комбинации значений "издатель-тип". Таким образом, ключевое слово **DISTINCT** применяется ко всему списку выбора, а не к отдельным столбцам.

***А различаются ли нулевые значения?***

При использовании ключевого слова **DISTINCT** все нулевые значения NULL в столбце считаются повторяющимися.

***DISTINCT \*.*** Если система допускает конструкцию **DISTINCT \***, можно сравнить результаты выполнения двух следующих запросов:

select distinct \*

from titles

select \*

from titles

Скорее всего, получатся одинаковые результаты, так как все строки в таблице должны быть уникальны. В противном случае нужно обратить серьезное внимание на структуру БД. Почему в одной таблице оказались строки с одинаковыми наборами значений? Как извлечь конкретную строку, если в таблице имеется несколько его "двойников"?

***DISTINCT и ORDER BY.*** В большинстве диалектов SQL каждый элемент предложения **ORDER BY** должен также находиться и в списке выбора. В системах, предоставляющих большую гибкость (когда элемент предложения **ORDER BY** может не включаться в список выбора), можно выполнять запросы с ключевым словом **DISTINCT**, примененным к списку выбора, и с предложением **ORDER BY**, содержащем элементы, не входящие в список выбора.

Например, в Sybase SQL Server при сортировке по столбцу, не входящему в список выбора, получается тот же результат, что и при включении этого столбца в список выбора с ключевым словом **DISTINCT**. В обоих случаях это приводит к увеличению количества выбранных строк.

select distinct pub\_id

from titles

order by type

Так как в базе представлены только три издателя, в результате можно было бы ожидать получение трех строк. Однако вот что получается на самом деле:

|  |
| --- |
| **pub\_id** |
| 0877 |
| 0736 |
| 1389 |
| 0877 |
| 1389 |
| 0736 |
| 0877 |

Почему в результате оказалось семь строк? Ответ заключен в предложении **ORDER BY**. Это же количество строк было бы получено и при сортировке по издателям и поиске различных типов.

select distinct type

from titles

order by pub\_id

Результат:

|  |
| --- |
| **type** |
| business |
| psychology |
| NULL |
| mod\_cook |
| trad\_cook |
| business |
| popular\_comp |

Теперь становится понятно, откуда взялись семь строк. Система находит все возможные уникальные комбинации значений элементов из списка выбора (с ключевым словом **DISTINCT**) и предложения **ORDER BY** (не входящие в список выбора). Таких комбинаций *pub\_id* и *type* ровно семь.

## Агрегирующие функции

Агрегирующие функции используются для получения обобщающих значений. Их можно применять к **наборам** (set) строк: ко всем строкам таблицы, строкам, определенным в предложении **WHERE**, или к группам строк в предложении **GROUP BY** (описывается далее). В любом случае, независимо от структуры набора строк, *для каждого из них получается единственное значение.*

Обратите внимание на различие в результатах выполнения следующих запросов: первый находит для каждой строки таблицы *titles* соответствующее значение годового объема продаж, второй вычисляет общую сумму, полученную за год от продажи всех книг (одно значение для набора, состоящего из всех строк таблицы).

select ytd\_sales

from titles

Результат:

|  |
| --- |
| **ytd\_sales** |
| 4095 |
| 4095 |
| 3336 |
| 4072 |
| 3876 |
| 2032 |

select sum(ytd\_sales)

from titles

Результат:

--------

21506

В первом случае результат возвращается для каждой отдельной строки таблицы, во втором получается одно значение для всех строк. Получаемые таким образом столбцы результата выводятся без заголовков. Чтобы сделать результат более понятным, можно использовать следующий запрос:

select sum(ytd\_sales) as Total

from titles

|  |
| --- |
| **Total** |
| 21056 |

В большинстве диалектов SQL запрещается смешивать в одном предложении обычные значения и значения агрегирующих функций. Таким образом, список выбора может быть либо пустым, либо состоять из имен столбцов и выражений (обрабатывающих значения строк), либо включать только агрегирующие функции (применяющиеся к наборам значений). Единственным исключением является предложение **GROUP BY**, которое рассматривается далее. Ниже приведен пример обычно недопустимого запроса:

select price, sum(price)

from titles

В этом случае проблема заключается в том, что *price* возвращает значение для каждой строки, тогда как *sum(price)* возвращает значение для целого набора строк (в данном случае для всех строк таблицы).

В случае обработки системой подобного запроса, вероятнее всего, на экране отобразится следующий результат:

|  |  |
| --- | --- |
| **price** |  |
| 19.99 | 236.26 |
| 11.95 | 236.26 |
| 2.99 | 236.26 |
| 19.99 | 236.26 |
| 19.99 | 236.26 |
| 2.99 | 236.26 |
| NULL | 236.26 |

В столбце без названия содержится значение общей стоимости всех книг, тогда как в левом столбце представлена цена каждой книги. Нужно обратить внимание, что все значения в правом столбце одинаковы, так как в каждой строке вычислялась одна и та же сумма.

### Синтаксис агрегирующих функций

Агрегирующие функции всегда имеют **аргументы** (argument). Аргументы являются выражениями и заключаются в скобки. Общая форма агрегирующей функции имеет следующий вид:

aggregate\_function ([DISTINCT] выражение)

Ниже приведен список агрегирующих функций

|  |  |
| --- | --- |
| **Агрегирующая функция** | **Результат** |
| SUM ([DISTINCT] выражение) | Сумма (различных) значений |
| AVG ([DISTINCT] выражение) | Средняя величина (различных) значений |
| COUNT ([DISTINCT] выражение) | Количество (различных) ненулевых значений |
| COUNT (\*) | Количество выбранных строк |
| MAX (выражение) | Максимальное значение |
| MIN (выражение) | Минимальное значение |

Со всеми агрегирующими функциями, за исключением COUNT(\*), можно использовать ключевое слово **DISTINCT**. Тем не менее, не имеет никакого смысла использовать его с функциями MAX и MIN, так как получаемые значения в обоих случаях одинаковы.

В качестве аргументов агрегирующих функций обычно используются названия столбцов, но также допускаются константы, функции и любые их комбинации с арифметическими операторами.

Например, с помощью следующего оператора можно найти среднюю стоимость всех книг в случае удвоения их цены:

select avg(price \*2)

from titles

Результат:

------

29.5325

***Функции COUNT и COUNT(\*).*** Несмотря на внешнюю схожесть, функции COUNT и COUNT(\*) используются для разных целей. COUNT в качестве аргумента использует столбец или выражение и подсчитывает общее количество его ненулевых значений, тогда как COUNT(\*) находит общее количество строк, независимо от наличия в них нулевых значений. Сравните эти функции на следующем примере:

select count(price), count(\*)

from titles

Результат:

-----

16 18

Получаемые с помощью этих функций результаты отличаются, так как две строки в таблице *titles* содержат нулевые значения в столбце *price*. При использовании столбца без нулевых значений результаты работы функций COUNT и COUNT(\*) будут одинаковы.

select count(title\_id), count(\*)

from titles

Результат:

-----

18 18

Совместное использование этих функций может быть полезно при подсчете нулевых значений в заданных столбцах.

***Агрегирующие функции и типы данных.*** Функции SUM и AVG можно применять только к числовым столбцам. Функции MIN, MAX, COUNT и COUNT(\*) работают со всеми типами данных.

Например, функцию MIN можно использовать для поиска наименьшего значения (в символьном столбце – самого первого значения в алфавитном порядке):

select min(au\_lname)

from authors

Результат:

------

Bennet

Конечно, суммировать или находить среднее значение по столбцу с фамилиями авторов абсолютно бессмысленно.

***DISTINCT и агрегирующие функции.*** С функциями SUM, AVG, COUNT, MIN и MAX можно использовать ключевое слово **DISTINCT** (помещается перед аргументом). Как уже отмечалось раньше, **DISTINCT** не влияет на результаты выполнения функций MAX и MIN.

При использовании ключевого слова **DISTINCT** при вычислении суммы, среднего значения или общего количества значений не учитываются повторяющиеся строки. Ниже приведен соответствующий пример:

select count(price)

from titles

Результат:

------

16

select count(distinct price)

from titles

Результат:

-----

11

В первом запросе находится количество ненулевых значений цены в таблице *titles*. Во втором подсчитывается количество различных ненулевых значений цены. При сравнении полученных результатов видно, что пять книг имеют одинаковую стоимость.

В некоторых системах при использовании ключевого слова **DISTINCT** аргумент не может быть арифметическим выражением, а должен являться исключительно именем столбца. Соответствующую информацию можно всегда найти в руководстве по конкретной системе. Также можно проверить работоспособность следующих запросов:

select count(price \* 2)

from titles

Результат:

-----

16

SQL (вариант):

select count(distinct price \* 2)

from titles

Результат:

-----

11

В первом запросе удваиваются все цены и подсчитывается их общая сумма. Во втором выполняется та же операция, но учитываются только неповторяющиеся значения. Если система не позволяет использовать выражение вместе с ключевым словом **DISTINCT**, то при выполнении второго запроса появится сообщение об ошибке.

**DISTINCT** также не применяется вместе с функцией COUNT(\*), потому что она всегда возвращает единственную строку. В этом случае использование **DISTINCT** абсолютно бессмысленно.

Как правило, ключевое слово **DISTINCT** в списке выбора можно использовать только один раз. При отсутствии агрегирующих функций **DISTINCT** применяется ко всему списку выбора, а не к отдельным столбцам. Однако это ограничение может привести к возникновению некоторых проблем при использовании агрегирующих функций. Для примера достаточно рассмотреть результат выполнения двух операций. При подсчете суммы и количества значений в столбце *price* получаются следующие результаты:

select count(price), sum(price)

from titles

Результат:

-------

16 236.26

Таким образом, средняя цена одной книги составляет 236,26 долларов делить на 16.

При использовании ключевого слова **DISTINCT** с одним из столбцов результат, скорее всего, будет менее полезен:

select count(price), sum(distinct price)

from titles

Результат:

-----

16 161.35

Различие результатов связано с существованием в столбце одинаковых значений цены, которые не учитываются при подсчете суммы. Тем не менее общее количество значений осталось неизменным. Поэтому на основе этих данных уже нельзя правильно вычислить среднюю стоимость книги. Точно так же неправильно средняя стоимость будет получена и при использовании ключевого слова **DISTINCT** в функции **COUNT**.

select count(distinct price), sum(price)

from titles

Результат:

-----

11 236.26

Очевидно, что в этом случае нужно либо вообще не использовать ключевое слово **DISTINCT**, либо использовать его в обеих функциях. Ниже приведен соответствующий пример:

select count(distinct price), sum(distinct price)

from titles

Результат:

-----

11 161.35

Обратите внимание на то, что использование ключевого слова **DISTINCT** в списке выбора и в агрегирующей функции приводит к разным результатам:

select count(au\_id)

from titleauthors

Результат:

-----

25

select count(distinct au\_id)

from titleauthors

Результат:

-----

19

select distinct count(au\_id)

from titleauthors

Результат:

----

25

С помощью первого запроса находится общее количество идентификационных номеров. С помощью второго подсчитываются только различные номера. Применение ключевого слова **DISTINCT** ко всему списку выбора дает тот же результат, что и в первом запросе. Это связано с тем, что сначала выполняется функция, которая возвращает в результате одну строку, а затем к этой строке применяется ключевое слово **DISTINCT**. А так как эта строка единственная, по определению не может существовать никаких повторяющихся значений.

***Агрегирующие функции и предложение WHERE.*** Агрегирующие функции можно использовать в списке выбора, как в предыдущих примерах, либо в предложении **HAVING** оператора **SELECT**. При этом их нельзя использовать в предложении **WHERE**. Однако предложение **WHERE** можно использовать для ограничения числа строк, участвующих в агрегирующих вычислениях. С помощью следующего оператора вычисляется средний расход по книгам и годовой доход от продаж по всем строкам таблицы *titles*.

select avg(advance), sum(ytd\_sales)

from titles

where type = 'business'

Результат:

-----

6281.25 30788

Как взаимодействуют предложение **WHERE** и агрегирующие функции? Сначала выполняется предложение **WHERE**, в результате чего находятся все книги по бизнесу. Затем к выбранным строкам применяются агрегирующие функции.

***Нулевые значения и агрегирующие функции*.** Если в столбце, к которому применяется агрегирующая функция, имеются нулевые (NULL) значения, они просто игнорируются. Исключением является функция COUNT(\*), которая всегда подсчитывает общее количество строк, независимо от наличия в них нулевых значений. Если условиям запроса не удовлетворяет ни одна строка, функция COUNT возвращает нуль. Все другие функции в этом случае возвращают значение NULL. Ниже приведен соответствующий пример:

select count(distinct title)

from titles

where type = 'poetry'

Результат:

-----

0

select avg(advance)

from titles

where type = 'poetry'

Результат:

-----

NULL

# Группировка данных и построение отчетов

## Предложение GROUP BY

Предложение **GROUP BY** неразрывно связано с агрегирующими функциями, без них оно практически не используется. Данное предложение разделяет таблицу на наборы, а агрегирующая функция вычисляет для каждого из них итоговое значение. Эти значения называются агрегирующим вектором. Сравним скалярные и векторные запросы:

select avg(price)

from titles

Результат:

-----

14,77

Выше представлен скалярный агрегирующий запрос, возвращающий единственное значение.

select type, avg(price)

from titles

group by type

Результат:

|  |  |
| --- | --- |
| **type** |  |
| NULL | NULL |
| Business | 13.73 |
| mod\_cook | 11.49 |
| popular\_comp | 21.48 |
| psychology | 13.5 |
| trad\_cook | 15.96 |

Выше представлен векторный агрегирующий запрос, возвращающий несколько значений.

### Синтаксис предложения GROUP BY

В контексте оператор **SELECT** предложение **GROUP BY** имеет следующий вид:

SELECT список\_выбора

FROM список\_таблиц

[WHERE условия]

[GROUP BY список\_группировки]

[ORDER BY список\_порядка]

В большинстве диалектов SQL каждый элемент из списка **GROUP BY** должен обязательно присутствовать в списке выбора – другими словами, группировать можно только выбираемые элементы. Разные системы, кроме имен столбцов, позволяют использовать в списке группировки выражения, заголовки столбцов и их порядковые номера в списке выбора. Ниже приводится пример, в котором группировка выполняется по одному столбцу:

select pub\_id, count(type)

from titles

group by pub\_id

Результат:

|  |  |
| --- | --- |
| **pub\_id** |  |
| 0736 | 6 |
| 0877 | 5 |
| 1389 | 6 |

В список выбора включается столбец, по которому выполняется группировка и агрегирующая функция. Все строки из первой группы имеют значение 0736 в столбце *pub\_id,* из второй группы – 0877, из третьей – 1389. Функция COUNT вычисляет для каждой группы соответствующее ей единственное значение.

***Группировка внутри групп***. Путем сортировки одновременно по нескольким элементам можно создавать группы внутри групп. Разделяя элементы, по которым будет проводиться группировка, с помощью запятых можно разбить большие группы на подгруппы.

select pub\_id, type, count(type)

from titles

group by pub\_id, type

Результат:

|  |  |  |
| --- | --- | --- |
| **pub\_id** | **type** |  |
| 0736 | business | 1 |
| 0736 | psychology | 5 |
| 0877 | NULL | 0 |
| 0877 | mod\_cook | 2 |
| 0877 | trad\_cook | 3 |
| 1389 | business | 3 |
| 1389 | popular\_comp | 3 |

Этот пример во многом аналогичен предыдущему, но в нем используются вложенные группы. Сначала строки таблицы разделяются по издателям, а затем каждая полученная группа разделяется по типу. В результате получается семь групп, или наборов. После этого к каждому такому набору применяется агрегирующая функция, которая вычисляет для каждого издателя количество книг по каждой теме.

***Ограничения.*** Несмотря на простой синтаксис, предложение **GROUP BY** часто является источником головной боли пользователей SQL.

Ниже приводится кажущийся резонным запрос, который не будет работать в большинстве систем:

SQL (вариант):

select pub\_id, type, count(type)

from titles

group by pub\_id

Так как таблица разделяется на наборы по издателям (group by pub\_id), результат запроса должен состоять не более чем из трех строк, и каждый элемент из списка выбора должен иметь по одному значению для набора. Однако у издателей имеются книги нескольких типов, и поэтому в большинстве систем этот запрос не пройдет. Решить эту проблему можно, добавив в предложение **GROUP BY** столбец *type*, как в предыдущем примере.

Другое ограничение в некоторых реализациях SQL касается использования выражений. В предложении **GROUP BY** всегда можно использовать имена столбцов, а поддержка выражений, заголовков столбцов и их порядковых номеров в списке выбора зависит от конкретной системы.

А как быть с множественными итоговыми значениями при наличии нескольких уровней группировки? Предположим, что выполняется сортировка по столбцам *pub\_id* и *type*. Требуется найти общее количество книг для каждого издателя и количество книг по каждой теме. На первый взгляд с этой задачей должен справиться следующий запрос:

select pub\_id, count(title\_id), type, count(title\_id)

from titles

group by pub\_id, type

Однако результат свидетельствует об обратном:

Результат:

|  |  |  |  |
| --- | --- | --- | --- |
| **pub\_id** |  | **type** |  |
| 0736 | 1 | business | 1 |
| 0736 | 5 | psychology | 5 |
| 0877 | 1 | NULL | 1 |
| 0877 | 2 | mod\_cook | 2 |
| 0877 | 3 | trad\_cook | 3 |
| 1389 | 3 | business | 3 |
| 1389 | 3 | popular\_comp | 3 |

Количество книг по издателям и по типам совпадают, что явно не соответствует тому, что требовалось получить. В числовых столбцах содержится количество книг для каждой комбинации издатель/тип, так как это нижний уровень группы. Для получения требуемых результатов нужно выполнить два отдельных запроса: в первом задать группировку по издателям, а во втором – по издателям, а затем по типам.

select pub\_id, count(title\_id)

from titles

group by pub\_id

Из результатов первого запроса становится ясно, что каждый издатель выпустил по шесть книг.

Результат:

|  |  |
| --- | --- |
| **pub\_id** |  |
| 0736 | 6 |
| 0877 | 6 |
| 1389 | 6 |

Во втором запросе находится общее количество книг для каждой комбинации издатель/тип.

select pub\_id, type, count(title\_id)

from titles

group by pub\_id, type

Результат:

|  |  |  |
| --- | --- | --- |
| **pub\_id** | **type** |  |
| 0736 | business | 1 |
| 0736 | psychology | 5 |
| 0877 | NULL | 1 |
| 0877 | mod\_cook | 2 |
| 0877 | trad\_cook | 3 |
| 1389 | business | 3 |
| 1389 | popular\_comp | 3 |

В случае группировки только по типу результаты будут другими:

select type, count(title\_id)

from titles

group by type

|  |  |
| --- | --- |
| **type** |  |
| NULL | 1 |
| business | 4 |
| mod\_cook | 2 |
| popular\_comp | 3 |
| psychology | 5 |
| trad\_cook | 3 |

В последнем случае в бизнес-категорию попадают четыре книги. Однако эти книги были выпущены двумя издателями, три книги – одним издателем и одна – другим. Таким образом, из результатов этого запроса неясно, сколько каких книг выпустил каждый издатель.

Поскольку зачастую возникает необходимость в просмотре итоговых значений на разных уровнях группировки, многие поставщики СУБД предоставляют своим пользователям специальные генераторы отчетов. Однако эти генераторы обычно являются отдельным приложением, а не частью SQL, так как подобные отчеты нельзя получить в рамках чисто реляционной модели.

Microsoft SQL предоставляет для решения этой задачи специальное расширение. Ниже приводится соответствующий пример:

select pub\_id, type, title\_id

from titles

order by pub\_id, type

compute count(title\_id) by pub\_id, type

compute count(title\_id) by pub\_id

Результат:

|  |  |  |
| --- | --- | --- |
| **pub\_id** | **type** | **title\_id** |
| 0736 | business | BU2075 |

Count

------

1

|  |  |  |
| --- | --- | --- |
| **pub\_id** | **type** | **title\_id** |
| 0736 | psychology | PS1372 |
| 0736 | psychology | PS2091 |
| 0736 | psychology | PS2106 |
| 0736 | psychology | PS3333 |
| 0736 | psychology | PS7777 |

Count

-----

5

Count

-----

6

|  |  |  |
| --- | --- | --- |
| **pub\_id** | **type** | **title\_id** |
| 0877 | NULL | MC3026 |

Count

-----

1

|  |  |  |
| --- | --- | --- |
| **pub\_id** | **type** | **title\_id** |
| 0877 | mod\_cook | MC2222 |
| 0877 | mod\_cook | MC3021 |

Count

-----

2

|  |  |  |
| --- | --- | --- |
| **pub\_id** | **type** | **title\_id** |
| 0877 | trad\_cook | TC3218 |
| 0877 | trad\_cook | TC4203 |
| 0877 | trad\_cook | TC7777 |

Count

-----

3

Count

-----

6

|  |  |  |
| --- | --- | --- |
| **pub\_id** | **type** | **title\_id** |
| 1389 | popular\_comp | PC1035 |
| 1389 | popular\_comp | PC8888 |
| 1389 | popular\_comp | PC9999 |

Count

-----

3

Count

-----

6

В этом отчете содержатся значения строк, итоговые данные по группам и подгруппам. В отличие от других запросов SQL, эти результаты не являются реляционными и не могут использоваться в других операторах SQL.

***Нулевые значения и группы.*** Так как нулевые значения (NULL) представляют неизвестные данные, то нельзя определенно сказать, что одно нулевое значение больше или меньше другого нулевого значения.

Однако, если столбец, по которому выполняется группировка, содержит больше одного нулевого значения, все они будут собраны в одну группу.

Столбец *type* в таблице *titles* содержит нулевое значение. Поэтому при группировке по этому столбцу и подсчете строк в каждой группе будет получен следующий результат:  
select type, count(\*)

from titles

group by type

Результат:

|  |  |
| --- | --- |
| **type** |  |
| NULL | 1 |
| business | 4 |
| mod\_cook | 2 |
| popular\_comp | 3 |
| psychology | 5 |
| trad\_cook | 3 |

В результате содержится строка с типом NULL. Если в запросе вместо функции *count(\*)* использовать функцию *count(type)*, то вместо 1 во втором столбце этой строки будет стоять 0:

select type, count(type)

from titles

group by type

Результат:

|  |  |
| --- | --- |
| **type** |  |
| NULL | 0 |
| business | 4 |
| mod\_cook | 2 |
| popular\_comp | 3 |
| psychology | 5 |
| trad\_cook | 3 |

Тогда как функция *count(\*)* подсчитывает общее количество строк в группе, независимо от значений в конкретных столбцах, функция *count()* работает с определенным столбцом и учитывает при подсчете только ненулевые значения. При выполнении предложения **GROUP BY** система обнаруживает существование типа NULL и создает для него отдельную группу. Затем функция *count()* подсчитывает количество элементов в группе. В результате находится только значение NULL, которое не учитывается при подсчете, и поэтому в правый столбец записывается нуль.

А что, если в столбце, по которому ведется группировка, имеется несколько нулевых значений? Чтобы ответить на этот вопрос, рассмотрим следующий запрос:

select advance, count(\*)

from titles

group by advance

Результат:

|  |  |
| --- | --- |
| **advance** |  |
| NULL | 2 |
| 0 | 1 |
| 2000 | 1 |
| 2275 | 1 |
| 4000 | 2 |
| 5000 | 3 |
| 6000 | 1 |
| 7000 | 3 |
| 8000 | 2 |
| 10125 | 1 |
| 15000 | 1 |

Затраты на две книги неизвестны. Обе они включаются в группу NULL. Нужно обратить внимание, что книги без затрат (*advance* = 0) составляют отдельную группу, так как NULL и 0 – это абсолютно разные значения.

***Предложение GROUP BY без агрегирующих функций.*** Без агрегирующих функций предложение **GROUP BY** напоминает предложение **DISTINCT**. Оно разделяет таблицу на группы и для каждой из них возвращает по одной строке. Нужно помнить, что при использовании предложения **GROUP BY** для каждого элемента из списка выбора будет генерироваться по одному значению на выбор. Несколько следующих примеров наглядно это демонстрируют. Сначала получим список всех издателей из таблицы *titles*.

select pub\_id

from titles

Результат:

|  |
| --- |
| **pub\_id** |
| 1389 |
| 1389 |
| 0736 |
| 0736 |
| 1389 |
| 0877 |
| 0877 |
| 0877 |
| 1389 |

Теперь сгруппируем данные по издателям:

select pub\_id

from titles

group by pub\_id

Результат:

|  |
| --- |
| **pub\_id** |
| 0736 |
| 0877 |
| 1389 |

Такой же результат получился бы и при использовании ключевого слова **DISTINCT**.

***Предложение GROUP BY с агрегирующими функциями.*** Фактически агрегирующие функции и предложение **GROUP BY** просто созданы друг для друга. **GROUP BY** создает набор, а агрегирующие функции вычисляют для него конкретное значение. С их помощью можно получить очень полезную информацию.

С помощью следующего оператора находятся средние затраты и сумма доходов от продаж по *каждому типу книг*:

select type, avg(advance), sum(ytd\_sales)

from titles

group by type

Результат:

|  |  |  |
| --- | --- | --- |
| **type** |  |  |
| NULL | NULL | NULL |
| business | 6281.25 | 30788 |
| mod\_cook | 7500 | 24278 |
| popular\_comp | 7500 | 12875 |
| psychology | 4255 | 9939 |
| trad\_cook | 6333.33333333333 | 19566 |

С помощью следующего запроса можно проследить зависимость между ценовыми категориями и средними расходами:

select price, avg(advance)

from titles

group by price

Результат:

|  |  |
| --- | --- |
| **price** |  |
| NULL | NULL |
| 2.99 | 12562.5 |
| 7 | 6000 |
| 7.99 | 4000 |
| 10.95 | 2275 |
| 11.95 | 4500 |

***GROUP BY с предложением WHERE.*** При отсутствии в запросе предложения **GROUP BY** агрегирующие функции применяются ко всей таблице целиком. В этом случае для выбора строк, участвующих в вычислениях, можно было использовать предложение **WHERE**. Это же верно и при наличии групп.

Совместная работа предложений **WHERE** и **GROUP BY** происходит следующим образом. Сначала находятся все строки, удовлетворяющие условиям предложения **WHERE**. Затем предложение **GROUP BY** делит отобранные строки на группы. Ниже приведен соответствующий пример:

select type, avg(price)

from titles

where advance > $5000

group by type

Результат:

|  |  |
| --- | --- |
| **type** |  |
| business | 2.99 |
| mod\_cook | 2.99 |
| popular\_comp | 21.48 |
| psychology | 14.3 |
| trad\_cook | 17.97 |

Этот же запрос без предложения **WHERE** приводит к другим результатам:

select type, avg(price)

from titles

group by type

Результат:

|  |  |
| --- | --- |
| **type** |  |
| NULL | NULL |
| business | 13.73 |
| mod\_cook | 11.49 |
| popular\_comp | 21.47 |
| psychology | 13.5 |
| trad\_cook | 15.96 |

В результате выполнения этого запроса появляется новая строка (с типом и средней ценой NULL), а также изменяются все средние цены, за исключением книг по компьютерам. Появление новой строки объяснить очень просто – в предложении **WHERE** ищутся все строки со значением расходов выше $5000, поэтому строки с нулевым значением расходов сразу отвергаются.

Нужно обратить внимание, что столбцы в предложении **WHERE** никоим образом не связаны со столбцами из списка выбора и списка группировки.

### Упорядоченные группы

Предложение **GROUP BY** разделяет строки на наборы, но при этом не упорядочивает их. Чтобы расположить результаты в определенном порядке, нужно использовать предложение **ORDER BY**. Например, чтобы найти среднюю стоимость книг по каждому типу, затраты на которые превысили $5000, и упорядочить результаты по цене, нужно выполнить следующий запрос:

select type, avg(price)

from titles

where advance > $5000

group by type

order by 2

Результат:

|  |  |
| --- | --- |
| **type** |  |
| mod\_cook | 2.99 |
| business | 2.99 |
| psychology | 14.295 |
| trad\_cook | 17.97 |
| popular\_comp | 21.475 |

## Предложение HAVING

В самом общем смысле предложение **HAVING** работает аналогично предложению **WHERE**, но применяется к группам. **WHERE** накладывает ограничения на строки, а **HAVING** – на группы. Как правило, предложение **HAVING** используется с предложением **GROUP BY**.

Если в списке выбора имеются агрегирующие функции, предложение **WHERE** выполняется перед ними, тогда как предложение **HAVING** применяется ко всему запросу в целом, после вычисления значений функций и разбиения на группы. Чтобы не запутаться в тонкостях, нужно просто запомнить порядок предложений в операторе **SELECT**. Предложение **WHERE** всегда стоит после предложения **FROM**, а предложение **HAVING** – после предложения **GROUP BY**.

С точки зрения синтаксиса условного выражения, предложения **HAVING** и **WHERE** идентичны, отличие состоит лишь в том, что в условии предложения **WHERE** не могут находиться агрегирующие функции. Кроме того, в большинстве систем элементы предложения **HAVING** должны включаться в список выбора. Также в данном предложении может содержаться любое количество условий.

### Разновидности предложения HAVING

Предложение **HAVING** работает следующим образом: сначала **GROUP BY** разделяет строки на наборы (по типу), затем на полученные группы накладываются условия предложения **HAVING**. В следующем примере устраняются наборы, содержащие только одну книгу:

select type, count(\*)

from titles

group by type

having count(\*) > 1

Результат:

|  |  |
| --- | --- |
| **type** |  |
| business | 4 |
| mod\_cook | 2 |
| popular\_comp | 3 |
| psychology | 5 |
| trad\_cook | 3 |

Нужно обратить внимание, что в этом случае нельзя просто заменить предложение **HAVING** на **WHERE**, так как последнее не допускает использования агрегирующих функций. Ниже приведен пример использования предложения **HAVING** без агрегирующих функций. В данном случае строки таблицы *titles* группируются по типу и удаляются типы, название которых не начинается с буквы "p":

select type

from titles

group by type

having type like 'p%'

Результат:

|  |
| --- |
| **type** |
| popular\_comp |
| psychology |

Если в предложении **HAVING** есть несколько условий, они объединяются с помощью операторов AND, OR и NOT. Например, оператор, группирующий строки таблицы *titles* по издателям и включающий в конечный результат только группы издателей с идентификационными номерами, большими 0800, суммарными расходами, превышающими $15000, и средней ценой книг меньше $20, имеет следующий вид:

select pub\_id, sum(advance), avg(price)

from titles

group by pub\_id

having sum(advance) > $15000

AND avg(price) < $20

AND pub\_id > '0800'

Результат:

|  |  |  |
| --- | --- | --- |
| **pub\_id** |  |  |
| 0877 | 34000 | 14.174 |
| 1389 | 30000 | 18.976 |

В следующем примере иллюстрируется совместное использование предложений **GROUP BY, HAVING, WHERE** и **ORDER BY** в операторе **SELECT**. В результате его выполнения получаются те же группы, что и в предыдущем примере, но все вычисления выполняются без учета книг стоимостью меньше $5. Результаты также упорядочиваются по столбцу *pub\_id*:

select pub\_id, sum(advance), avg(price)

from titles

where price >= $5

group by pub\_id

having sum(advance) > $15000

AND avg(price) < $20

AND pub\_id > '0800'

order by pub\_id

Результат:

|  |  |  |
| --- | --- | --- |
| **pub\_id** |  |  |
| 0877 | 19000 | 16.97 |
| 1389 | 30000 | 18.976 |

### Предложения HAVING и WHERE

Хотя предложения **HAVING** и **WHERE** во многом аналогичны, нужно всегда помнить о различиях в выполняемых ими действиях. Например, если условие *price >= $5* вместо **WHERE** поместить в предложение **HAVING**, то получится совсем другой результат. (В большинстве систем элементы предложения **HAVING** должны включаться в список выбора. Если это имеет место в вашей системе, при попытке выполнить следующий запрос система выдаст сообщение об ошибке, так как столбец *price* не содержится в списке выбора).

SQL (вариант):

select pub\_id, sum(advance), avg(price)

from titles

group by pub\_id

having sum(advance) > $15000

AND avg(price) < $20

AND pub\_id > '0800'

AND price >= $5

order by pub\_id

Причина этих различий заключается в том, что предложение **WHERE** отсеивает строки до группировки, а предложение **HAVING** – после.

## Еще о нулевых значениях

При выполнении запроса всегда нужно помнить, что в таблице могут существовать нулевые значения NULL.

### Значения по умолчанию в качестве альтернативы нулевым значениям

Одной из альтернатив нулевым значениям являются значения по умолчанию, которые автоматически вводятся СУБД в случае, если пользователь не ввел конкретное значение. Эти значения по умолчанию можно определять в команде **CREATE TABLE**.

Одно из преимуществ значений по умолчанию перед нулевыми значениями состоит в том, что они обеспечивают вместо неопределенности конкретные значения в столбцах. Подходящим значением по умолчанию для столбца *type* может быть значение "неизвестно", для столбца *date* – текущая дата.

С помощью функции Transact-SQL, которая называется **ISNULL()**, типу NULL можно задать другое название:

SQL(вариант):

select isnull(type, 'What?'), count(\*)

from titles

group by type

Результат:

|  |  |
| --- | --- |
| **type** |  |
| What? | 1 |
| business | 4 |
| mod\_cook | 2 |
| popular\_comp | 3 |
| psychology | 5 |
| trad\_cook | 3 |

Функция **ISNULL()** имеет два аргумента – имя столбца и значение, используемое вместо всех найденных в этом столбце нулей. Естественно, реальные значения в БД при этом не изменяются.

Ниже приведен запрос, с помощью которого выводятся значения *title\_id, advance* и *price* для книг, расходы на которые либо составили меньше $6000, либо неизвестны:

select title\_id, advance, price

from titles

where advance < $6000 or advance is null

order by price

Результат:

|  |  |  |
| --- | --- | --- |
| **title\_id** | **advance** | **price** |
| PC9999 | NULL | NULL |
| MC3026 | NULL | NULL |
| PS7777 | 4000 | 7.99 |
| PS2091 | 2275 | 10.95 |
| TC4203 | 4000 | 11.95 |
| BU1111 | 5000 | 11.95 |
| BU7832 | 5000 | 19.99 |

В двух строках значения расходов и цены неизвестны (NULL). Чтобы заменить нули на другие значения, необходимо выполнить следующий запрос:

SQL (вариант):

select title\_id, isnull(advance, 4000), isnull(price, 35.00)

from titles

where advance < $6000 or advance is null

order by price

Результат:

|  |  |  |
| --- | --- | --- |
| **title\_id** | **advance** | **price** |
| PC9999 | 4000 | 35.00 |
| MC3026 | 4000 | 35.00 |
| PS7777 | 4000 | 7.99 |
| PS2091 | 2275 | 10.95 |
| TC4203 | 4000 | 11.95 |
| BU1111 | 5000 | 11.95 |
| BU7832 | 5000 | 19.99 |

В результате изменились значения в двух строках.

# Объединение таблиц и сложный анализ данных

Объединение – последний из трех операторов (выбор, проектирование и объединение), которые должен поддерживать любой язык реляционных запросов. С помощью операции объединения в одном операторе **SELECT** можно манипулировать данными из разных таблиц. Объединение таблиц сродни сборке из отдельных частей конструктора единого сооружения.

SQL-92 предоставляет для операции объединения несколько ключевых слов (**CROSS JOIN, NATURAL, INNER, OUTER**), однако они редко используются в коммерческих реализациях. В большинстве систем объединение определяется в предложении **WHERE** оператора **SELECT**. Объединение определяется неявным образом. Каждое объединение задается для двух таблиц, хотя в одном операторе **SELECT** может выполняться несколько объединений. Столбец, по которому выполняется объединение, называется **столбцом соединения** (connecting column) или **столбцом объединения** (join column). Столбцы соединения должны иметь совпадающие или легко сравнимые значения, описывающие одинаковые или сходные данные в объединяемых таблицах.

Столбцы соединения обычно должны иметь одинаковый тип данных. Тогда значения в этих столбцах будут **совместимы для объединения** (join-compatible), так как будут принадлежать к одному общему классу данных.

## Объединение

### Синтаксис операции объединения

В общих чертах, синтаксис операции объединения имеет следующий вид:

SELECT список\_выбора

FROM таблица\_1, таблица\_2 [ , таблица\_3]…

WHERE [таблица\_1.] столбец оператор\_объединения [таблица\_2. ] столбец

В списке таблиц предложения **FROM** по меньшей мере должны содержаться две таблицы, а столбцы в предложении **WHERE** должны быть совместимы для объединения. Если столбцы, по которым выполняется объединение, имеют одинаковые имена, то в списке выбора и в предложении **WHERE** нужно указать соответствующие им таблицы. Например, чтобы узнать имена редакторов книги *Secrets of Silicon Valley*, нужно выполнить объединение по столбцам *ed\_id*, имеющимся в обеих таблицах:

select ed\_lname, ed\_fname, ed\_pos

from editors, titleeditors

where editors.ed\_id = titleeditors.ed\_id

AND titleeditors.title\_id = 'PC8888'

Результат:

|  |  |  |
| --- | --- | --- |
| **ed\_lname** | **ed\_fname** | **ed\_pos** |
| DeLongue | Martinella | project |
| Samuelson | Bernard | project |
| Kaspchek | Christof | acquisition |

В результате этого объединения выбираются имена редакторов со значением "PC8888" в столбце *title\_id*. Объединение выполняется по столбцу *ed\_id,* содержащемуся в обеих таблицах.

## Почему необходимо объединение

Для выполнения сложного анализа данных, скорее всего, их придется собирать из разных таблиц. Используя реляционную модель, в рамках которой правила нормализации рекомендуют разбивать данные на множество таблиц, описывающих отдельные объекты, с помощью операции объединения можно выполнять самые разнообразные запросы и получать сложные отчеты. Таким образом, операция объединения занимает в реляционной модели одно из ключевых мест.

### Объединения и реляционная модель

Операция объединения допускается в СУРБД вследствие того, что в рамках этой модели можно брать данные из разных таблиц и создавать на их основе новые таблицы, и отслеживать непредусмотренные заранее зависимости. Зависимости между данными проявляются при выполнении запроса, а не при создании БД.

Не нужно заранее знать, какие данные будут впоследствии объединяться. Новые зависимости между данными из разных таблиц становятся видны при их объединении. Например, чтобы узнать имена членов организации, которые внесли свой вклад в избирательную кампанию, можно использовать следующий запрос:

select board.members.name

from board\_members, political\_contributors

where board\_members.name = political\_contributors.name

Или, используя нашу БД, с помощью объединения можно найти авторов, которые когда-либо выполняли обязанности редакторов:

select ed\_lname

from editors, authors

where ed\_id = au\_id

Этот запрос с изюминкой. Так как авторы и редакторы могут иметь одинаковые фамилии, объединение выполняется по столбцу с идентификационными номерами, которые однозначно идентифицируют авторов и редакторов. Таким образом, первичные ключи часто могут оказываться весьма полезными и при объединении таблиц.

## Пример объединения

Для выполнения объединения, как минимум, нужно выполнить два действия.

* Поместить имя одной или нескольких таблиц в список предложения **FROM**.
* Добавить условия в предложение **WHERE**, на основе которых будет выполняться объединение по заданным столбцам.

Для определения связи между столбцами, по которым будет выполняться объединение, можно использовать любые реляционные операторы (однако, чаще всего, применяется оператор равенства). Например, чтобы узнать, кто из редакторов живет недалеко от офиса издателя, нужно выполнить соответствующее объединение. В следующем запросе находятся имена редакторов, живущих в том же городе, в котором расположена компания Algodata Infosystems:

select ed\_lname, ed\_id, editors.city, pub\_name, publishers.city

from editors, publishers

where editors.city = publishers.city

AND pub\_name = 'Algodata Infosystems'

Результат:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ed\_lname** | **ed\_id** | **editors.city** | **pub\_name** | **publishers.city** |
| Kaspchek | 943-88-7920 | Berkeley | Algodata Infosystems | Berkeley |
| DeLongue | 321-55-8906 | Berkeley | Algodata Infosystems | Berkeley |

В результате выполнения запроса на объединение находятся два редактора, живущие в том же городе, в котором расположена Algodata Infosystems.

### Проверка правильности объединения

Запросы на объединение, в большей степени, чем другие запросы SQL, нуждаются в тщательной проверке, чтобы быть уверенными, что в результате их выполнения получается именно то, что необходимо. Недостаточно конкретный запрос может вернуть результат, с которым придется долго возиться, чтобы извлечь из него действительно полезную информацию.

## Как получить хорошее объединение

Идеально для объединения подходят ключевые столбцы таблицы – первичные или внешние. Если ключ является составным, то объединение можно выполнять по всем входящим в него столбцам.

Так как первичный ключ логически связан с соответствующими внешними ключами других таблиц, то ключевые столбцы обычно наилучшим образом подходят для объединения.

Чтобы в результате объединения получались полноценные результаты, сравниваемые столбцы должны содержать аналогичные значения – значения, принадлежащие одному общему классу данных. Помимо типа, нужно учитывать семантику, так как, например, возраст авторов и количество проданных книг имеют одинаковый тип данных, но логически не взаимосвязаны.

Типы данных объединяемых столбцов должны быть совместимыми – система должна уметь преобразовывать их друг в друга. Например, такие преобразования могут выполняться между числовыми столбцами (типы integer, decimal, float) или между символьными столбцами (типы character, varchar, datetime).

### Объединения и нулевые значения

Если в столбцах, по которым проводится объединение, имеются нулевые значения (NULL), они пропускаются, так как нет никаких оснований считать одно неизвестное значение совпадающим с другим неизвестным значением.

## Улучшение читаемости результатов объединения

### Выбор столбцов для запросов на объединение

При использовании оператора **SELECT** \* столбцы помещаются в результат в порядке их определения в операторе **CREATE TABLE**. Чтобы найти названия книг, связанные с заказом номер 1, необходимо объединить таблицы *salesdetails* и *titles* по столбцу *title\_id*:

select \*

from titles, sailsdetails

where titles.title\_id = salesdetails.title\_id

AND sonum = 1

Результат:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **title\_id** | **title** | **type** | **pub\_id** | **price** |
| PS2091 | Is Anger the Enemy? | Psychology | 0736 | 10.95 |

продоложение

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **advance** | **ytd\_sales** | **contract** | **notes** | **pubdate** |
| 2275 | 2045 | 1 | Carefully researched study of the effects of strong emotions on body | 15.06.85 |

продолжение

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **sonum** | **qty\_ordered** | **qty\_shipped** | **title\_id** | **date\_shipped** |
| 1 | 75 | 75 | PS2091 | 15.09.85 |

В результате получается таблица с одной строкой и пятнадцатью столбцами.

### Псевдонимы в списке таблиц улучшают читаемость запросов

Чтобы ускорить ввод запросов и сделать их более понятными, в списке таблиц можно определить псевдонимы таблиц (сокращенные имена). Это особенно полезно при выполнении объединения по столбцам с одинаковыми именами, когда всякий раз при их использовании приходится указывать имена соответствующих таблиц.

Вот как можно использовать псевдонимы в запросе, с помощью которого находятся авторы, живущие в одном городе с каким-либо издателем:

select au\_lname, au\_fname

from authors a, publishers p

where a.city = p.city

Результат:

|  |  |
| --- | --- |
| **au\_lname** | **au\_fname** |
| Carson | Cheryl |
| Bennet | Abraham |

## Определение условий объединения

Объединение обычно основывается на равенстве или совпадении значений в объединяемых столбцах. Объединение, основанное на равенстве, легко распознать по логическому оператору "=" в соответствующей объединению части предложения **WHERE**. Объединения также могут быть построены и на других условиях: любом логическом операторе или специальном операторе для определения **внешнего объединения** (outer join).

Помимо равенства, для задания условия могут использоваться следующие логические операторы:

***Символ* *Значение***

> Больше чем

>= Больше или равно

< Меньше чем

<= Меньше или равно

!=(или <>) Не равно

В Transact-SQL также имеются операторы !> и !<, соответственно эквивалентные операторам <= и >=.

На жаргоне РБД объединения, построенные только на операторах сравнения, называются **theta-объединениями**.

### Объединения, основанные на равенстве

Объединения, основанные на равенстве, как бы "сшивают" столбцы из участвующих в объединении таблиц. В следующем запросе находятся названия книг, связанные с определенным номером заказа:

select title, t.title\_id, sonum, sd.title\_id

from titles t, salesdetails sd

where sonum = 14

AND t.title\_id = sd.title\_id

Результат:

|  |  |  |  |
| --- | --- | --- | --- |
| **title** | **t.title\_id** | **sonum** | **sd.title\_id** |
| Computer Phobic | PS1372 | 14 | PS1372 |
| Life Without Fear | PS2106 | 14 | PS2106 |
| Prolonged Data Deprivation | PS3333 | 14 | PS3333 |
| Emotional Security | PS7777 | 14 | PS7777 |

Так как в данном случае нет никакой необходимости в дублировании информации, из результата можно изъять один из столбцов объединения. Такое отображение называется **естественным объединением** (natural join). Предыдущий запрос примет следующий вид:

select title, t.title\_id, sonum

from titles t, salesdetails sd

where sonum = 14

AND t.title\_id = sd.title\_id

Результат:

|  |  |  |
| --- | --- | --- |
| **title** | **t.title\_id** | **sonum** |
| Computer Phobic | PS1372 | 14 |
| Life Without Fear | PS2106 | 14 |
| Prolonged Data Deprivation | PS3333 | 14 |
| Emotional Security | PS7777 | 14 |

Теперь получилось естественное объединение, так как столбец *title\_id* не появляется в результатах дважды. Не имеет значения, из какой таблицы включается столбец *title\_id*, однако в списке выбора необходимо указать его принадлежность определенной таблице.

### Объединения, не основанные на равенствах

Объединения, не основанные на равенствах, могут быть описаны в терминах используемых в них условий, например, "объединение меньше чем", "объединение больше чем" и т.д. В следующем примере объединения типа меньше чем находятся все заказы, отправленные после получения денег:

select distinct s.sonum, s.stor\_id, s.sdate, sd.date\_shipped

from sales s, salesdetails sd

where s.sdate < sd.date\_shipped

AND s.sonum = sd.sonum

Результат:

|  |  |  |  |
| --- | --- | --- | --- |
| **sonum** | **stor\_id** | **sdate** | **date\_shipped** |
| 1 | 7066 | 13.09.85 | 15.09.85 |
| 2 | 7067 | 14.09.85 | 15.09.85 |
| 3 | 7131 | 14.09.85 | 18.09.85 |
| 4 | 7131 | 14.09.85 | 18.09.85 |
| 6 | 8042 | 14.09.85 | 22.09.85 |
| 7 | 6380 | 13.09.85 | 20.09.85 |
| 9 | 8042 | 11.03.88 | 28.03.88 |

В следующем примере по столбцу *title\_id* объединяются таблицы *titles* и *roysched*:

select t.title\_id, t.ytd\_sales, r.royalty

from titles t, roysched r

where t.title\_id = r.title\_id

AND t.ytd\_sales >= r.lorange AND t.ytd\_sales <= r.hirange

Результат:

|  |  |  |
| --- | --- | --- |
| **title\_id** | **ytd\_sales** | **royalty** |
| PC8888 | 4095 | 0.1 |
| BU1032 | 4095 | 0.1 |
| PS7777 | 3336 | 0.1 |
| PS3333 | 4072 | 0.1 |
| BU1111 | 3876 | 0.1 |
| MC2222 | 2032 | 0.12 |
| TC7777 | 4095 | 0.1 |

### Объединение таблицы с самой собой: самообъединение

Самообъединение является другим вариантом объединения на основе равенства. При самообъединении сравниваются значения внутри столбца одной таблицы. Например, самообъединение можно использовать для поиска авторов, живущих в Окленде, Калифорнии, и имеющих одинаковые почтовые коды. Так как в этом запросе таблица *authors* объединяется сама с собой, то она выступает сразу в двух ролях. Поэтому для различения этих ролей в списке таблиц предложения **FROM** ей нужно задать два разных псевдонима – *au1* и *au2*. Эти псевдонимы также будут использоваться при задании имен столбцов. Вот так выглядит этот запрос:

select au1.au\_fname, au1.au\_lname, au1.zip

from authors au1, authors au2

where au1.city = 'Oakland'

AND au1.zip = au2.zip

|  |  |  |
| --- | --- | --- |
| **au\_fname** | **au\_lname** | **zip** |
| Marjorie | Green | 94618 |
| Dirk | Stringer | 94609 |
| Dirk | Stringer | 94609 |
| Dirk | Stringer | 94609 |
| Dick | Straight | 94609 |
| Dick | Straight | 94609 |
| Dick | Straight | 94609 |
| Livia | Karsen | 94609 |
| Stearns | MacFeather | 94612 |

В этих результатах достаточно сложно разобраться и они не кажутся правильными. Чтобы прояснить ситуацию, нужно сначала устранить повторяющиеся строки для трех авторов. Для этого используем в списке выбора предложение **DISTINCT**:

select distinct au1.au\_fname, au1.au\_lname, au1.zip

from authors au1, authors au2

where au1.city = 'Oakland'

AND au1.zip = au2.zip

Результат:

|  |  |  |
| --- | --- | --- |
| **au\_fname** | **au\_lname** | **zip** |
| Dick | Straight | 94609 |
| Dirk | Stringer | 94609 |
| Livia | Karsen | 94609 |
| Marjorie | Green | 94618 |
| Stearns | MacFeather | 94612 |

Теперь в результате нет повторяющихся строк, но при этом в нем просто перечисляются все авторы, живущие в Окленде, а не только имеющие одинаковые почтовые коды. Что происходит?

При самообъединении все значения почтовых кодов сравниваются сами с собой и поэтому в результат попадают все авторы из Окленда. Чтобы устранить из результата авторов, чьи почтовые коды совпадают только с их же собственными кодами, в предложение **WHERE** нужно добавить дополнительное условие:

select distinct au1.au\_fname, au1.au\_lname, au1.zip

from authors au1, authors au2

where au1.city = 'Oakland'

AND au1.zip = au2.zip

AND au1.au\_id != au2.au\_id

Результат:

|  |  |  |
| --- | --- | --- |
| **au\_fname** | **au\_lname** | **zip** |
| Dick | Straight | 94609 |
| Dirk | Stringer | 94609 |
| Livia | Karsen | 94609 |

### Использование при самообъединении оператора неравенства

В предыдущем примере для устранения повторяющихся строк при самообъединении использовался оператор неравенства. Вместо него также может применяться ключевое слово NOT. Выражение *NOT имя\_столбца = имя\_столбца* эквивалентно выражению *имя\_столбца != имя\_столбца.* Таким образом, предыдущий запрос можно записать в таком виде:

select distinct au1.au\_fname, au1.au\_lname, au1.zip

from authors au1, authors au2

where au1.city = 'Oakland'

AND au1.zip = au2.zip

AND NOT au1.au\_id != au2.au\_id

В качестве другого примера использования при самообъединении оператора неравенства рассмотрим запрос, с помощью которого находятся значения идентификаторов книг и авторов для всех книг, написанных в соавторстве. Другими словами, находятся все строки в таблице *titleauthors* с одинаковыми значениями *title\_id*, но разными значениями *au\_id.*

select distinct t1.title\_id, t1.au\_id

from titleauthors t1, titleauthors t2

where t1.title\_id = t2.title\_id

AND t1.au\_id != t2.au\_id

order by t1.title\_id

Результат:

|  |  |
| --- | --- |
| **title\_id** | **au\_id** |
| BU1032 | 213-46-8915 |
| BU1032 | 409-56-7008 |
| BU1111 | 267-41-2394 |
| BU1111 | 724-80-9391 |
| MC3021 | 722-51-5454 |
| MC3021 | 899-46-2035 |
| PC8888 | 427-17-2319 |
| PC8888 | 846-92-7186 |

### Объединение нескольких таблиц

Таблица *titleauthors* хорошо иллюстрирует ситуацию, когда может потребоваться объединение более двух таблиц. Для получения полной информации о книгах и их авторах в случае БД bookbiz необходимо объединить три таблицы.

Например, чтобы найти названия всех книг определенного типа (*trad\_cook*) и имена их авторов, нужно выполнить следующий запрос:

select au\_lname, au\_fname, title

from auhtors a, titles t, titleauthors ta

where a.au\_id = ta.titleauthors\_id

AND t.title\_id = ta.title\_id

AND t.type = 'trad\_cook'

Результат:

|  |  |  |
| --- | --- | --- |
| **au\_lname** | **au\_fname** | **title** |
| O'Leary | Michael | Sushi, Anyone? |
| Gringlesby | Burt | Sushi, Anyone? |
| Yokomoto | Akiko | Sushi, Anyone? |
| Blotchet-Halls | Reginald | Fifty Years in Buckingham Palace |
| Panteley | Sylvia | Onions, Leeks, and Garlic |

### Внешние объединения

При выполнении описываемых до сих пор объединений в результат включались только строки, удовлетворяющие условиям объединения. Однако вы можете захотеть поместить туда также и строки одной из таблиц, не удовлетворяющие условиям объединения.

При внешнем объединении обычно в результат дополнительно включаются строки из первой таблицы, не удовлетворяющие условиям объединения. С помощью другого оператора внешнего объединения в результат можно включить все строки из второй таблицы, не удовлетворяющие условиям объединения, или даже такие строки из обеих таблиц. SQL поддерживает только либо левое (LEFT), либо правое (RIGHT) внешнее объединение.

Для определения внешнего объединения в предложении **WHERE** используются следующие обозначения:

***Символ Значение***

\*= Включить все строки первой таблицы

=\* Включить все строки второй таблицы

В результате выполнения запроса, в котором ищутся авторы, живущие в одном городе с издателем, возвращаются два имени (Abraham Bennet и Cheryl Carson).

Чтобы включить в результат все имена из таблицы *authors*, независимо от того, удовлетворяют ли они условиям объединения, можно использовать внешнее объединение:

select au\_fname, au\_lname, pub\_name

from authors, publishers

where authors.city \*= publishers.city

Результат:

|  |  |  |
| --- | --- | --- |
| **au\_fname** | **au\_lname** | **pub\_name** |
| Abraham | Bennet | Algodata Infosystems |
| Marjorie | Green | NULL |
| Cheryl | Carson | Algodata Infosystems |
| Albert | Ringer | NULL |
| Anne | Ringer | NULL |
| Michel | DeFrance | NULL |
| Sylvia | Panteley | NULL |

Оператор внешнего объединения "\*=" заставляет систему включить в результат все строки из первой таблицы (в нашем случае *authors*), независимо от того, совпадают ли значения в ее столбце *city* с соответствующими значениями из таблицы *publishers*.

Оператор внешнего объединения "=\*" заставляет систему включить в результат все строки из второй таблицы, независимо от того, совпадают ли их значения со значениями из первой таблицы.

Подставив в предыдущий запрос оператор правого внешнего объединения, отобразятся имена издателей, находящихся в городе, в котором живет какой-либо автор, а также имена издателей, не удовлетворяющих этому условию:

select au\_fname, au\_lname, pub\_name

from authors, publishers

where authors.city =\* publishers.city

Результат:

|  |  |  |
| --- | --- | --- |
| **au\_fname** | **au\_lname** | **pub\_name** |
| NULL | NULL | New Age Books |
| NULL | NULL | Binnet & Hardley |
| Cheryl | Carson | Algodata Infosystems |
| Abraham | Bennet | Algodata Infosystems |

Как и в случае обычных объединений, для ограничения результатов внешнего объединения можно использовать условные операторы. Для начала следует рассмотреть объединение, основанное на равенстве, а затем сравнить его результаты с результатами внешнего объединения. Например, чтобы найти названия книг, количество проданных экземпляров которых превысило 50, можно использовать следующий запрос:

select sonum, title

from salesdetails sd, titles t

where qty\_ordered > 50

AND sd.title\_id = t.title\_id

Результат:

|  |  |
| --- | --- |
| **sonum** | **title** |
| 1 | Is Anger the Enemy? |

Чтобы, кроме того, найти все названия книг, не удовлетворяющих этому условию, нужно воспользоваться внешним объединением:

select sonum, title

from salesdetails sd, titles t

where qty\_ordered > 50

AND sd.title\_id =\* t.title\_id

|  |  |
| --- | --- |
| **sonum** | **title** |
| NULL | Secrets of Silicon Valley |
| NULL | The Busy Executive's Database Guide |
| NULL | Emotional Security |
| NULL | Prolonged Data Deprivation |
| NULL | Cooking with Computers |
| NULL | Silicon Valley Gastronomic |
| NULL | Sushi, Anyone? |
| NULL | Fifty Years in Buckingham Palace |
| NULL | But Is It User Friendly? |
| NULL | The Psychology of Computer Cooking |
| NULL | You Can Combat Computer Stress! |
| 1 | Is Anger the Enemy? |
| NULL | Life Without Fear |

## Виды оператора JOIN

Для примеров будем использовать две таблицы: *users* и *departments*.

Таблица *users*:

|  |  |  |
| --- | --- | --- |
| **us\_id** | **us\_name** | **dep\_id** |
| 1 | Vladimir | 1 |
| 2 | Anton | 2 |
| 3 | Alexander | 6 |
| 4 | Boris | 2 |
| 5 | Yuriy | 4 |

Таблица *departments*:

|  |  |
| --- | --- |
| **dep\_id** | **dep\_name** |
| 1 | sales |
| 2 | support |
| 3 | finance |
| 4 | logistic |

### INNER JOIN

Оператор внутреннего соединения **INNER JOIN** соединяет две таблицы. Порядок таблиц для оператора неважен, поскольку оператор является симметричным.

select u.us\_id, u.us\_name, d.dep\_name

from users u inner join departments d on u.dep\_id = d.dep\_id

Результат:

|  |  |  |
| --- | --- | --- |
| **us\_id** | **us\_name** | **dep\_name** |
| 1 | Vladimir | Sales |
| 2 | Anton | Support |
| 4 | Boris | Support |
| 5 | Yuriy | Logistic |

В результате отсутствует пользователь Alexander с id = 3, так как он принадлежит к dep\_id = 6, которого не существует. Также в выборке нет отдела dep\_id = 3 Finance, так как ни один из сотрудников не принадлежит к указанному департаменту. В данном случае объединяются лишь совпадающие данные из объединяемых таблиц.

### OUTER JOIN

Оператор внешнего соединения **OUTER JOIN** соединяет две таблицы с обязательным вхождением в результат выборки всех строк либо одной, либо обеих таблиц. Такое объединение вернет данные из обеих таблиц (совпадающие по условию объединения) и дополнит выборку оставшимися данными из внешней таблицы, которые по условию не подходят, заполнив недостающие данные значением NULL.

#### LEFT OUTER JOIN

Оператор **левого внешнего объединения** **LEFT OUTER JOIN** соединяет две таблицы. Порядок таблиц для оператора важен, поскольку он не является симметричным. В нашем примере внешней таблицей будет находящаяся слева в запросе таблица *users*.

select u.us\_id, u.us\_name, d.dep\_name

from users u left outer join departments d on u.dep\_id = d.dep\_id

|  |  |  |
| --- | --- | --- |
| **us\_id** | **us\_name** | **dep\_name** |
| 1 | Vladimir | Sales |
| 2 | Anton | Support |
| 3 | Alexander | NULL |
| 4 | Boris | Support |
| 5 | Yuriy | Logistic |

Если добавить в данный запрос фильтр поиска авторов, которые принадлежат неизвестному департаменту, то получится следующий результат:

select u.us\_id, u.us\_name, d.dep\_name

from users u left outer join departments d on u.dep\_id = d.dep\_id

where d.dep\_id IS NULL

|  |  |  |
| --- | --- | --- |
| **us\_id** | **us\_name** | **dep\_name** |
| 3 | Alexander | NULL |

#### RIGHT OUTER JOIN

Оператор **правого внешнего объединения RIGHT OUTER JOIN** соединяет две таблицы. Порядок таблиц для оператора важен, так как он не является симметричным. В нашем примере внешней таблицей будет находящаяся слева в запросе таблицы *departments*.

select u.us\_id, u.us\_name, d.dep\_name

from users u right outer join departments d on u.dep\_id = d.dep\_id

|  |  |  |
| --- | --- | --- |
| **us\_id** | **us\_name** | **dep\_name** |
| 1 | Vladimir | Sales |
| 2 | Anton | Support |
| 4 | Boris | Support |
| NULL | NULL | Finance |
| 5 | Yuriy | Logistic |

В результате получили список всех департаментов и соответствующих им пользователей. При этом в департаменте Finance идентификатор пользователя и его имя заполнились значениями NULL.

#### FULL OUTER JOIN

Оператор **полного внешнего объединения FULL OUTER JOIN** соединяет две таблицы. Порядок таблиц для оператора неважен, поскольку он является симметричным. **FULL OUTER JOIN** возвращает "объединение" объединений LEFT и RIGHT таблиц, комбинируя результат двух запросов.

select u.us\_id, u.us\_name, d.dep\_name

from users u full outer join departments d on u.dep\_id = d.dep\_id

|  |  |  |
| --- | --- | --- |
| **us\_id** | **us\_name** | **dep\_name** |
| 1 | Vladimir | Sales |
| 2 | Anton | Support |
| 3 | Alexander | NULL |
| 4 | Boris | Support |
| 5 | Yuriy | Logistic |
| NULL | NULL | Finance |

Изначально система возвращает объединение по левому внешнему объединению, а затем по правому, как бы воссоединяя **LEFT OUTER JOIN** и **RIGHT OUTER JOIN**.

### CROSS JOIN

Оператор **перекрестного соединения** **CROSS JOIN** соединяет две таблицы. Порядок таблиц для оператора неважен, поскольку оператор является симметричным. **CROSS JOIN** возвращает перекрестное объединение двух таблиц. Результатом будет выборка всех записей первой таблицы, объединенная с каждой строкой второй таблицы. Важным моментом является то, что для **CROSS JOIN** не нужно указывать условие объединения.

select us\_id, us\_name, dep\_name

from users cross join departments

В результате выбираются все возможные варианты объединения двух таблиц. Альтернативой данному запросу будет:

select \*

from users, departments

Только в последнем случае в выборке мы увидим абсолютно все столбцы этих двух таблиц, а не только *us\_id, us\_name* и *dep\_name*.

## Оператор UNION

Хотя оператор **UNION** не выполняет объединение, он позволяет объединить результаты нескольких запросов. Этот оператор полезен, когда нужно одновременно увидеть аналогичные данные из различных таблиц. Упрощенный синтаксис оператора **UNION** имеет следующий вид:

оператор\_select

UNION

оператор\_select

В результате выполнения следующего запроса находятся все авторы и редакторы, живущие в Окленде или Беркли:

select au\_fname, au\_lname, city

from authors

where city in ('Oakland', 'Berkeley')

UNION

select ed\_fname, ed\_lname, city

from editors

where city in ('Oakland', 'Berkeley')

Нужно обратить внимание, что в обоих запросах используется одинаковое число элементов с совместимыми типами данных. В качестве заголовков столбцов **UNION** использует имена столбцов из первого запроса.

Результат:

|  |  |  |
| --- | --- | --- |
| **au\_fname** | **au\_lname** | **city** |
| Abraham | Bennet | Berkeley |
| Bernard | Samuelson | Oakland |
| Cheryl | Carson | Berkeley |
| Christof | Kaspchek | Berkeley |
| Dick | Straight | Oakland |
| Dirk | Stringer | Oakland |
| Livia | Karsen | Oakland |
| Marjorie | Green | Oakland |
| Martinella | DeLongue | Berkeley |
| Stearns | MacFeather | Oakland |

По умолчанию оператор **UNION** устраняет из результата повторяющиеся строки. Это может привести к определенным недоразумениям, особенно если в запросе фигурирует только один столбец. Например, в таблице *authors* присутствуют 23 города, а в таблице *publishers* – 3. При выполнении оператора **UNION** в результате получается только 18 строк.

select count(city)

from authors

Результат:

-----

23

select count(city)

from publishers

Результат:

-----

3

select city

from authors

UNION

select city

from publishers

Результат:

|  |
| --- |
| **city** |
| Ann Arbor |
| Berkeley |
| Boston |
| Corvallis |
| Covelo |
| Gary |
| Lawrence |
| Menlo Park |
| Nashville |
| Oakland |
| Palo Alto |
| Rockville |
| Salt Lake City |
| San Francisco |
| San Jose |
| Vacaville |
| Walnut Creek |
| Washington |

Оператор **UNION** устранил из результата все повторяющиеся строки. Чтобы отобразить все строки, после оператора **UNION** нужно добавить ключевое слово **ALL**.

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

Пусть, например, нужно получить список книг, указав для каждой из них процентное снижение цены и новую стоимость. Стоимость книг до $7 снижается на 20%, между $7 и $15 – на 10%, выше $15 – на 30%.

Без оператора **UNION** потребовалось бы выполнить три отдельных запроса и поместить результаты в новую таблицу. Оператор **UNION** выполняет все это за один проход:

select '20% off' AS discount, title, price AS old, price \* .80 AS new

from titles

where price < $7.00

UNION

select '10% off', title, price, price \* .90

from titles

where price BETWEEN $7.00 AND $15.00

UNION

select '30% off', title, price, price \* .70

from titles

where price > $15.00

Результат:

|  |  |  |  |
| --- | --- | --- | --- |
| **discount** | **title** | **old** | **new** |
| 10% off |  | 11.95 | 10.755 |
| 10% off |  | 7.99 | 7.191 |
| 10% off |  | 11.95 | 10.755 |
| 10% off |  | 10.95 | 9.855 |
| 10% off |  | 7 | 6.3 |
| 10% off |  | 14.99 | 13.491 |
| 20% off |  | 2.99 | 2.392 |
| 20% off |  | 2.99 | 2.392 |
| 30% off |  | 22.95 | 16.065 |
| 30% off |  | 21.59 | 15.113 |
| 30% off |  | 20.95 | 14.665 |
| 30% off |  | 19.99 | 13.993 |
| 30% off |  | 20 | 14 |
| 30% off |  | 19.99 | 13.993 |
| 30% off |  | 19.99 | 13.993 |
| 30% off |  | 19.99 | 13.993 |

Чтобы отсортировать результаты по цене, можно воспользоваться предложением **ORDER BY**.

## Подзапросы

### Подзапросы, начинающиеся с операторов сравнения и включающие ключевые слова ANY или ALL

***Что такое ALL и ANY.*** Если в качестве примера воспользоваться оператором сравнения ">", то "> ALL" означает "больше, чем каждое значение" (другими словами, "больше, чем наибольшее значение"). Таким образом, "> ALL (1, 2, 3)" означает "больше, чем 3". При этом "> ANY" означает "больше, чем, по крайней мере, одно значение" (другими словами, "больше, чем наименьшее значение"). Таким образом, "> ANY (1, 2, 3)" означает "больше, чем 1". Различия между этими ключевыми словами можно посмотреть в таблице:

|  |  |  |  |
| --- | --- | --- | --- |
| **ALL** | **Результат** | **ANY** | **Результат** |
| > ALL (1, 2, 3) | > 3 | > ANY (1, 2, 3) | > 1 |
| < ALL (1, 2, 3) | < 1 | < ANY (1, 2, 3) | < 3 |
| = ALL (1, 2, 3) | = 1 или = 2 или = 3 | = ANY (1, 2, 3) | = 1 или = 2 или = 3 |

***Подзапросы с ключевым словом ALL***. Можно, например, задать следующий вопрос: "Аванс за какие книги превышает аванс за любую книгу, опубликованную издательством New Age Books?" Этот вопрос можно было бы перефразировать, чтобы прояснить его "перевод" на язык SQL: "Аванс за какие книги превышает наибольший аванс, выплаченный издательством New Age Books?" Ключевое слово **ALL** (но не ключевое слово **ANY!**) – именно то, что требуется в данном случае:

select title

from titles

where advance > ALL

(select advance

from publishers, titles

where titles.pub\_id = publishers.pub\_id

AND pub\_name = 'New Age Books')

Результат:

title

-----

The Gourmet Microwave

Для каждого названия внутренний запрос находит список значений авансов, выплаченных издательством New Age Books. Внешний запрос находит наибольшее значение в списке и определяет, не выплачен ли за книгу, которая рассматривается в данный момент, еще больший аванс.

Если внутренний подзапрос, начинающийся с **ALL** и оператора сравнения, возвращает в качестве одного из своих значений NULL, считается, что запрос в целом завершился неудачно. Например, значения авансов за книги издательства Algodata Infosystems. Например, значения авансов за книги издательства Algodata Infosystems выглядят так:

select advance

from publishers, titles

where titles.pub\_id = publishers.pub\_id

AND pub\_name = 'Algodata Infosystems'

Результат:

|  |
| --- |
| **advance** |
| 8000 |
| NULL |
| 5000 |
| 5000 |
| 5000 |
| 7000 |

Если пользователя интересуют авансы, превышающие любые авансы, выплаченные издательством Algodata Infosystems, он не получит никаких результатов, поскольку невозможно сказать, что больше значения NULL.

select title

from titles

where advance > ALL

(select advance

from publishers, titles

where titles.pub\_id = publishers.pub\_id

AND pub\_name = 'Algodata Infosystems')

Результат:

title

-----

***Подзапросы с ключевым словом ANY.*** Запрос с ключевым словом **ANY** находит значения, превышающие "некоторое" значение из подзапроса. Приведенный ниже запрос находит книги, за которые был выплачен аванс, превышающий минимальное значение аванса ($5000), выплаченного издательством Algodata Infosystems.

select title, advance

from titles

where advance > ANY

(select advance

from titles, publishers

where titles.pub\_id = publishers.pub\_id

AND pub\_name = 'Algodata Infosystems')

Результат:

|  |  |
| --- | --- |
| **title** | **advance** |
| Secrets of Silicon Valley | 8000 |
| Sushi, Anyone? | 8000 |
| But Is It User Friendly? | 7000 |
| You Can Combat Computer Stress! | 10125 |
| Life Without Fear | 6000 |
| The Gourmet Microwave | 15000 |
| Onions, Leeks, and Garlic | 7000 |
| Computer Phobic and Non-Phobic Individuals | 7000 |

Для каждой книги внутренний запрос находит список величин аванса, выплаченных издательством Algodata Infosystems. Внешний запрос просматривает все значения в этом списке и определяет, не выплачен ли за книгу, которая рассматривается в данный момент, больший, чем любая из этих величин. Если подзапрос не возвращает никаких значений, то считается, что запрос в целом завершился неудачно.

***Сравнение ключевых слов IN, ANY, ALL*.** Оператор "= ANY" полностью эквивалентен оператору IN.

Однако оператор "! = ANY" отличается от оператора NOT IN. "! = ANY" означает "не = а или не = b или не = с", в то время как NOT IN означает "не = а и не = b и не = с". Допустим, нужно найти авторов, которые проживают в городе, в котором нет ни одного издательства. Для этого можно было бы воспользоваться следующим запросом:

select au\_lname, au\_fname

from authors

where city != ANY

(select city

from publishers)

Результат:

|  |  |
| --- | --- |
| **au\_lname** | **au\_fname** |
| Bennet | Abraham |
| Green | Marjorie |
| Carson | Cheryl |
| Ringer | Albert |
| Ringer | Anne |
| DeFrance | Michel |
| Panteley | Sylvia |

Полученные результаты включают всех авторов. Это произошло потому, что каждый автор проживает в каком-то городе, где нет какого-нибудь издательства (поскольку каждый автор живет в одном и только одном городе). Внутренний запрос находит все города, в которых расположены издательства, а затем внешний запрос – для каждого города – находит авторов, которые там живут. Вот что произойдет, если в этот запрос подставить оператор NOT IN:

select au\_lname, au\_fname

from authors

where city NOT IN

(select city

from publishers)

order by city

Результат:

|  |  |  |
| --- | --- | --- |
| **au\_lname** | **au\_fname** | **city** |
| del Castillo | Innes | Ann Arbor |
| Blotchet-Halls | Reginald | Corvallis |
| Gringlesby | Burt | Covelo |
| DeFrance | Michel | Gary |
| Smith | Meander | Lawrence |
| White | Johnson | Menlo Park |
| Greene | Morningstar | Nashville |

Именно эти результаты и нужно было получить. Они включают всех авторов, за исключением Cheryl Carson и Abraham Bennet, которые проживают в Беркли, где расположено издательство Algodata Infosystems.

## Подзапросы, возвращающие единственное значение

Подзапрос, начинающийся с **немодифицированного оператора сравнения** (оператор сравнения, не сопровождаемый ключевыми словами ANY или ALL), должен возвращать единственное значение. (В противном случае система выдаст сообщение об ошибке, и запрос не будет обработан).

Если, например, предположить, что каждое издательство размещается только в одном городе, и хотите найти фамилии авторов, проживающих в городе, где размещается издательство Algodata Infosystems, то можно написать оператор SQL с подзапросом, начинающимся с простого оператора сравнения "=":

select au\_lname, au\_fname

from authors

where city =

(select city

from publishers

where pub\_name = 'Algodata Infosystems')

Результат:

|  |  |
| --- | --- |
| **au\_lname** | **au\_fname** |
| Bennet | Abraham |
| Carson | Cheryl |

### Агрегирующие функции гарантируют единственное значение

Подзапросы с оператором сравнения часто включают агрегирующие функции, поскольку эти функции гарантированно возвращают единственное значение. Например, если требуется найти названия всех книг с ценами выше текущей минимальной цены, надо выполнить следующий запрос:

select title

from titles

where price >

(select min(price)

from titles)

Результат:

|  |
| --- |
| **title** |
| Secrets of Silicon Valley |
| The Busy Executive's Database Guide |
| Emotional Security |
| Prolonged Data Deprivation |
| Cooking with Computers |

Сначала внутренний запрос находит минимальную цену в таблице *titles*, затем внешний запрос использует это значение для выбора соответствующих названий.

## Подзапросы, выполняющие проверку на существование

Когда тот или иной подзапрос начинается с ключевого слова **EXISTS**, этот подзапрос функционирует как "тест на существование". Ключевое слово **EXISTS** в предложении **WHERE** выполняет проверку на существование (или несуществование) данных, которые удовлетворяют критериям соответствующего подзапроса.

Если необходимо найти названия всех издательств, которые публикуют книги по бизнесу, надо выполнить следующий запрос:

select distinct pub\_name

from publishers

where exists

(select \*

from titles

where pub\_id = publishers.pub\_id

AND type = 'business')

Результат:

|  |
| --- |
| **pub\_name** |
| Algodata Infosystems |
| New Age Books |

**EXISTS** выполняет проверку на наличие или отсутствие "пустого набора" строк. Если подзапрос возвращает хотя бы одну строку, этот результат оценивается как "истина". Это означает, что результат выполнения **EXISTS** будет успешным, а результат выполнения **NOT EXISTS** будет неудачным. Если подзапрос возвращает пустой набор (строк нет), этот результат оценивается как "ложь". Это означает, что результат выполнения фразы **NOT EXISTS** будет успешным, а результат выполнения фразы **EXISTS** будет неудачным.

Нужно обратить внимание, что синтаксис подзапросов, начинающихся с **EXISTS**, отличается от синтаксиса других подзапросов в следующих отношениях:

* Ключевому слову **EXISTS** не предшествует имя столбца, константа или какое-то другое выражение.
* Список выбора подзапроса, начинающегося с **EXISTS**, почти всегда состоит из "звездочки" (\*). Нет никакого смысла вводить имена столбцов, поскольку вы лишь выполняете тест на существование строк, которые удовлетворяют условиям подзапроса, а они указываются в предложении **WHERE** этого подзапроса (а не в предложении **SELECT** этого подзапроса).

### NOT EXISTS отыскивает пустой набор

**NOT EXISTS** выполняет действие, обратное **EXISTS**. Успешный результат выполнения запросов с **NOT EXISTS** соответствует случаю, когда подзапрос вообще не возвращает строк. Чтобы найти, например, названия издательств, которые не публикуют книги по бизнесу, запрос должен иметь следующий вид:

select pub\_name

from publishers

where NOT EXISTS

(select \*

from titles

where pub\_id = publishers.pub\_id

AND type = 'business')

Результат:

pub\_name

-----

Binnet & Hardley

Следующий запрос позволяет найти названия книг, ни одна из которых еще не была продана:

select title

from titles

where NOT EXISTS

(select title\_id

from salesdetails

where title\_id = titles.title\_id)

Результат:

|  |
| --- |
| **title** |
| The Psychology of Computer Cooking |
| Net Etiquette |

### Использование EXISTS для поиска пересечения и разности

Подзапросы, начинающиеся с **EXISTS** и **NOT EXISTS**, можно использовать для выполнения двух операций из теории множеств: **пересечения** и **разности**. Пересечение двух множеств содержит все элементы, которые принадлежат обоим исходным множествам. Разность двух множеств содержит элементы, которые принадлежат только первому из двух множеств.

Пересечением *authors* и *publishers* по столбцу *city* является множество городов, в которых проживает какой-то автор и расположено какое-то издательство:

select distinct city

from authors

where EXISTS

(select \*

from publishers

where authors.city = publishers.city)

Результат:

city

-----

Berkeley

Разностью между *authors* и *publishers* по столбцу *city* является множество городов, в которых проживает какой-то автор, но нет никакого издательства (то есть все города, за исключением Беркли):

select distinct city

from authors

where NOT EXISTS

(select \*

from publishers

where authors.city = publishers.city)

Результат:

|  |
| --- |
| **city** |
| Ann Arbor |
| Corvallis |
| Covelo |
| Gary |
| Lawrence |
| Menlo Park |
| Nashville |

## Подзапросы с разными уровнями вложения

Подзапрос сам может включать один или несколько подзапросов. Можно организовать вложенную структуру, включающую любое число подзапросов.

Примером задачи, которая может быть решена с помощью оператора с несколькими уровнями вложенных запросов, является следующая: "Найти фамилии авторов, участвовавших в написании по крайней мере одной книги по компьютерам".

select au\_lname, au\_fname

from authors

where au\_id in

(select au\_id

from titleauthors

where title\_id in

(select title\_id

from titles

where type = 'popular\_comp'))

Результат:

|  |  |
| --- | --- |
| **au\_lname** | **au\_fname** |
| Carson | Cheryl |
| Dull | Ann |
| Hunter | Sheryl |
| Locksley | Chastity |

Самый внутренний запрос возвращает идентификационные номера названий PC1035, PC8888, PC9999. Запрос на следующем по старшинству уровне выполняется на основе этих идентификаторов названий и возвращает идентификационные номера авторов. Наконец, внешний запрос использует найденные идентификаторы авторов для поиска их фамилий.

Этот запрос также можно представить в виде объединения:

select au\_lname, au\_fname

from authors inner join titles on authors.au\_id = titleauthors.au\_id

inner join titleauthors on titles.title\_id = titleauthors.title\_id

where type = 'popular\_comp'

## Подзапросы в операторах UPDATE, DELETE и INSERT

Подзапросы могут быть вложены в операторы **UPDATE**, **DELETE** и **INSERT** так же, как и в оператор **SELECT**.

Приведенный ниже запрос увеличивает в два раза цену всех книг, опубликованных издательством New Age Books. Данный оператор обновляет таблицу *titles*; в его подзапросе есть обращение к таблице *publishers*.

update titles

set price = price \* 2

where pub\_id in

(select pub\_id

from publishers

where pub\_name = 'New Age Books')

Эквивалентный оператор **UPDATE**, в котором используется объединение (для систем, которые допускают наличие предложения **FROM** в операторе **UPDATE**), имеет следующий вид:

update titels

set price = price \* 2

from titles inner join publishers on titles.pub\_id = publishers.pub\_id

where pub\_name = 'New Age Books'

С помощью следующего вложенного оператора **SELECT** можно удалить все записи, связанные с заказами на продажи книг по бизнесу:

delete salesdetails

where title\_id in

(select title\_id

from titles

where type = 'business')

Эквивалентный оператор **DELETE**, в котором используется объединение (для систем, которые в операторе **DELETE** допускают наличие нескольких таблиц в предложении **FROM**), имеет следующий вид:

delete salesdetails

from salesdetails inner join titles on salesdetails.title\_id = title.title\_id

where type = 'business'

# Создание и использование виртуальных таблиц (курсоров)

Подобно операции объединения, курсоры являются неотъемлемой частью реляционной модели. Курсор создается с помощью оператора **SELECT** и обеспечивает гибкость при анализе и обработке данных. Курсор можно представлять себе как подвижный кадр или окно, через которое пользователь видит данные.

## Создание курсоров

Вот упрощенный синтаксис оператора, определяющего курсор:

CREATE VIEW имя\_курсора [ (имя\_столбца [, имя\_столбца]…) ]

AS

SELECT\_оператор

В следующем примере создается курсор, который отображает фамилии авторов, живущих в Окленде и Калифорнии, а также названия их книг:

create view oaklanders

as

select au\_fname, au\_lname, title

from authors inner join titles on authors.au\_id = titleauthors.au\_id

join titleauthors on titles.title\_id = titleauthors.title\_id

where city = 'Oakland'

select \*

from oaklanders

Результат:

|  |  |  |
| --- | --- | --- |
| **au\_fname** | **au\_lname** | **title** |
| Marjorie | Green | The Busy Executive's Database Guide |
| Marjorie | Green | You Can Combat Computer Stress! |
| Dick | Straight | Straight Talk About Computers |
| Livia | Karsen | Computer Phobic and Non-Phobic Individuals |
| Stearns | MacFeather | Computer Phobic and Non-Phobic Individuals |
| Stearns | MacFeather | Cooking with Computers |

Первая строка оператора **CREATE VIEW** присваивает курсору имя, а последующий оператор **SELECT** определяет его. Как видно, вовсе не обязательно, чтобы оператор **SELECT** выполнял простой выбор строк и столбцов в одной конкретной таблице. Курсор можно создать на основе нескольких таблиц, других курсоров или того и другого; при этом оператор **SELECT** может иметь почти любую сложность, а для определения столбцов и строк, которые требуется включить в курсор, используются операции проектирования и выбора.

### Удаление курсоров

Синтаксис удаления курсоров выглядит следующим образом:

DROP VIEW имя\_курсора

Для удаления курсора *oaklanders* используется следующий запрос:

drop view oaklanders

Если какой-то курсор зависит от таблицы (или другого курсора), которая была удалена, им не удастся воспользоваться. Однако при создании под тем же именем новой таблицы (или курсора), которая заменит соответствующий удаленный объект, использование этого курсора становится возможным.

## Преимущества курсоров

С курсором можно работать как с обычной таблицей. К примеру, руководителю отдела рекламы может понадобиться упорядочить результаты просмотра в алфавитном порядке (по фамилиям авторов). Например:

select \*

from books

order by au\_lname

В данном случае курсором является *books*.

Бухгалтеру может потребоваться другое представление. Его не интересует, кто из авторов является первым, а кто вторым. Допустим, что бухгалтера интересует только нижняя строка: на чье имя должны быть выписаны чеки и на какую сумму. Такой запрос связан с вычислением того, сколько книг было продано и по какой цене, а также какой процент от суммы полагается каждому автору. Для создания такого курсора ему необходимо выполнить следующий запрос:

create view royaltychecks

as

select au\_fname, au\_lname, sum (price \* ytd\_sales \* royaltyshare) as Total\_Income

from authors inner join titles on authors.au\_id = titleauthors.au\_id

inner join titelauthors on titles.title\_id = titleauthors.title\_id

inner join roysched on titles.title\_id = roysched.title\_id

where ytd\_sales BETWEEN lorange AND hirange

group by au\_lname, au\_fname

Для отображения созданного курсора нужно выполнить следующий запрос:

select \*

from royaltychecks

Результаты (кто получает чек и на какую сумму) будут иметь следующий вид:

|  |  |  |
| --- | --- | --- |
| **au\_lname** | **au\_fname** | **Total\_Income** |
| Bennet | Abraham | 4911.54 |
| Yokomoto | Akiko | 2455.36 |
| Ringer | Albert | 1421.27 |
| Dull | Ann | 4095.00 |
| Ringer | Anne | 4669.35 |
| Gringlesby | Burt | 1841.52 |
| Locksley | Chastity | 2665.46 |

### Причины использования курсоров

Курсорами можно пользоваться для "фокусировки", упрощения и "настройки" БД под конкретного пользователя. Кроме того, с помощью курсоров реализуется механизм обеспечения безопасности. Наконец, они могут защитить пользователей при изменении структуры БД.

### Правила присвоения имен столбцам курсора

Столбцам курсора можно присваивать псевдонимы. Если не задать имена в операторе **CREATE TABLE**, столбцы курсора "наследуют" их от столбцов в таблице (таблицах), на основе которой простроен данный курсор. При необходимости установления новых имен нужно указать их в скобках вслед за именем курсора, отделяя их друг от друга запятыми.

Бывают, однако, ситуации, когда новые имена для столбцов курсора являются обязательными. (В случае переименования какого-либо столбца требуется перечисление всех столбцов):

* Один или несколько столбцов курсора получаются с помощью некоторого арифметического преобразования, встроенной функции или константы.
* Курсору приходится иметь дело с несколькими столбцами под одним и тем же именем (так как разные столбцы в объединенных таблицах могут иметь одно и то же имя).

Первую ситуацию можно проиллюстрировать оператором **CREATE VIEW** для курсора с именем *currentinfo*.

create view currentinfo(PUB#, TYPE, INCOME, AVG\_PRICE, AVG\_SALES)

as

select pub\_id, type, sum(price \* ytd\_sales), avg(price), avg(ytd\_sales)

from titles

group by pub\_id, type

Вычисляемые столбцы в списке выбора на самом деле не имеют имен, поэтому необходимо присвоить их в операторе **CREATE VIEW**. В противном случае невозможно будет сделать ссылку на них. Если начинается работа с курсором *currentinfo*, имеющим присвоенные имена, необходимо пользоваться новыми именами:

select PUB#, AVG\_SALES

from currentinfo

Использование старых имен (*pub\_id, avg(ytd\_sales)*) в таком случае не поможет.

Вторая ситуация, в которой присвоение столбцам новых имен является обязательным, обычно возникает, если объединяемые в операторе **SELECT** столбцы имеют одинаковые имена. Даже если в операторе **SELECT** для них указаны разные имена таблиц, для разрешения возникающей неоднозначности придется переименовать их:

create view cities (Author, Authorcity, Pub, Pubcity)

as

select au\_lname, authors.city, pub\_name, publishers.city

from authors, publishers

where authors.city = publishers.city

# Понятие триггера

С любым событием, вызывающим изменение содержимого таблицы, пользователь может связать сопутствующее действие (**триггер**), которое СУБД должна выполнять при каждом возникновении события. Тремя такими событиями, запускающими триггеры, являются попытки изменить содержимое таблицы инструкциями INSERT, DELETE и UPDATE. Действие, вызываемое событием, задается как последовательность инструкций SQL.

Чтобы понять, как работает триггер, рассмотрим конкретный пример. Когда в таблицу ORDERS добавляется новый заказ, в БД необходимо внести еще два изменения:

* значение в столбце SALES для служащего, принявшего заказ, должно быть увеличено на стоимость заказа;
* имеющееся количество заказываемого товара в столбце QTY\_ON\_HAND должно быть уменьшено на заказанное количество единиц товара.

Следующая инструкция Transact-SQL определяет триггер SQL Server по имени NEWORDER, который вызывает автоматическое выполнение описанных выше изменений.

CREATE TRIGGER neworder

ON orders

FOR INSERT

AS UPDATE salesreps

SET sales = sales + inserted.amount

FROM salesreps, inserted

WHERE salesreps.empl\_num = inserted.rep

UPDATE products

SET qty\_on\_hand = qty\_on\_hand - inserted.qty

FROM products, inserted

WHERE products.mfr\_id = inserted.mfr

AND products.product\_id = inserted.product;

В первой части определения триггера указывается, что он вызывается всякий раз, когда к таблице orders обращается инструкция INSERT. В оставшейся части определения (после ключевого слова AS) описывается действие, выполняемое триггером. Как видно из этого примера, в SQL с целью поддержки триггеров язык SQL был существенно расширен. К другим расширениям, не показанным здесь, относятся проверки **IF/THEN/ELSE**, циклы, вызовы процедур и даже инструкции **PRINT**, выводящие пользовательские сообщения.

Существуют значительные отличия в плане поддержки триггеров в различных СУБД. Некоторые из этих отличий чисто синтаксические, другие отражают особенности реализации триггеров непосредственно в самой СУБД.

Сказанное можно достаточно наглядно продемонстрировать на примере СУБД DB2. Вот определение описанного выше триггера neworder, записанное в синтаксисе DB2.

CREATE TRIGGER neworder

AFTER INSERT ON orders

REFERENCING NEW AS new\_ord

FOR EACH ROW MODE DB2SQL

BEGIN ATOMIC

UPDATE salesreps

SET sales = sales + new\_ord.amount

WHERE salesreps.empl\_num = new\_ord.rep;

UPDATE products

SET qty\_on\_hand - qty\_on\_hand - new\_ord.qty

WHERE products.mfr\_id = new\_ord.mfr

AND products.product\_id = new\_ord.product;

END

Первая часть определения триггера включает те же элементы, которые используются и в SQL Server, только в несколько иной последовательности. Вторая строка явно указывает на то, что триггер вызывается *после* (AFTER) добавления заказа в БД. DB2 позволяет также выполнять триггер непосредственно *перед* выполнением связанной с ним операции. В данном случае это смысла не имеет, поскольку триггер связан с инструкцией INSERT, но в случае инструкций UPDATE и DELETE это вполне может потребоваться.

Предложение REFERENCING в DB2 задает *псевдоним таблицы* (new\_ord), который будет использоваться для ссылки на добавляемую строку в оставшейся части определения триггера. Функционально этот псевдоним аналогичен ключевому слову inserted в SQL Server. Ключевое слово NEW указано потому что строка добавляется. В случае инструкции DELETE потребовалось бы указать слово OLD. Для инструкции UPDATE DB2 позволяет обращаться к обоим значениям – NEW (ссылается на таблицу после обновления) и OLD (ссылается на таблицу перед обновлением).

Предложения BEGIN ATOMIC и END по сути служат скобками вокруг SQL-инструкций, определяющих действие триггера.

## Триггеры и ссылочная целостность

Триггеры обеспечивают альтернативный способ реализации ограничений ссылочной целостности с использованием первичных и внешних ключей. Например, вот как выглядит триггер SQL Server, который обеспечивает ссылочную целостность отношения таблиц offices/salesreps и выводит сообщение при неудачной попытке обновления.

CREATE TRIGGER Rep\_Update

ON salesreps

FOR INSERT, UPDATE

AS IF ((SELECT COUNT(\*)

FROM offices, inserted

WHERE offices.office = inserted.rep\_office) = 0))

BEGIN

PRINT "Указан неверный идентификатор офиса."

ROLLBACK TRANSACTION

END;

Триггеры можно использовать и для обеспечения расширенных форм ссылочной целостности. Следующий триггер в SQL Server распространяет любое обновление столбца office в таблице offices на столбец rep\_office таблицы salesreps.

CREATE TRIGGER Change\_Rep\_Office

ON offices

FOR UPDATE

AS IF UPDATE(office)

BEGIN

UPDATE salesreps

SET salesreps.rep\_office = inserted.office

FROM salesreps, inserted, deleted

WHERE salesreps.rep\_office = deleted.office

END;

Для корректного выполнения поиска и модификации данных определение триггера должно быть способно различать значения до и после обновления.

# Хранимые процедуры

## Концепции хранимых процедур

Возможности хранимых процедур:

1. **Условное выполнение.** Структура IF…THEN…ELSE позволяет SQL-процедуре проверить условие и, в зависимости от результата, выполнить различные действия.
2. **Циклы**. Цикл WHILE или FOR либо иная подобная структура позволяет многократно выполнять последовательность инструкций SQL до тех пор, пока не выполнится заданное условие окончания цикла.
3. **Блочная структура**. Последовательность инструкций SQL может быть сгруппирована в единый блок и использована в других управляющих конструкциях как единая инструкция.
4. **Именованные переменные**. SQL-процедура может сохранить вычисленное, извлеченное из БД или полученное любым другим способом значение в переменной, а когда оно понадобится снова – извлечь его из этой переменной.
5. **Именованные процедуры**. Последовательность инструкций SQL объединить в группу, дать ей имя и назначить формальные входные и выходные параметры так, что получится обычная подпрограмма или функция, которые используются в традиционных языках программирования. Определенную однажды таким образом процедуру можно вызывать по имени, передавая ей нужные значения в качестве входных параметров. Если она является функцией, возвращающей значение, то ее можно использовать в выражениях.

Набор элементов, реализующих все эти возможности, составляет язык хранимых процедур (SPL – Stored Procedure Language).

Например, без использования хранимой процедуры необходимо добавить клиента XYZ Corporation с идентификатором 2137, лимитом кредита 30000 и планируемым на первый год объемом продаж 50000; этот клиент будет работать со служащим Полом Крузом, идентификатор которого 103, из чикагского офиса.

INSERT INTO customers(cust\_num, company, cust\_rep, credit\_limit)

VALUES(2317, 'XYZ Corporation', 103, 30000.00);

UPDATE salesreps

SET quota = quota + 50000.00

WHERE empl\_num = 103;

UPDATE officers

SET target = target + 50000.00

WHERE city = 'Chicago';

COMMIT;

Вся эта работа может выполняться одной заранее созданной хранимой процедурой. Эта процедура создается на диалекте PL/SQL СУБД Oracle. Она называется Add\_Cust и принимает шесть параметров: идентификатор и имя клиента, лимит кредита и предполагаемый объем продаж, идентификатор закрепленного за клиентом служащего и название города, в котором расположен назначенный клиенту офис.

/\* Процедура для добавления данных о новом клиенте \*/

CREATE PROCEDURE Add\_Cust(

c\_name IN varchar2, /\* Имя клиента \*/

c\_num IN integer, /\* Идентификатор клиента \*/

credit\_lim IN number, /\* Лимит кредита \*/

tgt\_sls IN number, /\* Объем продаж \*/

c\_rep IN integer, /\* Идентификатор служащего \*/

c\_offc IN varchar2) /\* Город расположения офиса \*/

AS

BEGIN

/\* Добавляем новую строку в таблицу customers \*/

INSERT INTO customers(cust\_num, company, cust\_rep, credit\_limit)

VALUES (c\_num, c\_name, c\_rep, cred\_lim);

/\* Обновляем запись в таблице salesreps \*/

UPDATE salereps

SET quota = quota + tgt\_sls

WHERE empl\_num = c\_rep;

/\* Обновляем запись в таблице offices \*/

UPDATE offices

SET target = target + tgt\_sls

WHERE city = c\_offc;

/\* Заверешение транзакции \*/

COMMIT;

END;

Поскольку процедура хранится прямо в БД, ее можно вызвать следующей простой инструкцией, в которой указаны значения всех шести параметров.

Add\_Cust('XYZ Corporation', 2317, 30000.00, 50000.00, 103, 'Chicago');

## Использование хранимых процедур

Практически во всех диалектах определение хранимой процедуры создается с помощью инструкции **CREATE PROCEDURE**. Дополняющая ее инструкция **DROP PROCEDURE** предназначена для удаления из БД процедуры, которая больше не нужна. Инструкция CREATE PROCEDURE задает следующие элементы:

* имя хранимой процедуры;
* количество и типы данных ее параметров;
* имена и типы данных всех локальных переменных, используемых процедурой;
* последовательность инструкций, которые выполняются при вызове процедуры.

### Создание хранимой процедуры

Для создания хранимой процедуры используется инструкция **CREATE PROCEDURE**. Она назначает новой процедуре имя, по которому в дальнейшем ее можно будет вызывать. Хранимая процедура может принимать нуль или более параметров (в нашем примере их шесть: c\_name, c\_num, cred\_lim, tgt\_sls, c\_rep и c\_offc). Все параметры данной процедуры являются *входными*, о чем говорят ключевые слова IN.

В дополнение к входным параметрам некоторые диалекты SPL поддерживают *выходные* параметры, с помощью которых хранимые процедуры могут возвращать значения, вычисленные в ходе выполнения процедуры. При интерактивном вызове хранимых процедур от выходных параметров мало пользы, а вот если одна хранимая процедура вызывает другую, выходные параметры позволяют им эффективно обмениваться информацией. Некоторые диалекты SPL поддерживают параметры, которые одновременно являются и входными, и выходными, то есть их значения передаются хранимой процедуре, та их меняет и результирующие значения возвращаются вызывающей процедуре.

Рассмотрим предыдущий пример процедуры Add\_Cust, написанный на диалекте Sybase Transact-SQL. В отличие от Oracle, в данном диалекте ключевое слово PROCEDURE может быть сокращено до PROC. При этом весь список параметров процедуры не заключен в скобки, а все имена параметров начинаются с символа @.

/\* Процедура для добавления данных о новом клиенте \*/

CREATE PROC Add\_Cust

@c\_name varchar(20),

@c\_num int,

@cred\_lim decimal(9,2),

@tgt\_sls decimal(9,2),

@c\_rep int,

@c\_offc varchar(15)

AS

BEGIN

/\* Добавляем новую строку в таблицу customers \*/

INSERT INTO customers(cust\_num, company, cust\_rep, credit\_limit)

VALUES(@c\_num, @c\_name, @c\_rep, @cred\_lim)

/\* Обновляем запись в таблице salesreps \*/

UPDATE salesreps

SET quota = quota + @tgt\_sls

WHERE empl\_num = @c\_rep

/\* Обновляем запись в таблице offices \*/

UPDATE offices

SET target = target + @tgt\_sls

WHERE city = @c\_offc

/\* Завершение транзакции \*/

COMMIT TRANSACTION

END

Для удаления процедуры нужно ввести следующую команду:

DROP PROC Add\_Cust

### Вызов хранимой процедуры

Хранимую процедуру можно вызывать по-разному: из приложения с помощью соответствующей инструкции SQL, из другой хранимой процедуры, а также в интерактивном режиме. Синтаксис вызова хранимых процедур зависит от используемого диалекта.

На диалекте Transact-SQL вызов процедуры имеет такой вид:

EXECUTE Add\_Cust 'XYZ Corporation', 2317, 30000.00, 50000.00, 103, 'Chicago';

Ключевое слово **EXECUTE** можно сократить до EXEC. Кроме того, задавая параметры, можно явно указать их имена, что позволяет перечислять параметры в любом порядке:

EXECUTE Add\_Cust

@c\_name = 'XYZ Corporation',

@c\_num = 2317,

@cred\_lim = 30000.00,

@tgt\_sls = 50000.00,

@c\_rep = 103,

@c\_offc = 'Chicago';

### Переменные хранимых процедур

В хранимых процедурах удобно (а иногда и просто необходимо) объявлять переменные для хранения некоторых промежуточных значений. Обычно переменные объявляются в начале тела процедуры, сразу за заголовком и перед последовательностью составляющих ее инструкций SQL. Для переменных можно использовать все те же типы данных, что и для столбцов таблиц.

Нужно обратить внимание, что диалект Transact-SQL требует, чтобы все имена переменных, как и имена параметров, начинались с символа @. Инструкция **DECLARE** объявляет локальные переменные процедуры. В примере имеются две переменные типа money и varchar.

/\* Процедура проверки общей стоимости заказов клиента \*/

CREATE PROC Chk\_Tot

@c\_num int // Один входной параметр

AS

/\* Объявляем две локальные переменные \*/

DECLARE @tot\_ord money, @msg\_text varchar(30)

BEGIN

/\* Вычисляем общую стоимость заказов клиента\*/

SELECT @tot\_ord = sum(amount)

FROM orders

WHERE cust = @c\_num

/\* В зависимости от величины суммы заносим в переменную соответствующее сообщение \*/

IF tot\_ord < 30000.00

SELECT @msg\_text = "Большой объем заказов"

ELSE

SELECT @msg\_text = "Малый объем заказов"

/\* Выполняем дальнейшую обработку сообщения\*/

...

END

Инструкция SELECT выполняет еще одну функцию – присваивает значения переменным.

### Блоки инструкций

Практически во всех хранимых процедурах, кроме самых простых, возникает необходимость объединения некоторой последовательности инструкций в группу, интерпретируемую как одна инструкция. В Transact-SQL структура блока инструкций очень проста.

/\* Блок инструкций Transact-SQL \*/

BEGIN

/\* Здесь должна располагаться последовательность инструкций SQL \*/

...

END

Единственной задачей ключевых слов BEGIN…END является ограничение блока инструкций. Они не влияют на область видимости локальных переменных или других объектов БД. Процедуры, циклы и другие конструкции Transact-SQL оперируют единственной инструкцией, и поэтому в них очень часто используются блоки, объединяющие несколько инструкций в одну.

Пример хранимой процедуры

CREATE PROC Newperson

@identif varchar(10),

@firstname varchar(20),

@lastname varchar(20),

@nick varchar(15),

@address varchar(30),

@telephone char(13),

@emailaddress varchar(30),

@passportnum char(8),

@idcodenum char(10),

@cityname varchar(20)

AS

BEGIN

INSERT INTO personnel(id, fname, lname, nickname, address, telephone, email, passport, idcode, city)

VALUES(@identif, @firstname, @lastname, @nick, @address, @telephone, @emailaddress, @passportnum, @idcodenum, @cityname)

PRINT 'Добавление завершено'

END;

Вызываем хранимую процедуру

EXEC Newperson '11', 'Elena', 'Petrova', 'lenka', 'Zhukovskaya, 16/7', '+380665652832', 'lenka55@mail.ru', 'MP233674', '4120155680', 'Kyiv';

### Функции

В дополнение к хранимым процедурам многие диалекты SPL поддерживают *хранимые функции*. От процедуры функция отличается тем, что при каждом вызове возвращает лишь одно значение (некоторые данные, объект, XML-документ), в то время как хранимая процедура может либо вернуть много разных значений, либо ни одного.

Пример функции для Oracle PL/SQL.

/\* Возвращает общую стоимость заказов клиента \*/

CREATE FUNCTION Get\_Tot\_Ords(c\_num IN number)

RETURN number

AS

/\* Объявляем локальную переменную для хранения итога \*/

DECLARE tot\_ord number(16, 2);

BEGIN

/\* Простой запрос, возвращающий итоговую сумму \*/

SELECT sum(amount) INTO tot\_ord

FROM orders

WHERE cust = c\_num;

/\* Возвращаем сумму в качестве значения функции \*/

RETURN tot\_ord;

END;

Многие диалекты SPL допускают использование хранимых функций в SQL-выражениях. В частности, это верно для диалекта Oracle PL/SQL, поэтому вышеуказанную функцию можно, например, использовать в условии отбора.

SELECT company, name

FROM customers, salespreps

WHERE cust\_rep = empl\_num

AND Get\_Tot\_Ords(cust\_num) > 10000.00;

Когда СУБД вычисляет условие отбора для каждой строки в таблице результатов запроса, она использует идентификатор клиента из этой строки в качестве аргумента функции Get\_Tot\_Ords и проверяет, не превысило ли значение, возвращаемое этой функцией, 10000.

Пример функции в Transact-SQL

CREATE FUNCTION PureBalance(@identifer decimal)

RETURNS decimal(16,2)

BEGIN

DECLARE @purebal decimal(16,2)

SELECT @purebal = balance - creditdebt

FROM pers\_bankname

WHERE bankpers\_id = @identifer

RETURN @purebal

END;

Вызываем функцию:

select dbo.PureBalance(5)

### Возврат значений через параметры

Некоторые диалекты SPL позволяют возвращать из процедуры более одного значения с помощью *выходных параметров*. Выходные параметры перечисляются в списке параметров процедуры так же, как и рассмотренные в предыдущих примерах входные параметры. Однако вместо того чтобы передавать данные в процедуру, они возвращают данные из процедуры.

Пример возврата информации о клиенте

/\* Возвращает информацию о клиенте по его номеру \*/

CREATE PROC Get\_Cust\_Info

@c\_num int,

@c\_name varchar(20) out,

@r\_name varchar(15) out,

@c\_offc varchar(15) out

AS

BEGIN

/\* Простой однострочный запрос, возвращающий интересующую нас информацию \*/

SELECT

@c\_name = company,

@r\_name = name,

@c\_offc = city

FROM customers, salesreps, offices

WHERE cust\_num = @c\_num

AND empl\_num = cust\_rep

AND office = rep\_office

END;

Когда эта процедура вызывается из другой процедуры Transact-SQL, второй, третий и четвертый параметры должны быть объявлены в *вызывающей* процедуре как выходные, как и в объявлении вызываемой процедуры.

/\* Получаем информацию о клиенте 2111 \*/

DECLARE the\_name varchar(20)

DECLARE the\_rep varchar(15)

DECLARE the\_city varchar(15)

EXEC Get\_Cuts\_Info @c\_num = 2111,

@c\_name = the\_name output,

@r\_name = the\_rep output,

@c\_offc = the\_city output;

Пример хранимой процедуры с исходящими параметрами:

CREATE PROC Return\_clinfo

@identif char(15),

@fname varchar(20) output,

@lname varchar(20) output,

@telephone char(13) output,

@email varchar(30) output

AS

BEGIN

SELECT

@fname = fname,

@lname = lname,

@telephone = telephone,

@email = email

FROM personnel

WHERE id = @identif

END;

DECLARE @thefname varchar(20)

DECLARE @thelname varchar(20)

DECLARE @thetelephone char(13)

DECLARE @theemail varchar(30)

EXEC Return\_clinfo 1,

@fname = @thefname output,

@lname = @thelname output,

@telephone = @thetelephone output,

@email = @theemail output;

PRINT @thefname + ' ' + @thelname + ' ' + @thetelephone + ' ' + @theemail;

### Условное выполнение

Одним из базовых элементов хранимых процедур является конструкция IF…THEN…ELSE, используемая для организации ветвлений внутри процедуры. Рассмотрим пример.

/\* Процедура для добавления данных о новом клиенте \*/

CREATE PROC Add\_Cust

@c\_name varchar(20), // Имя клиента

@c\_num int, // Идентификатор клиента

cred\_lim double, // Лимит кредита

tgt\_sls double, // Объем продаж

c\_rep int, // Идентификатор служащего

c\_offc varchar(20) // Город расположения офиса

AS

BEGIN

/\* Добавляем новую строку в таблицу CUSTOMERS \*/

INSERT INTO customers(cust\_num, company, cust\_rep, credit\_limit)

VALUES(c\_num, c\_name, c\_rep, cred\_lim)

/\* Обновляем запись в таблице SALESREPS \*/

IF tgt\_sls <= 20000.00

THEN

UPDATE salesreps

SET quota = quota + tgt\_sls

WHERE empl\_num = c\_rep

ELSE

UPDATE salesreps

SET quota = quota + 20000.00

WHERE empl\_num = c\_rep

END;

/\* Обновляем запись в таблице OFFICES \*/

UPDATE offices

SET target = target + tgt\_sls

WHERE city = c\_offc

END;

### Циклы

Предположим, что мы хотим циклически выполнять некоторую группу инструкций, изменяя значение переменной-счетчика с именем item\_num в пределах от 1 до 10. Вот как это делается в Oracle PL/SQL.

/\* Обрабатываем каждый из 10 элементов \*/

FOR item\_num IN 1. .10 loop

/\* Обрабатываем текущий элемент \*/

...

/\* Проверяем, не следует ли завершить цикл раньше \*/

EXIT WHEN (item\_num = special\_item);

END loop;

Последовательность инструкций, составляющих тело цикла, должна выполниться 10 раз, и при каждом проходе цикла значение переменной item\_num увеличивается на 1. Предложение **EXIT** обеспечивает возможность выхода из цикла раньше, причем это можно сделать как по заданному условию, так и безо всякого условия.

Часто условие окончания цикла встраивается прямо в структуру цикла. Рассмотрим пример на Transact-SQL.

/\* Уменьшаем плановые объемы продаж до тех пор, пока их сумма не станет меньше 2400000 \*/

WHILE(SELECT SUM(target) FROM offices) < 2400000.00

BEGIN

UPDATE offices

SET target = target - 10000.00

END;

Блок BEGIN…END в этом цикле не обязателен, поскольку его тело и так состоит из единственной инструкции, но большинство циклов WHILE в Transact-SQL его содержат. Если тело цикла содержит более одной инструкции, блок BEGIN…END обязателен.

### Другие управляющие конструкции

Некоторые диалекты SPL включают дополнительные управляющие конструкции.

EXIT FOR;

EXIT WHILE;

EXIT FOREACH;

CONTINUE FOR;

CONTINUE WHILE;

CONTINUE FOREACH;

В Transact-SQL единственная инструкция **BREAK** заменяет все три варианта инструкции EXIT; инструкция **CONTINUE** в этом диалекте тоже только одна.

Еще один способ изменения хода выполнения хранимых процедур – это безусловный переход к метке, выполняемой инструкцией **GOTO**. В большинстве диалектов метка представляет собой идентификатор, за которым следует двоеточие. Как правило, выход по метке за пределы цикла не допускается, как не допускается и переход внутрь цикла или условной конструкции.

### Циклы с курсорами

**Курсор** – это объект БД, который позволяет приложениям работать с записями "по одной", а не сразу с множеством, как это делается в обычных SQL.

Операторы управления явным курсором:

* Оператор **DECLARE** выполняет объявление явного курсора.
* Оператор **OPEN** открывает курсор, создавая новый результирующий набор на базе указанного запроса.
* Оператор **FETCH** выполняет последовательное извлечение строк из результирующего набора от начала до конца.
* Оператор **CLOSE** закрывает курсор и освобождает занимаемые им ресурсы.

В Transact-SQL нет специального цикла FOR для обработки результатов запроса. Однако можно использовать инструкции DECLARE CURSOR, OPEN, FETCH и CLOSE, аналогичные тем, которые применяются для работы с курсорами во встроенном SQL.

На следующем примере управление циклом обеспечивает системная переменная @@SQLSTATUS, в Transact-SQL являющаяся аналогом переменной SQLSTATE встроенного SQL. Эта переменная получает значение 0, если инструкция FETCH успешно извлекла следующую запись, и ненулевое значение, когда все записи получены.

CREATE PROC Sort\_Orders

AS

/\* Локальные переменные для хранения результатов \*/

DECLARE @ord\_amt decimal (16,2); // Стоимость заказа

DECLARE @c\_name varchar(20); // Имя клиента

DECLARE @r\_name varchar(15); // Имя служащего

/\* Объявляем курсор для запроса \*/

DECLARE o\_curs CURSOR FOR

SELECT amount, company, name

FROM orders, customers, salesreps

WHERE cust = cust\_num

AND rep = empl\_num

BEGIN

/\* Открываем курсор и извлекаем первую запись \*/

OPEN o\_curs

FETCH o\_curs INTO @ord\_amt, @c\_name, @r\_name

/\* Если строк больше нет, выходим из процедуры \*/

IF(@@sqlstatus = 2)

begin

CLOSE o\_curs

RETURN

END

/\* Цикл обработки всех строк таблицы результатов \*/

WHILE(@@sqlstatus = 0)

BEGIN

/\* Обработка малых заказов \*/

IF(@ord\_amt < 1000.00)

THEN INSERT INTO smallorders

VALUES(@r\_name, @ord\_amt)

/\* Обработка больших заказов \*/

ELSE IF(@ord\_amt > 1000.00)

THEN INSERT INTO bigorders

VALUES(@c\_name, @ord\_amt)

END

/\* Все записи обработаны; закрываем курсор \*/

CLOSE o\_curs

END;

### Обработка ошибок

Если операции над данными выполняются хранимой процедурой, она сама должна обрабатывать ошибки.

В Transact-SQL информацию о происшедших ошибках можно получить из специальных системных переменных. Имеется огромное (более 100) количество глобальных системных переменных, хранящих информацию о состоянии сервера и транзакции, открытых подключениях и т.п. Однако для обработки ошибок чаще всего используются только две из них:

* @@ERROR – код ошибки, произошедшей при выполнении последней инструкции SQL.
* @@SQLSTATUS – состояние последней операции FETCH.

Признаком нормального завершения операции в обеих переменных является значение 0. Другие значения указывают на ошибки или нестандартные ситуации. В хранимых процедурах Transact-SQL глобальные переменные используются точно так же, как и локальные. В частности, их можно применять для организации циклов и ветвления.

## Преимущества хранимых процедур

Выделяют следующие преимущества хранимых процедур:

1. Производительность.
2. Повторное использование.
3. Сокращение сетевого трафика.
4. Безопасность.
5. Инкапсуляция.
6. Простота доступа.
7. Обеспечение бизнес-правил.

## Системные хранимые процедуры

Сегодня сотни хранимых процедур Transact-SQL облегчают выполнение множества полезных функций, таких как, например, управление записями пользователей, заданиями, распределенными серверами, репликацией и т.д. Большинство системных процедур Transact-SQL названо в соответствии со следующими соглашениями:

* sp\_add\_*имя* – добавление нового объекта (пользователя, сервера, реплики и т.п.);
* sp\_drop\_*имя* – удаление существующего объекта;
* sp\_help\_имя – получение информации об объекте или объектах.

Например, процедура sp\_helpuser возвращает информацию о пользователях текущей БД.

## Внешние хранимые процедуры

Для преодоления ограниченных возможностей можно обращаться к внешним хранимым процедурам. Внешняя хранимая процедура – это процедура, написанная на одном из традиционных языков программирования и скомпилированная вне СУБД. Для ее использования нужно предоставить СУБД объявление процедуры – ее имя, параметры и другую информацию, необходимую для ее вызова, наподобие соглашения о передаче параметров.

Microsoft SQL Server предоставляет программистам набор системных внешних процедур, обеспечивающих доступ к функциям операционной системы. Например, процедура xp\_sendmail позволяет отправлять пользователям электронные почтовые сообщения с информацией о событиях, происходящих в базе данных.

xp\_sendmail @RECIPIENTS = 'Joe', 'Sam',

@MESSAGE = 'Отчет готов';

Другая процедура, xp\_cmdshell, выполняет команды ОС, в которой работает SQL Server. Допускается также создание пользовательских внешних процедур, которые хранятся в динамически компонуемых библиотеках (DLL) и вызываются из хранимых процедур SQL Server.

## Триггеры

### Преимущества и недостатки триггеров

Триггеры могут выполнять множество функций, включая следующие:

1. Контроль изменений. Триггер может отслеживать и отменять определенные изменения, не разрешаемые в конкретной БД.
2. Каскадные операции. Триггер может обнаруживать определенные операции и автоматически вносить соответствующие изменения в другие таблицы БД.
3. Поддержка целостности. Триггер может поддерживать более сложные связи между данными, чем те, которые могут быть выражены простыми ограничениями на значения столбцов и условиями ссылочной целостности.
4. Вызов хранимых процедур. В ответ на обновление БД триггер может вызвать одну или несколько хранимых процедур и даже выполнить какие-то действия вне БД, используя внешние процедуры.
5. Обнаружение системных событий. В СУБД с поддержкой триггеров для системных событий триггер может отслеживать такие события, как, например, подключение к БД определенного пользователя.

### Триггеры в диалекте Transact-SQL

В Transact-SQL триггеры создаются инструкцией CREATE TRIGGER. Рассмотрим пример:

CREATE TRIGGER upd\_tgt

/\* Триггер, активизирующийся при добавлении записи в таблицу salesreps \*/

ON salesreps

FOR INSERT

AS

IF(@@rowcount = 1)

BEGIN

UPDATE offices

SET target = target + inserted.quota

FROM offices, inserted

WHERE offices.office = inserted.rep\_office

END

ELSE

raiseerror 23456;

Первое предложение определяет имя триггера. Второе (является обязательным) указывает, с какой таблицей он связан. Третье предложение (также обязательно) определяет операцию, в ответ на которую должен выполняться данный триггер. Кроме инструкции INSERT, можно связывать триггеры с инструкциями UPDATE и DELETE, причем триггер может быть связан с одной, любыми двумя или всеми тремя инструкциями сразу – они перечисляются через запятую. Однако для любой из этих трех операций с таблицей может быть связан только один триггер. Тело триггера начинается с ключевого слова AS. Чтобы понять, что делает наш триггер, необходимо разобраться, как Transact-SQL обрабатывает строки целевой таблицы при ее модификации.

Специально для триггеров Transact-SQL определяет две виртуальные таблицы, структура которых идентична структуре таблицы, с которой связан триггер. Эти таблицы называются DELETED и INSERTED. Они заполняются строками из модифицируемой таблицы, причем их конкретное содержимое зависит от выполняемой операции.

* DELETE – все строки, удаленные из связанной таблицы, помещаются в таблицу DELETED; таблица INSERTED пуста.
* INSERT – все строки, добавленные в связанную таблицу, помещаются в таблицу INSERTED; таблица DELETED пуста.
* UPDATE – для каждой строки целевой таблицы, измененной инструкцией UPDATE, ее *исходная* версия помещается в таблицу DELETED, а *новая* версия – в таблицу INSERTED.

К этим двум виртуальным таблицам можно обращаться из тела триггера, и их данные можно использовать в триггере наряду с данными всех остальных таблиц. В нашем примере триггер проверяет, была ли в таблицу добавлена одна запись, для чего он обращается к системной переменной @@rowcount. Если это так, значение столбца quota из виртуальной таблицы inserted прибавляется к значению столбца target соответствующей строки таблицы offices. Чтобы найти в таблице offices соответствующую запись, мы соединяем эту таблицу с виртуальной таблицей inserted по идентификаторам офисов.

А вот еще один триггер, который отслеживает попытки удаления из БД записей о клиентах, для которых имеются записи о заказах. Обнаружив такую ситуацию, триггер отменяет всю транзакцию, включая инструкцию DELETE, в ответ на которую был вызван триггер.

CREATE TRIGGER chk\_del\_cust

/\* Триггер, активизирующийся при удалении записи из таблицы CUSTOMERS \*/

ON customers

FOR DELETE

AS

/\* Выясняем, имеются ли заказы у удаляемого клиента \*/

IF(SELECT count(\*)

FROM orders, deleted

WHERE orders.cust = deleted.cust\_num) > 0

BEGIN

ROLLBACK TRANSACTION

PRINT "Удаление невозможно: имеются заказы"

raiseerror 31234

END;

Для триггеров, связанных с инструкцией UPDATE, в Transact-SQL предусмотрена возможность выяснить, какие именно столбцы таблицы были изменены, и в ответ выполнить соответствующие действия. Для этого в триггерах может использоваться специальная форма инструкции IF – IF UPDATE. Следующий триггер активизируется в ответ на обновление записей в таблице salesreps и выполняет различные действия в зависимости от того, какой из столбцов изменен: quota или sales.

CREATE TRIGGER upd\_reps

/\* Триггер, активизирующийся при обновлении таблицы salesreps \*/

ON salesreps

FOR INSERT, UPDATE

AS

IF UPDATE(quota)

/\* Обрабатываем обновление столбца QUOTA \*/

...

IF UPDATE(sales)

/\* Обрабатываем обновление столбца SALES \*/

...

### Обработка ошибок

В стандарте SQL/PSM описаны предопределенные состояния ошибок, которые могут быть обнаружены и обработаны.

* SQLWARNING – одно из предупреждений, определенных стандартом SQL.
* NOT\_FOUND – ситуация, когда инструкция FETCH достигла конца результирующего множества.
* SQLSTATE значение – проверка определенного кодаошибки SQLSTATE.
* Пользовательское состояние – именованное в хранимой процедуре состояние.

Состояния обычно определяются в терминах значений переменной SQLSTATE. Вместо использования числовых кодов, можно назначать состояниям имена. Кроме того, можно описывать собственные состояния, требующие специальной обработки.

DECLARE bad\_err condition FOR sqlstate '12345';

DECLARE my\_err condition;

После описания состояния его можно явно сгенерировать в хранимой процедуре с помощью инструкции **SIGNAL**.

SIGNAL bad\_err;

SIGNAL sqlstate '12345';

Для обработчики состояний ошибки, которые могут возникать в ходе выполнения процедуры, можно создавать *обработчики*. В определении обработчика указывается перечень состояний, которые он должен обрабатывать, и действия, которые он должен предпринимать. Существует три типа обработчиков:

* CONTINUE – когда обработчик завершит свою работу, управление будет передано инструкции, следующей за той, при возникновении которой возникло состояние ошибки; таким образом, выполнение процедуры будет *продолжено* со следующей строки.
* EXIT – когда обработчик завершит свою работу, управление будет передано в конец блока инструкций, при выполнении которого возникло состояние ошибки; таким образом, будет осуществлен *выход* из текущего блока.
* UNDO – когда обработчик завершит свою работу, все изменения БД, сделанные в блоке инструкций, при выполнении которого возникла исключительная ситуация, будут отменены; результат будет таким же, как при отмене транзакции, начатой в начале блока.

Вот несколько примеров определений обработчиков.

/\* Обработать предупреждение SQL и продолжить \*/

DECLARE CONTINUE handler FOR sqlwarning

CALL my\_warn\_routine();

/\* Обработать ошибку и отменить изменения \*/

DECLARE UNDO handler FOR user\_disaster

BEGIN

/\* Исправляем последствия ошибки \*/

...

END;

Текст обработчика может быть довольно сложным, и в ходе его выполнения могут возникать новые ошибки. Чтобы избежать бесконечной обработки ошибок, сигнализация о состоянии ошибки на время обработки автоматически отключается. Однако стандарт позволяет перекрыть такое поведение при помощи инструкции **RESIGNAL**. Она работает в точности так же, как и инструкция SIGNAL, но используется исключительно в подпрограммах обработки ошибок.