Сергей Моисеенко

SQL

Задачи и решения

- Логика составления запросов
- Разбор типичных ошибок
- Решения с комментариями

Собрал книгу: Александров А.А. (youtube-канал: andrei-tash, ссылка на скачивание книги: gracefinance.ru/sql-ex.ru.zip)

Интерактивный учебник по SQL

Если вы хотите узнать, что такое SQL - этот сайт для вас.

Если вы знаете, что такое SQL, но хотите научиться писать запросы на этом языке - этот сайт для вас.

Если вы думаете, что умеете писать запросы, не торопитесь покидать сайт. Возможно, вы откроете что-то новое для себя.

Если вы не знаете как и где писать запросы - это то, что вы так долго искали. Вы сможете выполнять адресованные к учебным базам запросы непосредственно на сайте.

Если вам покажется все понятным в настоящем учебнике, испытайте себя на тестах **SQL-EX.RU**. Обучающий этап здесь уже доступен для разных СУБД, включая MSSQL, MySQL, Oracle, PostgreSQL.

Мы надеемся, что сайт окажется полезным как новичкам, так и профессионалам в SQL.

Запросы в тексте учебника выполняются реальной СУБД. Пока это Microsoft SQL Server, но мы планируем** возможность использования также и других серверов баз данных, начиная со свободно распространяемых и заканчивая коммерческими продуктами:

- **MySQL**
- **PostgreSQL**
- **Oracle**

с тем, чтобы можно было изучать особенности диалектов языка SQL у разных СУБД.

Однако уже сейчас вы можете адресовать запросы не только к MS SQL Server, но и к MySQL и PostgreSQL, используя консоль.

sql-ex team

* Требуется регистрация/авторизация

** Свои замечания и пожелания вы можете высказать, используя почтовую форму.

Содержание:

Введение

- (1.1) Чему посвящен этот учебник?
- (1.2) Что необходимо для работы с учебником?
- (1.3) Как работать с учебником?
- (1.4) Используемая терминология и особенности реализации
- (1.5) Благодарности

(Часть I) Характерные ошибки при решении задач на написание запросов на выборку (SELECT)

(Глава 1) База данных «Компьютерная фирма»

(Глава 2) База данных «Фирма вторсырья»

(Глава 3) База данных «Корабли»

(Глава 4) Подсказки и решения

(Часть II) Язык манипуляции данными в SQL

(Глава 5) Оператор SELECT

(Глава 6) Операторы модификации данных

(Часть III) Готовимся ко второму этапу тестирования

(Глава 7) Функции Transact-SQL для работы со строками и данными типа даты/времени

Числовые функции в SQL Server

(Глава 8) Типичные проблемы

(Часть IV) Новое в стандарте и реализациях языка SQL

- Оператор MERGE

- (Глава 9) Функции ранжирования
- (Глава 10) Операторы PIVOT и UNPIVOT
- (Глава 11) Общие табличные выражения (CTE)

- Оконные функции
- CROSS APPLY / OUTER APPLY
- Функция CONCAT
- Функция EOMONTH
- Функция STRING_AGG
- Функция STRING_SPLIT
- Функция CHOOSE

(Часть V) Заметки о типах данных

- CHAR и VARCHAR
- Float(n)
- Целочисленное деление
- Методы типа данных XML
- Язык определения данных (SQL DDL)
- Создание базовых таблиц
- Категорная целостность или целостность сущностей
- Проверочные ограничения
- Оператор ALTER TABLE
- Значения по умолчанию
- Ссылочная целостность: внешний ключ (FOREIGN KEY)
- Вложенные запросы в проверочных ограничениях
- Проверочное ограничение уровня таблицы
- INFORMATION_SCHEMA и Oracle
- (Глава 14) База данных «Аэрофлот»
- (Глава 15) База данных «Окраска»
- Ошибки в задачах DML

Вопросы оптимизации

- Основные операции плана выполнения SQL Server
- Описание операций плана выполнения в Postgresql
- Описание операций плана выполнения в Oracle
- MySQL. Использование переменных в запросе

- Графовые базы данных
- Графовые базы данных SQL Server
- Запросы к графовой базе данных

Приложения

- Приложение 1. Описание учебных баз данных
- Приложение 2. Список задач
- Приложение 3. Хроники Торуса
- Заключение
- Список цитируемых источников

Введение

Если вы хотите узнать, как получить информацию из базы данных, но не знаете, с чего начать, то эта книга для вас. Если же вы знакомы с языком SQL или даже являетесь специалистом по базам данных, то вам будет интересно оценить свои знания.

Цель этой книги — быстрое изучение языка SQL. При этом «быстрое» вовсе не означает «поверхностное». Напротив, оставляя в стороне многие аспекты языка, автор старается дать глубокое понимание логической структуры данных и, как следствие, правильного построения запросов с учетом этой структуры. Надеемся, что чтение этой книги облегчит понимание и того, о чем здесь не написано.

Чему посвящен этот учебник?

Книга посвящена практическому использованию языка SQL и, в первую очередь, извлечению информации из реляционной базы данных, то есть наиболее синтаксически сложному оператору SELECT. Однако здесь вы также найдете необходимую информацию по другим операторам подъязыка манипуляции данными (DML — Data Manipulation Language), а именно, операторам INSERT, UPDATE и DELETE, осуществляющим модификацию данных. В дальнейшем планируется добавить разделы, посвященные подъязыку определения схемы – DDL (Data Definition Language), посредством которого создаются и изменяются объекты базы данных, в частности, таблицы и представления.

Такая подача материала связана с тем, что учебник предназначен, в основном, для потенциальных пользователей и разработчиков приложений СУБД, которых, в первую очередь, интересуют вопросы

извлечения информации из существующих баз данных, и только потом — их модификации и создания структур хранения.

Все примеры учебника можно выполнять он-лайн на реальном сервере баз данных. Вы можете также редактировать существующие и создавать новые запросы при помощи встроенного редактора, и также выполнять их на сервере.

Ряд глав книги содержат упражнения, рекомендуемые для закрепления изучаемого материала. Вы можете воспользоваться системой проверки правильности решения этих упражнений на сайте **«Упражнения по SQL»**, откуда и заимствованы эти упражнения

Следуя испытанной практике, мы предлагаем вам поучиться на чужих ошибках. При этом, мы, разбирая ошибочные решения некоторых упражнений, в большинстве случаев не даем окончательных «правильных» решений. Этому есть две причины:

- задачу можно решить разными способами; поэтому, давая правильное решение, мы ограничивали бы творческую активность читателя;
- возможность самостоятельно решить задачу дает больший обучающий эффект, как показал опыт поддержки сайта SQL-EX.RU.

Справедливости ради, заметим, что после выяснения причины ошибки в результате анализа неверных решений их исправление не должно составлять большого труда.

Приведенные в книге ошибочные решения не являются надуманными. Эти запросы писали посетители сайта, которые после неудачных попыток решить задачу просили объяснить, почему верный, по их мнению, запрос не принимается системой проверки. Поэтому мы берем на себя смелость утверждать, что это объяснение причин таких «характерных» ошибок, позволит добиться значительно большего прогресса в изучении SQL, чем простое рассмотрение примеров использования тех или иных конструкций языка.

Что необходимо для работы с учебником?

Для решения рассматриваемых в учебнике задач вы можете:

1. Использовать имеющуюся консоль для выполнения запросов. Сейчас запросы выполняются на **Microsoft SQL Server**, но мы планируем расширить круг используемых в учебнике СУБД. В частности, мы предполагаем реализовать выполнение запросов на СУБД **Oracle**, **PostgreSQL**, **MySQL**.

2. Если вы хотите использовать имеющуюся у вас СУБД, вы можете скачать скрипты учебных баз данных, каждый из которых создаст необходимые таблицы в вашей базе данных и наполнит их данными.

3. Если вы уже знакомы с языком **SQL**, посетите www.sql-ex.ru, где в режиме он-лайн можно решать не только представленные здесь простые задачи для начинающих. Более сложные задачи позволят вам оценить свой уровень практического владения языком **SQL**, а рейтинговая система даст вам возможность посоревноваться с другими ее участниками, которых уже несколько сотен тысяч. Реализованная на сайте система проверки будет контролировать правильность ваших решений, а имеющийся форум позволит вам познакомиться с решениями других посетителей. Вы можете пройти хорошую школу, разобравшись в опубликованных на форуме решениях, так как многие из них писались истинными профессионалами.

Как работать с учебником?

Если вы только приступаете к изучению языка **SQL**, то лучше начать с Части II, где рассматривается синтаксис оператора **SELECT**, и каждая глава содержит номера упражнений, которые рекомендуется решить для закрепления соответствующего материала. Затем переходите к Части I, которая поможет преодолеть затруднения при решении упражнений и разобраться в нюансах. С операторами модификации данных можно познакомиться в Главе 6. Упражнения по этим операторам не рассматриваются в книге (хотя в будущем обязательно появятся), поскольку они достаточно просты, и сложные конструкции могут возникать здесь только за счет использования подзапросов. Однако достаточное число примеров в книге и решение задач на сайте SQL-EX.RU позволят вам в полной мере усвоить этот материал.

Если вы в какой-то степени уже знакомы с языком **SQL**, то начинайте с Части I, обращаясь к Части II лишь для получения справки по тем или иным предложениям оператора **SELECT** или чтобы узнать о специфических особенностях реализации (в настоящее время мы используем **MS SQL Server 2012**). Чтение остальных частей книги зависит от степени вашей подготовки. Надеюсь, что они также окажутся полезными для вас.

Я советую начать чтение книги с описания предметной области, моделируемой учебной базой данных; и внимательного изучения схемы данных. Затем стоит познакомиться с «неправильным» решением и самостоятельно найти в нем ошибку, исправить и выполнить, а затем прочитать объяснение. Убедиться в правильности своего понимания, вы можете на **SQL-EX.RU**.

Если вам все понятно и без объяснений, то эта книга вам не нужна. В таком случае заходите на SQL-EX.RU и попробуйте опередить лидеров рейтинга, выполнив упражнения повышенной сложности, которые не рассматриваются в этом учебнике.

На сайте имеется сервис оценки эффективности запросов. Там вы сможете не только научиться писать правильные запросы, но и решать задачу оптимально, то есть таким образом, чтобы запросы при их выполнении потребляли меньше ресурсов сервера.

Используемая терминология и особенности реализации

Будем считать синонимами термины таблица и отношение; строка и кортеж; столбец, атрибут и поле. Интересующихся тонкостями терминологии отсылаю к фундаментальной книге Дейта [1].

Примеры в учебнике, по мере возможности, выдержаны в рамках стандарта языка SQL. Однако там, где потребовались специфические средства (встроенные функции, например) при написании запросов использовался диалект языка **SQL Server** (на данный момент на сайтах sql-ex.ru и sql-tutorial.ru используется версия 2012 этой СУБД). Все приведенные в книге скрипты проверялись именно на этой СУБД. Справочную информацию по особенностям реализации вы найдете в Части III этой электронной книги.

Поддержка других СУБД постоянно расширяется как на сайте учебника, так и на sql-ex.ru. Достаточно сказать, что для решения задач обучающего этапа можно выбрать не только SQL Server, но также и MySQL, PostgreSQL, Oracle. Там, где решение типовых задач в рамках стандарта по тем или иным причинам невозможно, приводятся примеры запросов в диалектах разных СУБД.

Благодарности

В первую очередь я хотел бы поблагодарить своих коллег по созданию и сопровождению сайта SQL-EX.RU — Майстренко А.В., Лысенко О.В., Калинкина В.Ю., Валуева Д.И., Долгополова В., Курочкина П.А., Красовского Е.А., Коптельцева В.В. без помощи которых не было бы ни сайта, ни книги. Автор выражает особую признательность Гершовичу В.И., который первым предложил написать данную книгу, а также своим соавторам - Бежаеву А.Ю., Чебыкину Д.Н., Красовскому Е.А. - написавшим отдельные материалы для этой книги. Я благодарен всем авторам задач, лицам, внесшим свой вклад в развитие ресурса, а также многочисленным посетителям, ошибки которых здесь и анализируются. Я не стану здесь перечислять всех поименно, т.к. их имена всегда можно увидеть на сайте www.sql-ex.ru.

Две схемы данных и многие формулировки задач по этим схемам заимствованы из книги Дж.Ульмана и Дж.Уидома [2], которым автор выражает свою признательность.

Характерные ошибки при решении задач на написание запросов на выборку (SELECT)

Засим следовали несколько примерных диалогов между учителем и учеником, совершенно невразумительных для рационального уха и

любой дуалистической или двузначной логики; ответы учителей на вопросы учеников заключались, как правило, в том, чтобы дать палкой по башке, вылить на голову кувшин холодной воды, пинком вышвырнуть за дверь или, в лучшем случае, глядя им в лицо, повторить их же вопрос.

Х. Кортасар. Игра в классики

В этой части анализируются ошибочные решения задач обучающего этапа и первого рейтингового этапа тестирования на сайте **«Упражнения по SQL»**. Мы стараемся не давать «правильных» решений за исключением наиболее простых задач, чтобы облегчить вводный этап обучения для начинающих. Тем не менее, многие представленные решения являются *почти* правильными, и их исправление не составит труда, если вникнуть в причину ошибки. Некоторые подсказки вынесены в отдельную главу **«Подсказки и решения»**, где даются дополнительные пояснения, а также рассматриваются некоторые альтернативные подходы к решению задачи.

База данных «Компьютерная фирма»

Схема БД состоит из четырех таблиц (рис.1.1):

- Product(maker, model, type)
- PC(code, model, speed, ram, hd, cd, price)
- Laptop(code, model, speed, ram, hd, screen, price)
- Printer(code, model, color, type, price)

Таблица Product представляет производителя (maker), номер модели (model) и тип (PC — ПК, Laptop — портативный компьютер или Printer — принтер). Предполагается, что в этой таблице номера моделей уникальны для всех производителей и типов продуктов. В таблице PC для каждого номера модели, обозначающего ПК, указаны скорость процессора — speed (МГерц), общий объем оперативной памяти - ram (Мбайт), размер диска — hd (в Гбайт), скорость считывающего устройства - cd (например, '4x') и цена — price. Таблица Laptop аналогична таблице PC за исключением того, что вместо скорости CD-привода содержит размер экрана — screen (в дюймах). В таблице Printer для каждой модели принтера указывается, является ли он цветным — color ('y', если цветной), тип принтера — type (лазерный — Laser, струйный — Jet или матричный — Matrix) и цена — price.

Схема БД состоит из четырех таблиц (рис.1.1):

- Product(maker, model, type)
- PC(code, model, speed, ram, hd, cd, price)
- Laptop(code, model, speed, ram, hd, screen, price)
- Printer(code, model, color, type, price)

Таблица Product представляет производителя (maker), номер модели (model) и тип (PC — ПК, Laptop — портативный компьютер или Printer — принтер). Предполагается, что в этой таблице номера моделей уникальны для всех производителей и типов продуктов. В таблице PC для каждого номера модели, обозначающего ПК, указаны скорость процессора — speed (МГерц), общий объем оперативной памяти - ram (Мбайт), размер диска — hd (в Гбайт), скорость считывающего устройства - cd (например, '4x') и цена — price. Таблица Laptop аналогична таблице PC за исключением того, что вместо скорости CD-привода содержит размер экрана — screen (в дюймах). В таблице Printer для каждой модели принтера указывается, является ли он цветным — color ('y', если цветной), тип принтера — type (лазерный — Laser, струйный — Jet или матричный — Matrix) и цена — price.

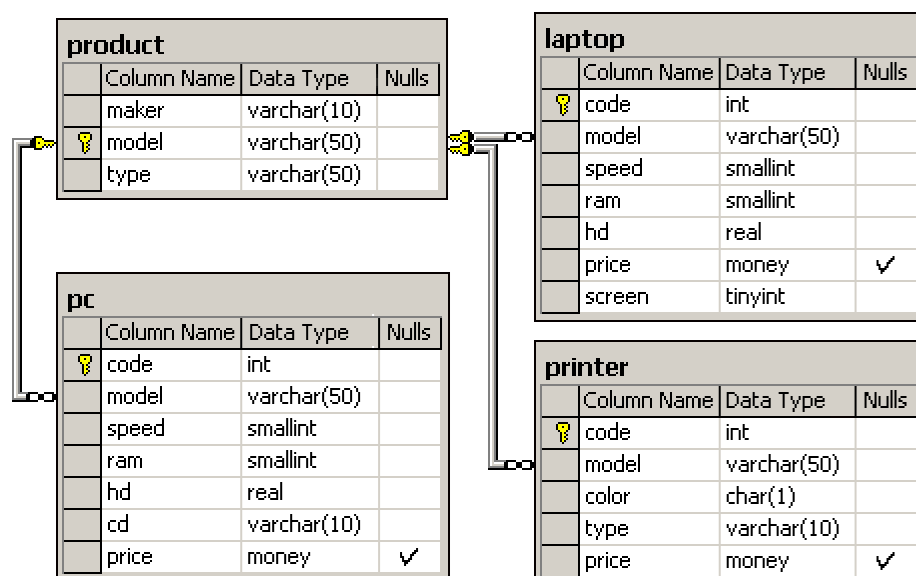


Рис. 1.1. Схема базы данных «Компьютерная фирма»

Дополнительную информацию можно извлечь из представленной на рис. 1.1 логической схемы данных. Таблицы по типам продукции (ПК, портативные компьютеры и принтеры) содержат внешний ключ (model) к таблице Product. Связь «один-ко-многим» означает, что в каждой из этих таблиц может отсутствовать модель, имеющаяся в таблице Product. С другой стороны, модель с одним и тем же номером может встречаться в такой таблице несколько раз, причем даже с полностью идентичными техническими характеристиками, так как первичным ключом здесь является столбец code. Последнее требует пояснения, так как разные люди вкладывают в понятие модели разный смысл. В рамках данной схемы считается, что модель — это единство производителя и технологии. Например, одинаковые модели могут комплектоваться технологически идентичными накопителями, но разной емкости, скажем, 60 и 80 Гбайт. В частности, это означает, что допустимо присутствие в таблице PC двух ПК с одинаковыми номерами модели, но по разной цене.

На языке предметной области данная схема может означать, что в таблице Product содержится информация обо всех известных поставщиках рассматриваемой продукции и моделях, которые они поставляют, а в остальных таблицах находятся имеющиеся в наличии (или продаже) модели. Поэтому вполне возможна ситуация, когда имеется поставщик (maker) с моделями, ни одной из которых нет в наличии.

Упражнение 1

Найдите номер модели, скорость и размер жесткого диска для всех ПК стоимостью менее 500 долларов. Вывести: model, speed и hd

Первая задача сложности 1. Даже новички легко справляются с ее решением. Действительно, одна таблица, одно условие отбора по стоимости и ограничение вывода тремя столбцами:

```
1.SELECT model, speed, hd
2.FROM PC
3.WHERE price < 500;
```

Казалось бы, какую пользу можно извлечь из анализа этой задачи? В ответ на этот вопрос предлагаем рассмотреть другое решение той же задачи:

```
4.SELECT Product.model, PC.speed, PC.hd
5.FROM Product, PC
6.WHERE Product.model = PC.model AND price < 500;
```

Решения дают один и тот же результат в силу того, что поддерживается целостность по ссылкам между таблицами PC и Product по номеру модели (столбец model). В частности, это означает, что в таблице PC не может быть модели, которой бы не было в таблице Product. Однако второй запрос не принимался системой, в результате чего автор получил возмущенное письмо от приславшего решение.

Оказалось, что при переносе баз на другой сервер некоторые связи были утеряны, в результате чего в таблице PC появилась модель с удовлетворяющими условиям задачи характеристиками и номером, который отсутствовал в таблице Product. Естественно, второе решение не выдавало этой строки, и система проверки правильности не принимала такого решения.

Несогласованность данных была устранена, связь восстановлена, и второе решение стало благополучно проходить проверку. Мораль же этой истории

заключается в том, что не нужно соединять таблицы, если в этом нет необходимости. По условиям задачи нам не нужна была информация из таблицы Product, поэтому не следовало ее использовать в запросе. Это не оправдание допущенной ошибки при удалении связи, хотя, как вы видели, даже при несогласованных данных первое решение продолжало давать отвечающий условию результат, а именно, выдавало все ПК со стоимостью менее 500 долларов.

Совет:

Если наша цель не просто научиться писать запросы, а создавать их по возможности эффективными, то следует, безусловно, избегать излишних соединений таблиц.

Помимо того, что сама операция соединения весьма затратная с точки зрения ресурсов, она может вызвать наложение ненужных блокировок на таблицы (в нашем примере – на таблицу Product), что будет приводить в состояние ожидания параллельно выполняющиеся запросы (например, на модификацию данных), адресуемые к этим таблицам. В результате будет снижаться производительность всей системы.

Упражнение 2

Найдите производителей принтеров.
Вывести: maker.

Здесь впервые встречается ошибка, характерная для нескольких задач (например, **20**, **27**, **28**). Причина в невнимательном изучении **схемы данных**. Неправильное решение:

```
1.SELECT DISTINCT maker
2.FROM Product
3.WHERE model IN (SELECT model
4.FROM Printer);
```

Таким образом, для каждой строки из таблицы Product проверяется, есть ли такая модель в таблице Printer. Связь между этими таблицами (один-ко-многим) допускает наличие модели в таблице Product, которая отсутствовала бы в таблице Printer.

Пусть, например, фирма занимается ремонтом принтеров. При этом в таблице Product содержится информация обо всех известных моделях принтеров, а в таблице Printer только о тех, которые обслуживает фирма. Например, если фирма не занимается ремонтом принтеров Sharp, то модели Sharp будут находиться в таблице Product, а в таблице Printer - нет.

В результате мы можем потерять производителя принтеров, если его моделей нет среди обслуживаемых (в таблице Printer). Как уже говорилось при обсуждении схемы данных, тип продукции в таблице Product, задается атрибутом type, который и упускается из виду.

Если вам еще не ясно, как решить эту задачу, загляните в главу 4 «Подсказки и решения».

Внимание:

Если к данной задаче имеется пояснение или приведен вариант правильного решения, в конце анализа задачи будет ставиться ссылка на соответствующую страницу этой главы - **ПиР**.

Упражнение 3

Найдите номер модели, объем памяти и размеры экранов ноутбуков, цена которых превышает 1000 долларов.

Еще одна простая задача. Однако и здесь была допущена одна поучительная ошибка. Вот то решение, которое ее содержит:

```
1. SELECT model, ram, screen
2. FROM Laptop
```

```
3.WHERE price > '1000';
```

Строковые константы в операторах SQL заключаются в одинарные кавычки. Константы числовых типов в кавычки не заключаются. Таким образом, последний предикат следовало бы записать как price > 1000. Однако здесь есть одна особенность, связанная с неявным преобразованием типов. Подробнее об этом вы можете почитать в **главе 5 (пункт 5.9)**. Здесь же следует сказать, что в SQL Server 2000 не выполняется неявное преобразование строки к значению типа money (деньги). Поэтому рассматриваемый запрос приводил к появлению сообщения об ошибке:

Disallowed implicit conversion from data type varchar to data type money, table 'Laptop', column 'price'. Use the CONVERT function to run this query.

(«Запрещено неявное преобразование типа данных varchar к типу данных money; таблица Laptop, столбец price. Используйте функцию CONVERT для выполнения этого запроса».)

Заметим, что если бы столбец price был любого другого числового типа, например, float, то неявное преобразование было выполнено, и ошибки бы не возникало. Конечно, можно выполнить явное преобразование типа; вот вполне корректная версия данного запроса:

```
1.SELECT model, ram, screen
2.FROM Laptop
3.WHERE price > CAST('1000' AS MONEY);
```

Если вы сейчас выполните запрос, вызывавший ошибку, то он вернет результирующий набор, а не указанное выше сообщение об ошибке. Дело в том, что в на сайте происходит обновление версий, и уже в SQL Server 2005 это странное отличие поведения типа money от других числовых типов при неявном приведении типов было устранено.

Таким образом, вы получите правильный результат, используя неявное приведение типа. Вот только зачем заставлять сервер тратить на это ресурсы, если можно вообще обойтись без приведения типов?

Упражнение 5

Найдите номер модели, скорость и размер жесткого диска ПК, имеющих 12х или 24х CD и цену менее 600 долларов.

При решении этой задачи обычно совершают две типичные ошибки.

Первая ошибка связана с интуитивным предположением, что скорость CD-устройства является целочисленным значением. В результате запрос

```
1.SELECT PC.model, PC.speed, PC.hd
2.FROM PC
3.WHERE PC.cd IN (12, 24) AND
4.price < 600;
```

выдаст ошибку приведения несовместимых типов данных:

Conversion failed when converting the varchar value '12x' to data type int.

(«Ошибка при преобразовании значения '12x' типа varchar к типу данных int»)

Внимательное чтение **схемы данных** скажет, что столбец cd имеет тип varchar. Поэтому, чтобы получить правильное решение достаточно переписать запрос в виде:

```
5.SELECT PC.model, PC.speed, PC.hd
6.FROM PC
7.WHERE PC.cd IN ('12x', '24x') AND
8.price < 600;
```

Вторая ошибка логическая и заключается в неправомерном использовании предиката BETWEEN. Вот это решение:

```
1.SELECT model, speed, hd
2.FROM PC
3.WHERE price < 600 AND
4.cd BETWEEN '12x' AND '24x';
```

Даже если предположить, что между моделями 12x и 24x-скоростных CD-приводов нет других моделей (скажем, 20x), решение не будет верным в силу правила сравнения строковых значений. Это правило гласит, что строки сравниваются посимвольно до первого отличающегося символа. Далее вывод о сравнении строк целиком делается на основании результата сравнения отличающихся символов. Например, справедливо 'abcz' < 'abd', так как первый отличающийся символ в первой строке ('c') меньше соответствующего символа второй строки ('d'). Если одна строка является префиксом второй (например, 'упражнения' и 'упражнениями'), то истинным будет сравнение 'упражнения' < 'упражнениями'.

Здесь уместно заметить, что сравнение (и соответственно порядок сортировки) зависит от параметра **COLLATION [3]**. Повсюду, если не оговорено противное, мы будем предполагать, что все текстовые поля имеют одинаковую установку этого параметра, обеспечивающего сравнение, независящее от регистра.

Предикат BETWEEN эквивалентен одновременному выполнению двух простых операторов сравнения:

cd >= ‘12x' AND cd <= '24x'

Исходя из вышесказанного, этому предикату будут удовлетворять, помимо требуемых задачей, например, следующие значения:

‘130x’, 145’, ‘150000000000y’ и т. д.

Еще одним вариантом решения этой несложной задачи будет использование двух предикатов простого сравнения:

```
1.SELECT PC.model, PC.speed, PC.hd
2.FROM PC
3.WHERE (PC.cd = '12x' OR
4.PC.cd = '24x') AND
5.Price < 600;
```

Упражнение 6

Для каждого производителя, выпускающего ПК-блокноты с объёмом жесткого диска не менее 10 Гбайт, найти скорости таких ПК-блокнотов. Вывод: производитель, скорость.

Вот задача, в которой впервые потребовалась информация из нескольких таблиц: имя производителя (maker) находится в таблице Product, а скорость (speed) и объем жесткого диска (hd) в таблице Laptop.

Примечание:

Пусть простят нас искушенные в SQL читатели, что мы разбираем ошибки начинающих, но в задачах такой сложности других ошибок и не бывает. Сложность задачи указана в столбце "Уровень" в ***списке задач***. *Так что можно перейти к задачам с коэффициентом сложности 2 или 3.*

```
1.  SELECT DISTINCT Product.maker, Laptop.speed
2.  FROM Product, Laptop
3.  WHERE laptop.hd >= 10 AND
4.  type IN(SELECT type
5.  FROM Product
6.  WHERE type = 'laptop');
```

При этом автор этого решения пишет, что данный запрос выдает на 5 строк больше, чем в правильном ответе, а запрос, который кажется ему более правильным:

```
1.SELECT DISTINCT Product.maker, Laptop.speed
2.FROM Product, Laptop
3.WHERE Laptop.hd >= 10;
```

выдает результат со всеми типами продуктов.

Ошибка в том, что перечисление таблиц через запятую без указания способа их соединения есть не что иное, как декартово произведение, почитать о котором можно в главе 5 (пункт 5.6).

Мы согласны с тем, что второе решение более правильное. В нем не хватает только соединения таблиц. В то время как первое — это попытка подогнать решение, ограничив выдачу второго «правильного» решения только моделями блокнотов. Следует заметить, что попытка была достаточно неуклюжей, так как, если мы правильно поняли идею автора, вместо предиката

```
7.type IN (SELECT type
8.  FROM Product
9.  WHERE type = 'laptop'
10.         )
```

достаточно было написать

```
1.type = 'laptop'
```

Упражнение 7

Найдите номера моделей и цены всех имеющихся в продаже продуктов (любого типа) производителя B (латинская буква).

Продукция в базе данных может быть трех типов: ПК, ноутбуки и принтеры. Естественным решением является объединение трех наборов по каждому типу продукции. Вот как решал эту задачу один наш участник:

```
1.SELECT model, price
2.FROM PC
3.WHERE model = (SELECT model
4.  FROM Product
```

```
5.  WHERE maker = 'B' AND
6.  type = 'PC'
7.  )
8.UNION
9.SELECT model, price
10.      FROM Laptop
11.      WHERE model = (SELECT model
12.       FROM Product
13.       WHERE maker = 'B' AND
14.       type = 'Laptop'
15.       )
16.      UNION
17.      SELECT model, price
18.      FROM Printer
19.      WHERE model = (SELECT model
20.       FROM Product
21.       WHERE maker = 'B' AND
22.       type = 'Printer'
23.       );
```

При этом на основной базе решение дает правильный результат, а на проверочной SQL Server выдает следующую ошибку:

Subquery returned more than 1 value. This is not permitted when the subquery follows =, !=, <, <= , >, >= or when the subquery is used as an expression.

(«Подзапрос возвращает более 1 значения. Это недопустимо, если подзапрос используется как выражение или с операторами сравнения =, !=, <, <= , >, >=».)

Иначе говоря, мы не можем сравнивать отдельное значение с набором, который имеет место, если производитель B выпускает более одной модели какого-либо типа, что и имеет место в проверочной базе данных.

Поправить запрос несложно, достаточно заменить предикат простого сравнения («=») предикатом попадания в список значений (**IN**):

```
1.SELECT model, price
2.FROM PC
3.WHERE model IN (SELECT model
4.  FROM Product
```

```
5. WHERE maker = 'B' AND
6. type = 'PC'
7. )
8.UNION
9.SELECT model, price
10.     FROM Laptop
11.     WHERE model IN (SELECT model
12.      FROM Product
13.      WHERE maker = 'B' AND
14.      type = 'Laptop'
15.      )
16.     UNION
17.     SELECT model, price
18.     FROM Printer
19.     WHERE model IN (SELECT model
20.      FROM Product
21.      WHERE maker = 'B' AND
22.      type = 'Printer'
23.     );
```

Заметим, что возможные повторяющиеся здесь пары значений {модель, цена} будут устранены оператором **UNION**.

Однако налицо явная избыточность: в каждом из объединяемых запросов выполняется отбор моделей производителя B. Указанный недостаток можно устранить, сначала выполнив объединение, а затем отбор по производителю:

```
1.SELECT * FROM (SELECT model, price
2. FROM PC
3. UNION
4. SELECT model, price
5. FROM Laptop
6. UNION
7. SELECT model, price
8. FROM Printer
9. ) AS a
10.     WHERE a.model IN (SELECT model
11.      FROM Product
12.      WHERE maker = 'B'
13.      );
```

При этом здесь уже не может быть отбора по типу, однако в этом нет нужды, так как номер модели уникален в таблице Product, то есть один и тот же номер не может принадлежать продукции различных типов. В результате мы получим процедурный план, содержащий 8 операций вместо 12, что имело место в первом варианте решения. Соответственно и время выполнения последнего запроса будет меньше.

Предикат **IN** будет проверяться для каждой записи объединения. Поэтому эффективность выполнения такого запроса будет зависеть от того, как далеко в списке будет находиться искомая модель. Для исключаемых моделей придется просматривать весь список. В конечном итоге время выполнения таких запросов будет тем больше, чем длиннее список (то есть чем больше моделей имеет производитель B).

Можно вместо предиката **IN** использовать соединение, однако SQL Server дает для этих двух случаев идентичные планы выполнения.

```
1.  SELECT a.model, price
2.  FROM (SELECT model, price
3.  FROM PC
4.  UNION
5.  SELECT model, price
6.  FROM Laptop
7.  UNION
8.  SELECT model, price
9.  FROM Printer) AS a JOIN
10.    Product p ON a.model = p.model
11.    WHERE p.maker = 'B';
```

Альтернативой запросам, использующим объединение, могут служить запросы на основе соединения. В данной задаче такое решение будет иметь менее эффективный план выполнения, хотя в других случаях может оказаться предпочтительным. Так или иначе, в учебных целях будет полезно рассмотреть разные способы решения задачи, что и предлагается вам выполнить самостоятельно.

Упражнение 8

Найдите производителя, продающего ПК, но не ноутбуки.

Начнем с ошибки новичка:

```
1.SELECT DISTINCT maker
2.FROM Product
3.WHERE type = 'PC' AND
4. NOT (type = 'laptop');
```

Предикат в предложении **WHERE** проверяется для каждой строки, формируемой предложением **FROM**, то есть в нашем случае для каждой строки из таблицы Product. Каждая строка представляет собой некоторую модель, которая может быть чем-то одним, либо ПК, либо ноутбуком, либо принтером. Поэтому, если выполнен первый предикат (type = 'PC'), то автоматически будет выполнен и второй — **NOT** (type = 'laptop'). Другими словами, второй предикат здесь излишен. Нам же нужно убедиться в том, что если есть строка с типом PC, то нет другой строки с типом laptop для того же поставщика.

Второе решение, верное по логике, опирается на неверную трактовку предметной области, которую мы уже обсуждали:

```
1.SELECT DISTINCT p.maker
2.FROM Product p INNER JOIN
3. PC ON p.model = PC.model
4.WHERE p.maker NOT IN (SELECT ip.maker
5. FROM Laptop il INNER JOIN
6. Product ip ON il.model = ip.model
7. );
```

Здесь проверяется наличие модели ПК в таблице PC и отсутствие модели ноутбуков для одного и того же поставщика. Ошибка заключается в том, что мы можем получить как лишних поставщиков (если в текущем состоянии базы данных в таблице Laptop отсутствуют модели некоего производителя ПК, хотя

они и есть в Product), так и недосчитаться нужных (если в текущем состоянии базы данных в таблице PC нет ни одной модели некоторого поставщика, не производящего ноутбуки).

Упражнение 10

Найдите модели принтеров, имеющих самую высокую цену. Вывести: model, price

Задача обычно не вызывает затруднений, однако, иногда встречаются решения подобные следующему:

```
1.SELECT model, MAX(DISTINCT price)
2.FROM Printer
3. GROUP BY model;
```

Понятно естественное желание решить задачу без подзапросов. Если бы требовалось вывести только максимальную цену, то тогда группировка была бы не нужна, так как максимум находился бы по всему набору принтеров:

```
1.SELECT MAX(price)
2. FROM Printer;
```

Однако в задаче требуется вывести еще и номер (номера) модели, имеющей максимальную цену. Поскольку мы не можем в предложении **SELECT** использовать агрегатные значения наряду с детализированными (если не использовать группировку по детализированным значениям), то в результате и получаем представленное выше неправильное решение с группировкой по модели. Это решение дает максимальную цену по каждой модели, нам же нужно получить модели, которые имеют абсолютную (по всему набору принтеров) максимальную цену.

Итак, приходится использовать подзапрос, в котором вычисляется максимальная цена:

```sql
1.SELECT model, price
2.FROM Printer
3.WHERE price = (SELECT MAX(price)
4.               FROM Printer
5.               );
```

При этом подзапрос может вводиться не только с простым оператором сравнения («=»), но и с предложением **IN** или >= **ALL**.

Подзапрос можно использовать и в предложении **FROM**:

```sql
1.SELECT model, price
2.FROM Printer pr, (SELECT MAX(price) AS maxprice
3.                  FROM Printer
4.                  ) AS mp
5.WHERE price = mp.maxprice;
```

Однако это не дает выигрыша в производительности, так как в любом случае вычисление подзапроса выполняется один раз, а потом уже производится сравнение цен для каждой строки.

И все же, можно ли решить задачу без подзапроса?

Упражнение 13

Найдите среднюю скорость ПК, выпущенных производителем A

Решение 1.10.1. Характерная для начинающих ошибка, когда вновь изученные конструкции языка применяются к месту и ни к месту. Вот типичный пример:

```
1.SELECT AVG(speed) AS avg_speed
2.FROM PC
3.WHERE speed IN (SELECT speed
4.                FROM PC, Product
5.                WHERE product.model = PC.model AND
6.                      maker='A'
7.                );
```

Здесь в подзапросе предложения **WHERE** отбираются значения скорости процессора ПК, выпущенных производителем А. Далее вычисляется средняя скорость по всем тем ПК, скорость процессора у которых совпадает с одним из значений в списке, полученным из подзапроса. В результате будет учтена и скорость ПК, скажем, производителя В, если она совпадает со скоростью одного из ПК, выпущенного производителем А. Правильный результат будет получен только в том случае, если производители ПК обладают моделями с уникальными наборами скоростей процессора.

Упражнение 15

Найдите размеры жестких дисков, совпадающих у двух и более PC. Вывести: HD

Неверное решение связано с поверхностным знакомством со схемой данных:

Решение 1.11.1

```
1.SELECT DISTINCT t.hd
2.FROM PC t
3.WHERE EXISTS (SELECT *
4.              FROM PC
5.              WHERE pc.hd = t.hd AND
```

```
6.                          pc.model <> t.model
7.                  );
```

В запросе находятся такие ПК, для которых существует другая модель с таким же размером жесткого диска. Ошибка заключается в интуитивном представлении об уникальности модели в таблице PC. Однако, как мы уже говорили, номера моделей уникальны лишь в таблице Product, а здесь они могут повторяться, что и делает данный запрос неверным, так как исключает из рассмотрения одинаковые модели с одинаковыми размерами жестких дисков.

Упражнение 16

Найдите пары моделей PC, имеющих одинаковые скорость и RAM. В результате каждая пара указывается только один раз, то есть (i,j), но не (j,i), Порядок вывода: модель с большим номером, модель с меньшим номером, скорость и RAM

Вот решение, которое довольно часто встречается у посетителей сайта:

```
1. SELECT  MAX(model)  AS  'model',  MIN(model)  AS  'model',
   speed, ram
2. FROM PC
3. GROUP BY speed, ram
4. HAVING MAX(model) > MIN(model);
```

Не известно, по какой причине выводят только максимальную и минимальную модель для каждой совпадающей пары значений speed, ram. Возможно, в заблуждение вводит результат «правильного» запроса на основной базе.

В этой задаче требуется упорядочить все модели, а не только максимальную и минимальную. Экстремальные характеристики упомянуты для однозначности, то есть, чтобы выводить пару моделей один раз, например:

1122	1121

но не

1121	1122

То есть если, скажем, три модели — 1122, 1121, 1135 — имеют одинаковые характеристики, то вывод должен быть таким:

1135	1122
1135	1121
1122	1121

Ниже представлено почти правильное, хотя и громоздкое решение.

Решение 1.12.2

```
1.SELECT P.model, L.model, P.speed, P.ram
2.FROM PC P JOIN
3.      (SELECT speed, ram
4.       FROM PC
5.       GROUP BY speed, ram
6.       HAVING SUM(speed)/speed = 2 AND
7.              SUM(ram)/ram = 2
8.       ) S ON P.speed = S.speed AND
9.              P.ram = S.ram JOIN
10.            PC L ON L.speed = S.speed AND
11.                   L.ram = S.ram AND
12.                   L.model < P.model;
```

Здесь в подзапросе S отбираются уникальные пары характеристик (скорость, память), совпадающие у двух ПК (**SUM**(speed)/speed = 2) — сумма

одинаковых значений, деленная на это значение, дает количество ПК. Хотя с тем же успехом можно было написать такое предложение **HAVING**:

```
1.HAVING COUNT(*) = 2
```

Подзапрос дважды соединяется с таблицей PC по этой паре характеристик. При этом второе соединение выполняется лишь для того, чтобы упорядочить модели (L.model < P.model).

Ошибка данного решения состоит в том, что число ПК с одинаковыми характеристиками может быть больше двух. В этой ситуации ни одна из таких моделей не попадет в результирующий набор представленного решения.

Несмотря на то, что решение легко исправить, лучше написать его не в такой избыточной форме.

Еще одна типичная ошибка при решении данного упражнения вызвана возможным наличием в таблице PC компьютеров одинаковых моделей. В связи с этим при выводе пар ПК необходимо исключать дубликаты.

Упражнение 17

Найдите портативные компьютеры, скорость которых меньше скорости любого ПК. Вывести: type, model, speed

Ошибки, которые здесь допускаются, связаны с излишним использованием операций соединения. Самым вопиющим примером, по мнению автора, является следующий:

```
1.SELECT DISTINCT p.type, l.model, l.speed
2.FROM Product p, Laptop l, PC c
3.WHERE l.speed < (SELECT MIN (speed)
4.                 FROM PC
5.                 ) AND
```

```
6.        p.type = 'laptop';
```

В предложении **FROM** используется декартово произведение трех таблиц! Если присутствие таблицы Product еще можно как-то оправдать — ведь в задаче требуется указать еще и тип продукции, то таблицу PC можно смело исключить — это не повлияет на результат. Естественно, решение не будет оптимальным по скорости выполнения. Кроме того, могут возникнуть проблемы с памятью, так как мощность промежуточного результата может стать огромной даже для относительно небольших таблиц. Напомним, что мощность декартового произведения равна произведению мощностей операндов. Например, для таблиц с количеством строк 100, 500 и 1000 их декартово произведение будет содержать 50000000 строк!

И, тем не менее, решение является правильным, так как предложение **DISTINCT** исключает все дубликаты, появившиеся в результате декартового произведения.

Упражнение 18

Найдите производителей самых дешевых цветных принтеров. Вывести: maker, price

Найдите две ошибки в следующем решении:

```
1. SELECT c.maker, a.priceA price
2. FROM (SELECT MIN(price) priceA
3.       FROM Printer
4.       WHERE color ='y'
5.       ) a INNER JOIN
6.       Printer b ON a.priceA = b.price INNER JOIN
7.       Product c ON b.model = c.model;
```

Упражнение 20

Найдите производителей, выпускающих по меньшей мере три различных модели ПК. Вывести: Maker, число моделей

Новички часто задают вопрос: "Почему правильным решением считается производитель E с тремя моделями, хотя запрос

```
1. SELECT Product.maker, PC.model
2. FROM PC, Product
3. WHERE Product.model = PC.model;
```

показывает, что у этого производителя имеется всего одна модель?"

Модели, которые в принципе выпускаются тем или иным производителем, содержатся в таблице Product. О таблице PC можно сказать, что это информация о компьютерах имеющихся в наличии. Соединение таблиц ограничивает выборку только моделями (вернее, конкретными ПК), имеющимися в наличии.

Поскольку в задании говорится о моделях производителей, а не о имеющихся ПК, то нужно анализировать только таблицу Product. В результате обнаружатся недостающие модели.

Упражнение 23

Найдите производителей, которые производили бы как ПК со скоростью не менее 750 МГц, так и портативные компьютеры со скоростью не менее 750 МГц. Вывести: Maker

Нижеприведенный запрос содержит характерную ошибку, допускаемую при решении этого упражнения.

Решение 1.15.1

```
1.SELECT DISTINCT maker
2.FROM product
3.WHERE model IN (SELECT model
4. FROM PC
5. WHERE speed >= 750
6. ) OR
7. model IN (SELECT model
8. FROM Laptop
9. WHERE speed >= 750
10.       );
```

Ошибка состоит в том, что в результирующий набор попадет также и производитель, выпускающий что-нибудь одно: либо ПК, либо портативные компьютеры, так как предикат в предложении **WHERE** будет истинен при выполнении хотя бы одного из условий, соединяемых оператором **OR**. Такой подход не удовлетворяет условиям задачи и совершенно справедливо отвергается системой.

Вот попытка «изменить» ситуацию в лучшую сторону:

Решение 1.15.2

```
1.SELECT DISTINCT maker
2.FROM Product a, PC b, Laptop c
3.WHERE b.speed >= 750 AND
4. c.speed >= 750 AND
5. (a.model = b.model OR
6. a.model = c.model
7. );
```

Используя равенство предикатов

```
1.x AND (y OR z) = (x AND y) OR (x AND z),
```

выполним синтаксические преобразования рассматриваемого запроса:

```
1. SELECT DISTINCT maker
2. FROM Product a, PC b, Laptop c
3. WHERE ((b.speed >= 750 AND
4.  c.speed >= 750
5.  ) AND
6.  a.model = b.model
7.  ) OR
8.  ((b.speed >= 750 AND
9.  c.speed >= 750
10.       ) AND
11.       a.model = c.model
12.       );
```

В результирующий набор попадут строки, удовлетворяющие хотя бы одному из предикатов, соединяемых оператором **OR**. Рассмотрим, например, запрос с первым предикатом:

```
1. SELECT DISTINCT maker
2. FROM Product a, PC b, Laptop c
3. WHERE ((b.speed >= 750 AND
4.  c.speed >= 750
5.  ) AND
6.  a.model = b.model
7.  );
```

Перепишем его в синтаксически более удобной форме:

```
1. SELECT DISTINCT maker
2. FROM Product a JOIN
3.  PC b ON a.model = b.model,
4.  Laptop c
5. WHERE (b.speed >= 750 AND
6.  c.speed >= 750
7.  );
```

и далее

```
1. SELECT DISTINCT maker
2. FROM (SELECT maker
3.  FROM Product a JOIN
4.  PC b ON a.model = b.model
5.  WHERE b.speed >= 750
6.  ) x,
7.  (SELECT *
8.  FROM Laptop c
9.  WHERE c.speed >= 750
10.       ) y;
```

Теперь, пожалуй, уже можно проанализировать. Первый подзапрос, который мы обозначили x соединяет по внешнему ключу таблицу PC с таблицей Product, отбирая производителей ПК со скоростью больше или равной 750. Второй подзапрос (y) фильтрует модели портативных компьютеров со скоростью больше или равной 750.

То, как соединяются x и y, называется декартовым произведением. То есть производитель требуемых ПК будет в результирующем наборе сочетаться с каждой моделью ПК-блокнота, даже если она произведена другим производителем.

В результате мы опять получим производителей, которые могут производить только что-то одно. Некоторая разница по сравнению с первым решением заключается в том, что если ни один производитель не выпускает портативные компьютеры с требуемой скоростью, то мы получим пустой набор записей. Этот частично правильный результат не дает первый пример 1.15.1.

Совпадение результатов на основной базе является совершенно случайным. Так уж оказалось, что те производители, которые выпускают требуемые по условию задачи ПК, выпускаю также и нужные портативные компьютеры. Таким образом, несмотря на совпадение результатов на «видимой» базе, запрос не является правильным при любом совместимом со схемой состоянием базы данных.

Чтобы не быть голословным, покажем результаты оригинального запроса 1.15.2 с расширением списка выводимых столбцов:

```
1.SELECT maker, a.model a_m, b.model b_m, c.model c_m
2.FROM Product a, PC b, Laptop c
3.WHERE ((b.speed >= 750 AND
4.         c.speed >= 750
5.         ) AND
6.         a.model = b.model
7.        ) OR
8.        ((b.speed >= 750 AND
9.          c.speed >= 750
10.              ) AND
11.               a.model = c.model
12.             );
```

Рассмотрим пару строк из результирующего набора:

maker	a_m	b_m	c_m
B	1121	1121	1752
A	1752	1121	1752

Как видно, модель 1121 (ПК) принадлежит производителю B, а модель 1752 (портативный компьютер) — производителю A. Так что у нас нет никаких оснований считать, что оба эти производителя удовлетворяют условиям задачи.

Объединение требуемых моделей ПК и портативных компьютеров в один набор дает лишь иллюзию, что мы получаем и то, и другое:

Решение 1.15.3

```
1.SELECT maker
2.FROM (SELECT maker
3.      FROM Product INNER JOIN
4.           PC ON Product.model = PC.model
5.      WHERE type='PC' AND
6.            speed >= 750
7.      UNION ALL
```

```
8.        SELECT maker
9.        FROM Product INNER JOIN
10.                     Laptop ON Product.model = Laptop.model
11.             WHERE type='laptop' AND
12.                   speed >= 750
13.             ) S
14.       GROUP BY maker;
```

В результате будет получен список производителей, для которых имеется хотя бы одна строка в наборе из предложения **FROM**. Ниже более короткий вариант той же ошибки.

Решение 1.15.4

```
1.SELECT maker
2.FROM Product
3.WHERE model IN (SELECT model
4.                FROM PC
5.                WHERE speed >= 750
6.                UNION ALL
7.                SELECT model
8.                FROM Laptop
9.                WHERE speed >= 750
10.                     )
11.       GROUP BY maker;
```

Следующее решение использует соединение.

Решение 1.15.5

```
1.SELECT maker
2.FROM Product INNER JOIN
3.      PC ON Product.model = PC.model INNER JOIN
4.      Laptop ON Laptop.model = Product.model
5.WHERE PC.speed >= 750 AND
6.      Laptop.speed >= 750
7.GROUP BY maker;
```

Идея состоит в следующем: получить в результирующей таблице столбец с именем производителя, столбец со скоростью ПК этого производителя, а также столбец со скоростью портативного компьютера того же производителя. Поэтому, если задать нужные ограничения на скорости, то строки, удовлетворяющие этим ограничениям, должны дать то, что нужно (группировка по производителю устраняет дубликаты). Идея правильная, однако, соединяется не то и не по тем предикатам.

Действительно, первое внутреннее соединение даст нам список производителей и номера моделей ПК. Только ПК, так как model — первичный ключ в таблице Product. Поэтому второе внутреннее соединение с таблицей Laptop (по столбцу model!) даст нам пустой результирующий набор, так как в таблице Laptop нет (и не может быть!) моделей ПК.

Чтобы реализовать эту идею, нужно соединять производителей нужных ПК с аналогичными производителями портативных компьютеров, но не по номеру модели, а по имени производителя, или же использовать внешние соединения, а не внутренние.

Упражнение 24

Найдите номер модели продукта (ПК, ПК-блокнота или принтера), имеющего самую высокую цену. Вывести: model

Вот ошибочное решение, которое какое-то время назад принималось проверочной системой сайта:

Решение 1.16.1

```
1.SELECT  model
2.FROM (SELECT  model, price
3.      FROM PC
4.      WHERE price = (SELECT MAX(price)
5.                     FROM PC
6.                     )
7.      UNION
8.      SELECT  model,  price
```

```
9.      FROM Laptop
10.           WHERE price = (SELECT MAX(price)
11.                          FROM Laptop
12.                          )
13.           UNION
14.           SELECT  model,  price
15.           FROM Printer
16.           WHERE price = (SELECT MAX(price)
17.                          FROM Printer
18.                          )
19.           ) T
20.     WHERE price = (SELECT MAX(price)
21.                    FROM Laptop
22.                    );
```

Давайте разберемся, какие данные должны быть в проверочной базе, чтобы блокировались такие решения.

Но сначала посмотрим, что же делает этот запрос. В каждом из трех аналогичных подзапросов разыскиваются максимальные по цене модели по каждому из трех видов продукции — ПК, портативным компьютерам и принтерам. Далее используется оператор UNION для объединения найденных моделей, что, помимо этого, устраняет дубликаты строк модель, цена. Наконец, отбираются только те модели, цена которых совпадает с максимальной ценой на портативные компьютеры.

Поэтому, если максимальной окажется цена на принтеры, то данное решение не будет приниматься системой. Но тогда будет приниматься решение, в котором условие отбора будет следующим:

```
1.WHERE price = (SELECT MAX(price)
2.               FROM Printer
3.               )
```

Более того, если максимальная цена будет у моделей только одного типа продукции (скажем, принтеров), то будет приниматься еще более неправильно решение:

Решение 1.16.2

```
1.SELECT   DISTINCT  model FROM Printer
2.WHERE price = (SELECT MAX(price)
3.               FROM Printer
4.               );
```

Вывод. Каковы бы ни были данные, с помощью первого запроса можно подогнать решение максимум за три попытки. Второе решение вообще не будет проходить, если максимум достигается хотя бы на двух видах продукции. Однако тогда для подгонки первого решения потребуется всего две попытки. Если же в каждом виде продукции есть модель с одной и той же максимальной ценой, то достаточно будет одной попытки.

Кстати говоря, данные проверочной базы подобраны оптимально к рассматриваемым случаям, но, тем не менее, не спасают от «принятия» неправильных решений.

Выход, и не только для данной ситуации, можно найти в увеличении количества проверочных баз, где будут смоделированы различные варианты данных. Однако это замедлит работу системы, в результате чего пользователь будет дольше находиться в состоянии ожидания ответа. Отказываясь от увеличения числа проверочных баз данных, автор успокаивает себя мыслью, что посетителями сайта движет желание изучить язык SQL и повысить квалификацию, а не стремление обмануть систему.

Рассмотрим еще один, хотя и неправильный, подход без использования **UNION**. Решение использует соединение всех моделей с последующим перебором вариантов при помощи оператора **CASE**:

Решение 1.16.3

```
1.SELECT DISTINCT CASE
2.                   WHEN PC.price > = l.price AND
3.                        PC.price > = prn.price
4.                   THEN pc.model
5.                   WHEN l.price > = PC.price AND
6.                        l.price > = prn.price
7.                   THEN l.model
```

```
8.                      WHEN prn.price > = l.price AND
9.                           prn.price > = pc.price
10.                           THEN prn.model
11.                        END AS model
12.       FROM PC, laptop l, printer prn
13.       WHERE PC.price = (SELECT MAX(price)
14.                          FROM PC
15.                          ) AND
16.             l.price = (SELECT MAX(price)
17.                         FROM Laptop
18.                         ) AND
19.             prn.price = (SELECT MAX(price)
20.                           FROM Printer
21.                           );
```

В предложении **FROM** используется декартово произведение трех таблиц. С помощью предложения **WHERE** отбираются только те строки, которые содержат модели каждого типа продукции, имеющие максимальную цену в своей продукционной категории. Возникающая здесь избыточность (если, скажем, по две модели из каждой таблицы имеют максимальную цену, то результирующее число строк будет равно восьми — 2*2*2) не является ошибочной, так как возможные дубликаты моделей будут впоследствии устранены при помощи **DISTINCT** в предложении **SELECT**. Главное, что каждая строка будет содержать искомую глобальную максимальную цену.

Затем модели с этой глобальной максимальной ценой отбираются в операторе **CASE**. Вот здесь и кроется ошибка. Особенность обработки оператора **CASE** заключается в последовательной проверке предложений **WHEN**. Поэтому при первом выполнении условия будет возвращаться значение из соответствующего предложения **THEN**, и проверка последующих предложений **WHEN** выполняться уже не будет.

Рассмотрим с этой точки зрения следующий вариант данных. Пусть максимальную стоимость имеют модели принтера и ПК. Тогда первое предложение **WHEN** оператора **CASE** будет удовлетворено:

```
1.WHEN PC.price > = l.price AND
2.     PC.price > = prn.price
3.THEN pc.model
```

Действительно, оба предиката сравнения будут истинны, в результате чего запрос вернет только модель ПК, но не принтера. Если быть более точным, то в результате мы получим все модели ПК, которые имеют одинаковую максимальную цену.

Попробуйте исправить это решение, не используя оператор **UNION**.

Упражнение 25

Найдите производителей принтеров, которые производят ПК с наименьшим объемом RAM и с самым быстрым процессором среди всех ПК, имеющих наименьший объем RAM. Вывести: Maker

Ключевой здесь является фраза «имеющих наименьший объем RAM». Она не избыточна, как это может показаться на первый взгляд. Не достаточно найти все модели, имеющие максимальную скорость среди ПК с минимальной RAM.

Поясним сказанное демонстрацией неправильных решений. Для этой задачи их немало накопилось. Вот первый пример.

Решение 1.17.1

```
1.SELECT c.maker
2.FROM Product c,
3.      (SELECT b.model, MAX(b.speed) speed
4.       FROM PC b
5.       WHERE b.ram IN (SELECT MIN(a.ram)
6.                       FROM PC a
7.                       )
8.       GROUP BY b.model
9.       ) t
10.      WHERE c.model = t.model AND
```

```
11.            EXISTS (SELECT d.model
12.                    FROM Printer d, Product e
13.                    WHERE d.model = e.model AND
14.                          e.maker = c.maker
15.                    );
```

1. Ошибка в подзапросе

```
1.(SELECT b.model, MAX(b.speed) speed
2.FROM PC b
3.WHERE b.ram IN (SELECT MIN(a.ram)
4.                FROM PC a
5.                )
6.GROUP BY b.model
7.) t
```

Здесь выбираются модели ПК с минимальной памятью, и для каждой такой модели определяется ПК с максимальной скоростью. Ошибка состоит в том, что максимальную скорость нужно определять по всем ПК с минимальной памятью, а не по каждой модели. Кроме того, если у производителя будет две модели с минимальной памятью, то он дважды попадет в результирующий набор, так как в запросе отсутствует устранение дубликатов (DISTINCT, например).

2. Ошибка в определении производителей принтеров

```
1.AND EXISTS (SELECT d.model
2.             FROM Printer d, Product e
3.             WHERE d.model=e.model AND
4.                   e.maker = c.maker
5.             )
```

Мы уже обсуждали этот вопрос (пункт 1.2).

3. Однако мы еще не выявили главной ошибки решения, которую лучше проанализировать, устранив предыдущие. В следующем решении устранены дубликаты, правильно определены производители принтеров, а также находится глобальный максимум по скорости среди моделей с минимальной памятью.

Решение 1.17.2

```
1.SELECT DISTINCT maker
2.FROM Product
3.WHERE type = 'printer' AND
4.       maker IN(SELECT maker
5.                FROM Product
6.                WHERE model IN(SELECT model
7.                               FROM PC
8.                               WHERE speed = (SELECT
  MAX(speed)
9.    FROM (SELECT speed
10.     FROM PC
11.     WHERE ram=(SELECT MIN(ram)
12.     FROM PC
13.      )
14.     ) AS z4
15.     )
16.     )
17.     );
```

Вот как определяется здесь максимум по скорости среди моделей с минимальной памятью:

```
1.speed = (SELECT MAX(speed)
2.         FROM (SELECT speed
3.               FROM PC
4.               WHERE ram = (SELECT MIN(ram)
5.                            FROM PC
6.                            )
7.               ) AS z4
8.                   )
```

Что же осталось. Вернемся к формулировке, в которой требуются «ПК с наименьшим объемом RAM и с самым быстрым процессором среди всех ПК, имеющих наименьший объем RAM». Фактически, здесь содержится два условия:

ПК с наименьшим объемом RAM

и

ПК с самым быстрым процессором среди всех ПК, имеющих наименьший объем RAM

В рассматриваемом решении используется только второе из этих условий, а именно, определяются лишь модели, имеющие скорость, совпадающую с максимальной скоростью для моделей с минимальной памятью.

Поясним на примере. Пусть минимальная память для моделей ПК в БД — 64 Мбайт и имеются следующие модели:

Speed	ram
600	64
600	128
450	64

Код, используемый для определения искомой скорости,

```
1.SELECT MAX(speed)
2.FROM (SELECT speed
3.      FROM PC
4.      WHERE ram = (SELECT MIN(ram)
5.                   FROM PC
6.                   )
7.      ) AS z4;
```

даст 600. Действительно, это максимальная скорость для моделей с минимальной (64) памятью. А далее мы отбираем модели с этой скоростью, куда попадает и модель {600, 128}, хотя она и не отвечает условиям задачи. Если производитель этой модели выпускает еще и принтеры (а он

выпускает!), да к несчастью еще и не является производителем модели {600, 64}, то получаем «неверно» при проверке запроса.

Правильным выбором будет, естественно, лишь модель {600, 64}. Надеемся, что теперь решить эту задачу не составит труда.

Упражнение 26

Найдите среднюю цену ПК и портативных компьютеров, выпущенных производителем A (латинская буква). Вывести: одна общая средняя цена

Решение 1.18.1

```
1.SELECT AVG(av.p) AS avg_price
2.FROM (SELECT AVG(price) p
3. FROM Product m, PC
4. WHERE m.model = PC.model AND
5. maker = 'A'
6. UNION
7. SELECT AVG(price) p
8. FROM Product m, Laptop l
9. WHERE m.model = l.model AND
10.       maker = 'A'
11.       ) AS av;
```

В подзапросе предложения **FROM** для производителя A объединяются средние цены на ПК и портативные компьютеры, после чего в основном запросе вычисляется среднее этих средних значений. Ошибка чисто арифметическая, которая заключается в том, что общее среднее значение (которое и нужно посчитать) не равно в общем случае среднему от средних значений.

Решение 1.18.2

```
1.SELECT ((SELECT SUM(price)
2. FROM Product INNER JOIN
3. PC ON Product.model = PC.model
4. WHERE maker='A')
5.+
6. (SELECT SUM(price)
7. FROM Product INNER JOIN
8. Laptop ON Product.model = Laptop.model
9. WHERE maker='A')
10.       ) / ((SELECT COUNT(price)
11.       FROM Product INNER JOIN
12.       PC ON Product.model = PC.model
13.       WHERE maker='A')
14.       +
15.       (SELECT COUNT(price)
16.       FROM Product INNER JOIN
17.       Laptop ON Product.model = Laptop.model
18.       WHERE maker='A')
19.       ) AS AVG_price;
```

Во втором решении сумма цен ПК и портативных компьютеров делится на их общее количество. С точки зрения математики все правильно. Но с точки зрения **SQL** — нет. Дело в том, что если в базе данных нет ПК (или портативных компьютеров), которые выпускал бы производитель A, то функция **COUNT** вернет значение 0 (что согласуется с математическими представлениями), а вот функция **SUM** вернет **NULL**-значение. В результате чего суммарная цена окажется равной **NULL**, а не суммарной цене имеющейся продукции другого типа, что хотелось бы получить.

Упражнение 27

Найдите средний размер диска ПК каждого из тех производителей, которые выпускают и принтеры. Вывести: maker, средний размер HD.

Проследите за ходом решения задачи и найдите ошибки.

1. Определим всех производителей, которые выпускают принтеры.

```
1.SELECT Product.maker
2.FROM Product INNER JOIN
3.     Printer ON Product.model = Printer.model
4.GROUP BY Product.maker;
```

2. Выведем для каждого ПК размер жесткого диска и его производителя.

```
1.SELECT PC.hd, Product.maker
2.FROM PC INNER JOIN
3.     Product ON PC.model = Product.model;
```

3. Ограничимся в пункте 2 только теми строками, которые имеют производителя, найденного в пункте 1.

```
1.SELECT PC.hd, Product.maker
2.FROM PC INNER JOIN
3.     Product ON PC.model = Product.model
4.WHERE Product.maker IN (SELECT Product1.maker
5.                       FROM Product Product1  INNER JOIN
6.                            Printer ON Product1.model =
  Printer.model
7.                       GROUP BY Product1.maker
8.                       );
```

4. В окончательном решении получаем средние значения на основе запроса из пункта 3.

```
1. SELECT Result.maker, AVG(result.hd)
2. FROM (SELECT PC.hd, Product.maker
3.       FROM PC INNER JOIN
4.            Product ON PC.model = Product.model
5.       WHERE Product.maker IN (SELECT Product1.maker
6.                                  FROM Product Product1 INNER JOIN
7.                                       Printer ON Product1.model = Printer.model
8.                                  GROUP BY Product1.maker
9.                                  )
10.          ) AS result
11.      GROUP BY result.maker;
```

Упражнение 28

Используя таблицу Product, определить количество производителей, выпускающих по одной модели.

В поддержку часто присылают такое решение от новичков:

```
1. SELECT COUNT(Maker) AS qnty
2. FROM Product
3. GROUP BY maker
4. HAVING COUNT(model)=1;
```

Помимо того, что это неверное решение задачи, новичка выдают функции COUNT(maker)/COUNT(model). COUNT(maker) - это вовсе не число производителей, а число строк в группе, для которых maker не равен NULL. COUNT(model) оказалось равным числу моделей, но не потому, что использован аргумент model, а потому, что каждая строка в таблице представляет собой модель, а model является ключом и не может быть NULL. Поскольку maker, согласно схеме, тоже не может быть NULL, то имеем

```
1. COUNT(maker) = COUNT(model) = COUNT(*) = COUNT(1) = ...
```

И все это число моделей в группе, а именно число моделей производителя, т.к. группировка выполняется по имени производителя (maker).

Итак, вот что делает данный запрос.

1. Записи в таблице Product группируются по maker с подсчетом количества строк (моделей) для каждого производителя.
2. Выполняется фильтрация групп, ограничивающая эти количества значением 1.

В результате получим:

1
1
1
...

И таких строк будет столько, сколько у нас имеется производителей, выпускающих по одной модели. Таким образом, рассматриваемый запрос отвечает следующему условию:

Для каждого производителя, выпускающего по одной модели, получить количество моделей. Вывод: количество моделей.

Согласитесь, что это совсем не то, что требовалось найти в упражнении 28. Хотя, чтобы решить задачу, осталось сделать совсем немногое, а именно, пересчитать эти строки. Это можно сделать, используя вышеприведенный запрос в качестве подзапроса (или CTE).

Если этой подсказки вам недостаточно для решения задачи, изучите следующий материал:

Получение итоговых значений.

База данных «Фирма вторсырья»

Фирма занимается приемом вторсырья и имеет несколько пунктов приема. Каждый пункт получает деньги для их выдачи сдатчикам в обмен на сырье. Фактически, на схеме представлены две базы данных. В каждой задаче по этой схеме используется только одна пара таблиц (либо с суффиксом «_o», либо без него).

В таблицах Income_o и Outcome_o первичным ключом является пара атрибутов {point, date} — номер пункта приема и дата. Этот ключ должен моделировать ситуацию, когда сведения о получении денег на приемном пункте и их выдаче сдатчикам записываются в базу данных не чаще одного раза в день.

Income

	Column Name	Data Type	Nulls
🔑	code	int	
	point	tinyint	
	[date]	datetime	
	inc	smallmoney	

Outcome

	Column Name	Data Type	Nulls
🔑	code	int	
	point	tinyint	
	[date]	datetime	
	out	smallmoney	

Income_o

	Column Name	Data Type	Nulls
🔑	point	tinyint	
🔑	[date]	datetime	
	inc	smallmoney	

Outcome_o

	Column Name	Data Type	Nulls
🔑	point	tinyint	
🔑	[date]	datetime	
	out	smallmoney	

Рис. 2.1. Схема базы данных «Фирма вторсырья»

Примечание.

Значения данных в столбце date не содержат времени, например, 2001-03-22 00:00:00.000. К сожалению, использование для этого столбца типа данных datetime может вызвать непонимание, поскольку очевидно, что учет времени не позволит ограничить многократный ввод значений с одной и той же датой (и номером пункта), но отличающихся временем дня. Этот недостаток, связанный с отсутствием отдельных типов данных для даты и времени, уже преодолен в версии SQL Server 2008. При использовании же SQL

Server 2000 обеспечить правильность ввода можно при помощи, например, следующего ограничения (CK_Income_o):

```
1.ALTER TABLE Income_o ADD
2.CONSTRAINT PK_Income_o PRIMARY KEY
3.  (
4.  [point],
5.  [date]
6.  ),
7.CONSTRAINT CK_Income_o CHECK
8.  (
9.  DATEPART(hour,[date]) + DATEPART(minute,[date]) +
10.       DATEPART(second,[date])                                    +
  DATEPART(millisecond,[date]) = 0
11.       );
```

Это ограничение (сумма часов, минут, секунд и миллисекунд равна нулю) не позволит ввести какое-либо время, отличное от 00:00:00.000. При таком ограничении первичный ключ на данной таблице будет действительно гарантировать наличие лишь одной записи в день для каждой точки.

Таблица Income_o (point, date, inc) содержит информацию о поступлении денежных сумм (inc) на пункт приема (point). Аналогичная таблица — Outcome_o (point, date, out) — служит для контроля расхода денежных средств (out).

Вторая пара таблиц — Income (code, point, date, inc) и Outcome (code, point, date, out) — моделирует ситуацию, когда приход и расход денег может фиксироваться несколько раз в день. Следует отметить, что если записывать в последние таблицы только дату без времени (что и имеет место), то никакая естественная комбинация атрибутов не может служить первичным ключом, поскольку суммы денег также могут совпадать. Поэтому нужно либо учитывать время, либо добавить искусственный ключ. Мы использовали второй вариант, добавив целочисленный столбец code только для того, чтобы обеспечить уникальность записей в таблице.

Упражнение 30

В предположении, что приход и расход денег на каждом пункте приема фиксируется произвольное число раз (в обе таблицы добавлен первичный ключ code), написать запрос с выходными данными (point, date, out, inc), в котором каждому пункту за каждую дату соответствует одна строка.

В этой задаче требуется данные из двух таблиц собрать в одном результирующем наборе; при этом приход и расход денег на пункте приема в один и тот же день должны находиться в одной строке.

Аналогичная задача (29) для таблиц Income_o и Outcome_o, как правило, затруднений не вызывала. Суть проблемы демонстрирует следующее решение.

Решение 2.1.1

```
1.SELECT Income.point, Income.date, SUM(out), SUM(inc)
2.FROM Income LEFT JOIN
3. Outcome ON Income.point = Outcome.point AND
4. Income.date = Outcome.date
5.GROUP BY Income.point, Income.date
6.UNION
7.SELECT Outcome.point, Outcome.date, SUM(out), SUM(inc)
8.FROM Outcome LEFT JOIN
9. Income ON Income.point = Outcome.point AND
10.      Income.date = Outcome.date
11.     GROUP BY Outcome.point, Outcome.date;
```

Идея решения такова. Выполняется соединение таблицы, в которой фиксируются приходы денег, с таблицей расходов средств по совпадению номера пункта приема и даты. Левое соединение, которое здесь используется, гарантирует получение результата в том случае, если на пункте приема в некоторые дни есть только приход, но нет расхода (**NULL**). Далее выполняется объединение с запросом, в котором выполняется обратное левое соединение таблицы расхода с таблицей прихода. Таким образом, учитывается

случай, когда на пункте есть расход, но нет прихода. Исключение дубликатов строк (в случаях, когда есть и приход, и расход) выполняется использованием оператора **UNION**.

Запрос 2.1.1 дает неверный результат, когда в один день на пункте приема выполняется несколько операций по приходу/расходу денежных средств. В качестве примера возьмем характерный для этого случая день — 24 марта 2001 года. Выполним пару запросов:

```
1. SELECT * FROM Income
2. WHERE date = '2001-03-24 00:00:00.000' AND point = 1;
```

```
1. SELECT * FROM Outcome
2. WHERE date = '2001-03-24 00:00:00.000' AND point = 1;
```

Получим следующий результат:

Приход

code	point	date	inc
3	1	2001-03-24 00:00:00.000	3600.0000
11	1	2001-03-24 00:00:00.000	3400.0000

Расход

code	point	date	out
2	1	2001-03-24 00:00:00.000	3663.0000
13	1	2001-03-24 00:00:00.000	3500.0000

В данном случае, когда есть и приход, и расход, внешнее соединение эквивалентно внутреннему соединению, то есть каждая строка из одной таблицы соединяется с каждой строк из другой таблицы, если в этих строках совпадают и дата, и номер пункта приема. Поэтому перед группировкой будет получен следующий результат (показаны только столбцы прихода и расхода):

inc	out
3600.0000	3663.0000
3600.0000	3500.0000
3400.0000	3663.0000
3400.0000	3500.0000

После группировки и суммирования мы получаем удвоение результата, как для прихода, так и для расхода. Если бы прихода было три, то мы бы получили утроение расхода и т. д.

И дубликаты здесь ни при чем, так как каждый из объединяемых запросов дает аналогичный результат, то есть остается одна строка на каждую пару значений {пункт, дата}.

Упражнение 32

Одной из характеристик корабля является половина куба калибра его главных орудий (mw). С точностью до 2 десятичных знаков определите среднее значение mw для кораблей каждой страны, у которой есть корабли в базе данных.

Калибр орудий, как и страна, является атрибутом таблицы Classes. Таким образом, здесь нужно найти все корабли в базе данных, для которых известен класс. Замечание об учете кораблей из таблицы Outcomes означает, как обычно, что класс

головного корабля может быть известен, даже если его нет в таблице Ships.

Затем следует добавить вычисляемый столбец для определения веса снаряда и посчитать среднее значение этого веса, сгруппировав корабли по странам.

Рассмотрим следующий запрос, отбраковываемый системой.

Решение 3.14.1

```
1.SELECT DISTINCT Classes.country,
2.  (SELECT AVG( pen.p )
3.  FROM (SELECT (c1.bore*c1.bore*c1.bore)/2 AS p
4.  FROM Classes AS c1, Ships AS s1
5.  WHERE c1.class = s1.class AND
6.  c1.country = Classes.country AND
7.  c1.bore IS NOT NULL
8.  UNION ALL
9.  SELECT (c2.bore*c2.bore*c2.bore)/2
10.       FROM Classes AS c2, Outcomes
11.       WHERE c2.country = Classes.country AND
12.       c2.class = Outcomes.ship AND
13.       c2.bore IS NOT NULL AND
14.       Outcomes.ship NOT IN (SELECT ss.name
15.       FROM Ships AS ss
16.       )
17.       ) AS pen
18.       WHERE pen.p IS NOT NULL
19.       ) AS weight
20.      FROM Classes
21.      WHERE Classes.country IS NOT NULL;
```

Запрос интересен тем, что в нем не используется группировка, а среднее значение по стране определяется с помощью коррелирующего подзапроса, выполняемого для каждой страны из таблицы Classes. Кроме того, он выполнен в полном соответствии со стандартом. Можно сразу сделать замечание относительно эффективности выполнения этого запроса, так как если у страны несколько классов кораблей (что не является для нас большой неожиданностью), то фактически подзапрос будет выполняться для каждого

класса, что явно излишне. Появляющиеся при этом дубликаты записей устраняются при помощи **DISTINCT**, что тоже скажется на производительности. Но нас интересует другой вопрос, а именно, почему этот запрос неверен. Чтобы это понять, давайте рассмотрим его по частям.

Начнем с подзапроса, в котором объединяются (**UNION ALL**) два запроса:

(1)

```
1.SELECT (c1.bore*c1.bore*c1.bore)/2 AS p
2.FROM Classes AS c1, Ships AS s1
3.WHERE c1.class = s1.class AND
4. c1.country = Classes.country AND
5. c1.bore IS NOT NULL
```

и (2)

```
1.SELECT (c2.bore*c2.bore*c2.bore)/2
2.FROM Classes AS c2, Outcomes
3.WHERE c2.country = Classes.country AND
4. c2.class=Outcomes.ship AND
5. c2.bore IS NOT NULL AND
6. Outcomes.ship NOT IN (SELECT ss.name
7. FROM Ships AS ss
8. )
```

В запросе (1) вычисляется вес снарядов кораблей из таблицы Ships для страны, передаваемой из внешнего запроса (коррелирующий подзапрос). Условие *c1.bore IS NOT NULL*, на наш взгляд, совершенно излишне, так как даже если и есть классы с неизвестным калибром, такие значения автоматически будут исключены при вычислении среднего значения с помощью функции **AVG**. Но это не ошибка в решении задачи.

В запросе (2) аналогичные вычисления делаются для головных кораблей из Outcomes, которые отсутствуют в Ships.

Далее объединение с помощью **UNION ALL** позволяет сохранить все дубликаты веса, что необходимо, так как, по крайней мере, корабли одного класса имеют снаряды одного калибра (веса).

Во внешнем запросе вычисляется среднее значение по стране, отфильтровывая случай, когда калибр неизвестен для всех кораблей некоторой страны (*WHERE pen.p IS NOT NULL*). Это объясняется тем, что если **AVG** применяется к пустому набору записей, то результат вычисления будет **NULL**.

Наконец, в основном запросе выводим требуемые по условиям задачи данные.

Вы уже нашли ошибку? Если нет, то нам помогут знания предметной области. Что за таблица Outcomes? Здесь хранятся данные об участии кораблей в сражениях. А корабль, если он не был потоплен, может принимать участие в нескольких сражениях. Таким образом, мы потенциально учитываем головной корабль несколько раз. Если же рассуждать формально, то первичный ключ на этой таблице {корабль, сражение} допускает появление одного и того же корабля неоднократно.

При этом мы не можем вместо **UNION ALL** использовать **UNION** по описанным выше причинам, но, тем не менее, исправить теперь этот запрос вам будет несложно.

Разбирая ошибки наших посетителей, автор обычно указывает те варианты данных, на которых рассматриваемые запросы возвращают неверные данные. Советуем вам наполнять свою базу аналогичными данными, тогда тестирование ваших запросов также и на других задачах будет более эффективным.

Вот еще один запрос, кстати говоря, опубликованный на форуме сайта:

```
1.SELECT Country, AVG(Wght) Wght, ISNULL(AVG(Wght),0)
2.FROM (SELECT DISTINCT Country, name, ship,
3. CASE WHEN (Outcomes.Ship IS NULL AND
4. Ships.Class IS NULL)
5. THEN NULL
6. ELSE POWER(Bore,3)/2
7. END Wght
8. FROM Outcomes FULL OUTER JOIN
9. Ships ON Outcomes.Ship = Ships.Name RIGHT OUTER JOIN
```

```
10.        Classes ON Outcomes.Ship = Classes.Class OR
11.        Ships.Class = Classes.Class
12.        ) s
13.       GROUP BY Country;
```

Это решение выполнено иначе. Здесь есть группировка по странам, полное внешнее соединение и использование нестандартных функций (SQL Server):

ISNULL(var, sub) — возвращает вместо var значение sub, если var есть **NULL**;

POWER(var, N) — возвращает значение var в степени N (возведение в степень).

Несмотря на явные отличия, этот запрос содержит ту же ошибку, что и предыдущий, то есть один и тот же корабль учитывается неоднократно, если он принимал участие в нескольких сражениях. Попробуйте исправить эту ошибку, не меняя логику самого решения запроса.

Ниже представлены три решения, содержащие разные ошибки, то есть для каждого из этих решений существует такой вариант данных, на котором данное решение дает неповторяющийся в других рассмотренных здесь решениях результат. Поэтому остановимся, чтобы не повторяться, только на этих «уникальных» для каждого решения ошибках.

Решение 3.14.3

```
1.SELECT country, AVG(bore*bore*bore/2)
2.FROM Ships s FULL JOIN
3. Outcomes o ON s.name = o.ship LEFT JOIN
4. Classes c ON c.class = ISNULL(s.class, o.ship)
5.WHERE c.class IS NOT NULL
6.GROUP BY country;
```

Ошибка заключается в соединении

```
1.Ships s FULL JOIN Outcomes o ON s.name = o.ship
```

Она проявится в том случае, если какой-либо корабль участвовал в нескольких сражениях, поскольку тогда он будет учтен несколько раз в результирующем наборе. При этом мы не утверждаем, что здесь вообще нельзя использовать полное внешнее соединение. Конечно, нет. Однако нужно позаботиться об исключении дубликатов кораблей.

Решение 3.14.4

```
1.SELECT country, AVG(bore*bore*bore/2)
2.FROM (SELECT country, bore, name
3. FROM Classes LEFT JOIN
4. Ships ON Ships.class = Classes.class
5. UNION
6. SELECT DISTINCT country, bore, ship
7. FROM Classes C LEFT JOIN
8. Outcomes O ON O.ship = C.class
9. WHERE NOT EXISTS(SELECT name
10.       FROM SHips
11.       WHERE name = O.ship) AND
12.       NOT (ship IS NULL)
13.       ) ABC
14.     GROUP BY country;
```

Рассмотрим ситуацию, когда есть класс (допустим class_1, калибр орудий 12), но нет кораблей этого класса в базе данных, и есть класс (class_2, калибр орудий 14), у которого в базе данных есть только головной корабль, упомянутый в таблице Outcomes. При этом оба класса принадлежат одной стране, скажем, country_1. Тогда первый запрос в объединении, если добавить для наглядности столбец class, в предложении **FROM** даст:

country	bore	Name	class
country_1	12	NULL	class_1
country_1	14	NULL	class_2

в то время как из второго запроса получим правильный результат:

country	bore	Name	class
country_1	14	class_2	class_2

Как видно, мы учтем в результирующем наборе две лишних строки.

Решение 3.14.5

```
1.SELECT Country, AVG(bore*bore*bore)/2
2.FROM (SELECT c.country, bore
3. FROM Classes C,
4. Ships S
5. WHERE S.class = C.Class AND
6. NOT bore IS NULL
7. UNION ALL
8. SELECT country, bore
9. FROM Classes C,
10.       OutComes O
11.       WHERE O.Ship = C.Class AND
12.       NOT EXISTS (SELECT 1
13.       FROM Ships S
14.       WHERE s.Name = O.Ship
15.       ) AND
16.       NOT bore IS NULL
17.       GROUP BY country, bore
18.       ) AS Q1
19.      GROUP BY country;
```

В подзапросе объединяются двухатрибутные отношения {страна, калибр}. Предикат второго запроса:

```
1.NOT EXISTS (SELECT 1
2. FROM Ships S
```

```
3. WHERE s.Name = O.Ship
4. )
```

исключает возможность неоднократного учета корабля, если он присутствует в обеих таблицах — Ships и Outcomes, — что как бы оправдывает использование для объединения оператора **UNION ALL**. Дубликаты же в таблице Outcomes, которые могут появиться в случае участия корабля в нескольких сражениях, устраняются, по мнению автора решения, группировкой по стране и калибру.

Однако если у страны имеется несколько классов кораблей, имеющих на вооружении орудия одинакового калибра, то возможна ситуация, когда головные корабли этих классов будут присутствовать в таблице Outcomes. В результате вместо нескольких таких кораблей учтен будет только один, что и делает данное решение ошибочным.

Упражнение 37

Найдите классы, в которые входит только один корабль из базы данных (учесть также корабли в Outcomes).

Решение 3.2.1

Вот один из запросов, которые отвергает система проверки:

```
1.SELECT class
2.FROM Ships
3.GROUP BY class
4.HAVING COUNT(name) = 1
5.UNION
6.SELECT class
7.FROM Classes c, Outcomes o
8.WHERE c.class = o.ship AND
9. NOT EXISTS (SELECT 'x'
10.       FROM Ships s
11.       WHERE o.ship = s.class
12.       );
```

Первый запрос в объединении подсчитывает корабли каждого класса из таблицы Ships, оставляя в результирующем наборе только те классы, которые имеют в этой таблице только один корабль. Второй запрос определяет классы, у которых головной корабль находится в таблице Outcomes при условии, что кораблей такого класса нет в таблице Ships.

Рассмотрим следующий пример данных, для которых этот запрос будет давать неправильный результат.

Каждый, кто решал задачи по схеме данных «Корабли», знает, что такое «Бисмарк» (Bismarck). Это головной корабль, которого нет в таблице Ships. Теперь представим себе, что другой корабль класса «Бисмарк» имеется в таблице Ships, скажем, «Тирпиц» (Tirpitz).

Тогда первый запрос вернет класс «Бисмарк», так как в таблице Ships имеется один корабль этого класса. Второй запрос класс «Бисмарк» не вернет, так как предикат:

```
1.NOT EXISTS (SELECT 'x'
2. FROM Ships s
3. WHERE o.ship = s.class
4. )
```

для корабля «Бисмарк» в таблице Outcomes будет оценен как **FALSE**. В результате объединения этих запросов получим класс «Бисмарк» в выходных данных всего запроса.

Всякому, кто внимательно следил за ходом рассуждений, понятно, что в базе данных имеется два корабля класса «Бисмарк». То есть этот класс не должен присутствовать в результатах выполнения запроса.

Чтобы проверить это, добавьте в основную базу данных следующую строку:

```
1.INSERT INTO Ships VALUES('Tirpitz', 'Bismark', 1940);
```

Совет:

Все основные базы данных можно загрузить с http://www.sql-ex.ru/db_script_download.php.

Решение 3.2.2

Следующее решение было построено одним из посетителей сайта после получения приведенных выше объяснений. Оно также дает правильный результат на основной базе данных.

```
1.SELECT class
2.FROM Ships sh
3.WHERE NOT EXISTS (SELECT ship
4. FROM Outcomes
5. WHERE ship = sh.class
6. )
7.GROUP BY class
8.HAVING COUNT(*) = 1
9.UNION
10.      SELECT ship
11.      FROM Outcomes s
12.      WHERE EXISTS (SELECT class
13.       FROM Classes
14.       WHERE class = s.ship
15.       ) AND
16.       NOT EXISTS (SELECT class
17.       FROM Ships
18.       WHERE class = s.ship
19.       );
```

Здесь объединяются два запроса. Второй запрос отбирает из таблицы Outcomes головные корабли при условии, что в таблице Ships нет других кораблей класса данного головного корабля.

В первом же запросе выбираются все корабли из таблицы Ships кроме тех, для которых в таблице Outcomes имеется головной корабль. Далее выполняется группировка по классу и отфильтровываются (**HAVING**) только те классы, в которых оказался один корабль.

Таким образом, предполагается, что если в Outcomes имеется головной корабль, то кораблей в данном классе уже минимум два и, следовательно, этот класс не отвечает условиям задачи. В этом и состоит ошибка, так как ниоткуда не следует, что в таблице Ships не может быть головного корабля. Итак, если некий класс имеет один корабль в базе данных, и этот корабль является головным и присутствует в обеих рассматриваемых таблицах, то решение **3.2.2** ошибочно проигнорирует этот класс.

Решение 3.2.3

Посмотрите теперь, как можно более просто написать запрос, содержащий аналогичную ошибку:

```
1.SELECT class
2.FROM (SELECT class
3. FROM Ships
4. UNION ALL
5. SELECT ship
6. FROM Outcomes o
7. WHERE o.ship IN(SELECT class
8. FROM Classes
9. )
10.       ) AS cl
11.      GROUP BY class
12.      HAVING COUNT(class) = 1;
```

Идея такая. В подзапросе выбираются классы всех кораблей из таблицы Ships и объединяются с головными кораблями из таблицы Outcomes с сохранением дубликатов (**UNION ALL**). При этом используется тот факт, что имя головного корабля совпадает с именем класса (SELECT Ship) (!!!). То, что дубликаты сохраняются, — это совершенно правильно, так как в противном случае мы получим на класс одну строку для любого количества кораблей в классе. Затем делается группировка по классу, и фильтруются классы, содержащие один корабль. Решение выглядит значительно короче и понятней, чем решение 3.2.2. Его и исправить будет проще, а исправлять придется, так как решение даст неверный результат, если головной корабль присутствует как в таблице Ships, так и в таблице Outcomes, в результате чего мы его дважды посчитаем.

Внимание:

Стоит обратить внимание на плодотворную идею этого решения — сначала объединить все корабли, а уже потом выполнять группировку по классам.

Решение 3.2.4

Подход на основе объединения не является единственно возможным. Следующее решение использует соединения.

```
1.SELECT Classes.class
2.FROM Outcomes RIGHT OUTER JOIN
3. Classes ON Outcomes.ship = Classes.class LEFT OUTER JOIN
4. Ships ON Classes.class = Ships.class
5.GROUP BY Classes.class
6.HAVING (COUNT(COALESCE (Outcomes.ship, Ships.name)) = 1);
```

Правое соединение таблиц Outcomes и Classes дает нам головные корабли, при этом столбец Outcomes.ship будет содержать **NULL**-значение, если головного корабля нет в таблице Outcomes. Затем выполняется левое соединение таблиц Classes и Ships по внешнему ключу. Столбец Ships.name будет содержать **NULL**-значение, если класс не имеет кораблей в таблице Ships. Полученный набор записей группируется по классу, после чего выполняется фильтрация по предикату

```
1.COUNT(COALESCE (Outcomes.ship, Ships.name)) = 1
```

Остановимся подробнее на этом элегантном приеме.

***Примечание*:**

Автор не иронизирует, когда говорит «красивый», «элегантный» и т. д. о неправильных подходах к решению задачи. Большинство рассматриваемых здесь запросов писалось профессионалами, в совершенстве владеющих языком SQL. Анализировать такие решения — хорошая школа для начинающих. А ошибки в этих решениях как правило связаны с игнорированием той или иной особенности предметной области, и зачастую легко исправляются чисто косметическими средствами.

Чтобы все было до конца ясно, приведем примеры четырех возможных вариантов строк (взятых из доступной базы данных кроме последнего случая), которые получаются в результате соединения. Вот они:

Ship	class	name
Bismarck	Bismarck	NULL
Tennessee	Tennessee	California
Tennessee	Tennessee	Tennessee
NULL	Yamato	Musashi
NULL	Yamato	Yamato
NULL	Class_1	NULL

NULL в столбце name для класса Bismarck означает, что головной корабль имеется только в таблице Outcomes. Корабли California и Tennessee класса Tennessee имеются в таблице Ships, при этом головной корабль есть также в таблице Outcomes. В третьем случае два корабля класса Yamato присутствуют в таблице Ships, в таблице же Outcomes нет головного корабля данного класса. Для четвертого случая класс Class_1 не имеет кораблей в базе данных.

Вернемся к предикату. Функция **COALESCE** (Outcomes.ship, Ships.name) вернет первое не **NULL** значение своих аргументов или **NULL**, если оба аргумента есть **NULL**-значение. Подробнее о функции **COALESCE** можно почитать в пункте 5.10.

Агрегатная функция **COUNT**, имеющая аргумент, вернет количество не **NULL**-значений аргумента в группе. Поэтому для класса Bismarck мы получим 1, для Tennessee и Yamato — 2 и, наконец, для Class_1 — 0. В результирующий набор из этих четырех классов попадает только Бисмарк, так как только он отвечает предикату.

Это решение не содержит ошибки предыдущего решения 3.2.3, то есть головной корабль, присутствующий как в таблице Ships, так и в таблице Outcomes, будет учтен только один раз. Это обусловлено тем, что при используемом соединении в результирующем наборе будет только одна строка.

Внимание:

Всегда ли справедливо для нашей базы данных последнее утверждение? Приведите пример данных, когда это будет не так. В этом, кстати, состоит еще одна ошибка данного решения.

Кроме того, в этом решении содержится та же ошибка, что и в примере 3.2.1. Если добавить в базу данных указанную там строку, то в результате соединения мы получим только одну строку на два корабля класса Бисмарк:

Ship	Class	name
Bismarck	Bismarck	Tirpitz

Если вы еще не обнаружили ошибки, упомянутой на врезке, загляните в **ПиР**.

Упражнение 38

Найдите страны, владевшие когда-либо как обычными кораблями, так и крейсерами.

Таким образом, нужно найти страны, которые имели корабли типа bc и bb. Слова «владевшие когда-либо» должно, по мнению автора, задействовать следующую логическую цепочку рассуждений:

- В текущем состоянии БД может не быть корабля какого-либо класса, хотя страна могла их иметь.
- Тогда откуда мы можем узнать, что такие корабли были? Только по имеющимся в БД классам, поскольку только в таблице Classes имеется информация о типе и стране.
- Если есть класс, скажем, типа bc (крейсер), то были и корабли этого класса, даже если их нет в таблицах Ships и Outcomes, поскольку

информация о нереализованном проекте вряд ли станет доступной.

Вывод: для решения этой задачи нужно рассматривать только таблицу Classes. В результате получаем достаточно простую задачу.

Автор не писал бы это объяснение, если бы ему не присылали подобные нижеприведенному решения с просьбой объяснить причину, по которой их не принимает система. Вот это решение:

```
1.SELECT DISTINCT c1.country
2.FROM Classes c1 INNER JOIN
3. Classes c2 ON c1.country = c2.country INNER JOIN
4. Ships s1 ON c1.class = s1.class INNER JOIN
5. Ships s2 ON c2.class = s2.class
6.WHERE c1.type = 'bb' AND
7. c2.type = 'bc'
8.UNION
9.SELECT DISTINCT c1.country
10.      FROM Classes c1 INNER JOIN
11.       Classes c2 ON c1.country = c2.country INNER JOIN
12.       Ships s1 ON c1.class = s1.class INNER JOIN
13.       Outcomes s2 ON c2.class = s2.ship
14.      WHERE c1.type = 'bb' AND
15.       c2.type = 'bc' OR
16.       c2.type = 'bb' AND
17.       c1.type = 'bc';
```

Какой формулировке соответствует это решение? Найти страны, у которых в БД имеются корабли обоих типов? Если ответ «да», то это решение все равно не является правильным.

В первом запросе объединения определяются страны, которые в таблице Ships имеют корабли обоих типов. Во втором запросе определяются страны, которые имеют в таблице Ships корабль одного типа, а в таблице Outcomes — другого.

Но есть же еще один вариант, когда имеются только головные корабли в Outcomes, причем обоих типов. Добавьте, например, в свою базу данных следующие строки:

```
1.INSERT INTO Classes
2. VALUES('c_bb', 'bb' , 'AAA' ,10 ,15 , 35000);
3.INSERT INTO Classes
4. VALUES('c_bc', 'bc', 'AAA', 6, 15, 45000);
5.INSERT INTO Outcomes
6. VALUES('c_bb', 'Guadalcanal', 'ok');
7.INSERT INTO Outcomes
8. VALUES('c_bc', 'Guadalcanal', 'ok');
```

Страна AAA имеет корабли обоих типов. Однако вышеприведенный запрос не выведет эту страну, как это и следовало ожидать.

Замечу также, что предложение **DISTINCT** в обоих запросах совершенно излишне, так как **UNION** устранит возможные дубликаты. С точки зрения логики это замечание не является существенным. Однако с точки зрения оптимизации это достаточно важный момент. Сервер тратит значительные ресурсы на удаление дубликатов, поэтому не стоит это делать несколько раз. Сравните планы выполнения запросов с **DISTINCT** и без него.

А вот пример половинчатого решения, принимаемого системой на момент его написания:

```
1.SELECT DISTINCT country
2.FROM Classes RIGHT JOIN
3. (SELECT DISTINCT COALESCE(ship, name) AS name, class
4. FROM Outcomes FULL OUTER JOIN
5. Ships ON ship = name
6. )AS z ON z.name = Classes.class OR
7. z.class = Classes.class
8.WHERE type = 'bb' AND
9. country IN (SELECT country
10.       FROM classes
11.       WHERE type = 'bc'
12.       );
```

Здесь берутся все корабли из обеих таблиц — Ships и Outcomes. Далее результат соединяется с таблицей Classes, определяется класс кораблей, и отбираются только те из них, которые имеют тип bb (боевые корабли). Наконец, проверяется, что страна найденных кораблей имеет также классы bc. Решение оказалось правильным только потому, что страны, имеющие классы обоих типов, имеют в текущем состоянии БД корабли типа ‘bb’.

Заблокировать подобные решения очень просто: достаточно добавить в таблицу Classes два класса (типа ‘bc’ и ‘bb’) для страны, которая вообще не имеет кораблей в БД. Однако лучше уточнить формулировку, скажем, так:

Найдите страны, имеющие классы как обычных боевых кораблей ('bb'), так и крейсеров ('bc').

Изменение формулировки уже выполнено на сайте. Тем не менее, надеюсь, что проведенный анализ решений оказался полезным для вас.

Упражнение 39

Найдите корабли, «сохранившиеся для будущих сражений»; то есть выведенные из строя в одной битве (damaged), они участвовали в другой.

Вот пример неправильно понятого условия:

```
1.SELECT DISTINCT ship FROM Outcomes os
2.WHERE EXISTS (SELECT ship
3. FROM Outcomes oa
4. WHERE oa.ship = os.ship AND
5. result = 'damaged'
6. ) AND
7. EXISTS (SELECT SHIP
8. FROM Outcomes ou
9. WHERE ou.ship=os.ship
10.       GROUP BY ship
11.       HAVING COUNT(battle)>1
12.       );
```

Это решение исполнено в стиле реляционного исчисления, а именно, разыскиваются такие корабли в таблице Outcomes, которые были повреждены (первый предикат **EXISTS**) и которые участвовали более чем в одной битве (второй предикат **EXISTS**).

Ошибка здесь состоит в том, что проигнорировано условие «сохранившиеся для будущих сражений», которое означает, что после битвы, в которой корабль получил повреждение, он принимал участие в более позднем сражении. Таким образом, для получения правильного решения этой задачи нужно анализировать даты сражений, которые содержатся в таблице сражений Battles.

Решение 3.4.2. Тот же результат даст и решение, использующее самосоединение:

```
1.SELECT DISTINCT o.ship
2.FROM Outcomes AS o, Outcomes AS o2
3.WHERE (o.result = 'damaged' OR
4.       o2.result = 'damaged'
5.       ) AND
6.       o.battle <> o2.battle AND
7.       o.ship = o2.ship;
```

Здесь применяется соединение таблицы Outcomes с самой собой при условии, что корабль тот же самый, а битвы разные. Кроме того, в одной из битв корабль был поврежден. Как легко увидеть, отсутствует проверка на более раннюю дату сражения, в котором корабль был поврежден.

Решение 3.4.3

Как это ни покажется странным, но нижеприведенный запрос некоторое время принимался системой проверки.

```
1.SELECT s.name
2.FROM Ships s JOIN
3. Outcomes o ON s.name = o.ship JOIN
```

```
4.  Battles b ON o.battle = b.name
5.GROUP BY s.name
6.HAVING COUNT(s.name) = 2 AND
7.  (MIN(result) = 'damaged' OR
8.  MAX(result) = 'damaged'
9.  )
10.       UNION
11.       SELECT o.ship
12.       FROM Classes c JOIN
13.        Outcomes o ON c.class = o.ship JOIN
14.        Battles b ON o.battle = b.name
15.       WHERE o.ship NOT IN (SELECT name
16.        FROM Ships
17.        )
18.       GROUP BY o.ship
19.       HAVING COUNT(o.ship) = 2 AND
20.        (MIN(result) = 'damaged' OR
21.        MAX(result) = 'damaged'
22.        );
```

Во-первых, объединяются запросы, которые выполняют соединение участвующих в сражениях кораблей (таблица Outcomes) с таблицами Ships и Classes соответственно. Кстати говоря, предикат

```
1.o.ship NOT IN (SELECT name FROM Ships)
```

во втором запросе явно лишний, так как UNION исключит возможные дубликаты.

Эти соединения не просто избыточны, они ошибочны, так как в описании базы данных сказано, что в таблице Outcomes могут быть корабли, отсутствующие в Ships. То есть если найдется не головной корабль, которого нет в таблице Ships и который отвечает условиям задачи, то он не попадет в результирующий набор вышеприведенного запроса.

Во-вторых, предикат

```
1.HAVING COUNT (o.ship) = 2
```

ограничивает возможные варианты только двумя сражениями корабля. А почему корабль не может принимать участие более чем в двух сражениях? Он же не обязательно был потоплен после того, как получил повреждение. Причем он мог участвовать в сражениях и до повреждения (например, с результатом ok). Тогда, если в следующем и последним для корабля сражении он был поврежден, то запрос 3.4.3 выведет этот корабль, хотя это и не отвечает условиям задачи.

В-третьих, автору не вполне понятно условие:

```
1. (MIN(result) = 'damaged' OR MAX(result) = 'damaged')
```

Примечание:

В связи с последним предикатом хочется напомнить читателям, что запросы, которые мы анализируем, были написаны посетителями сайта. Допускаемые ошибки не надуманы, а являются естественным следствием процесса обучения, когда формальное знание языка программирования применяется к решению конкретных задач. Собственно, эта книга и написана для того, чтобы облегчить переход от формального знания к практическому умению.

Однако вернемся к условию. В соответствии с описанием предметной области корабль может быть:

- поврежденным (damaged);
- остаться невредимым (ok);
- быть потопленным (sunk).

Поэтому условие MIN(result) = 'damaged' будет выполнено, если в одной из битв корабль был поврежден (при естественной сортировке текстовых строк буква «d» идет раньше, чем буквы «o» и «s»). Однако это совсем не означает, что поврежден он был прежде, чем принял участие в следующем сражении, что требуется по условиям задачи. Здесь нужно оценивать даты сражений. Что же касается **MAX**(result) = 'damaged', то это условие не будет выполняться, если результаты сражений были разные; если же они были одинаковые, то это не даст ничего нового по сравнению с первым условием на минимум.

Вот такое наложение ошибок давало правильный результат на обеих проверочных базах. Меры уже приняты: добавлены проверочные данные, на которых данное решение дает неверный результат. Как это и должно быть по логике этого запроса.

Упражнение 46

Укажите названия, водоизмещение и число орудий, кораблей участвовавших в сражении при Гвадалканале (Guadalcanal).

Все нужные нам корабли, принимавшие участие в сражении при Гвадалканале, находятся в таблице Outcomes, а требуемые характеристики — в таблице Classes. Поэтому первое, что приходит в голову, — это соединить эти таблицы для получения нужного результата:

Решение 3.1.1

```
1. SELECT Outcomes.ship, Classes.displacement,
   Classes.numGuns
2. FROM Classes RIGHT JOIN
3.  Outcomes ON Classes.class = Outcomes.ship
4. WHERE Outcomes.battle = 'Guadalcanal';
```

Внешнее соединение здесь используется законно, так как поскольку в задании сказано о кораблях, участвовавших в сражении, то выводить нужно все такие корабли вне зависимости от того, совпадает его имя с именем класса или нет. Заметим, что внутреннее соединение вернет пустой набор записей, так как в основной базе данных не оказалось головных кораблей, участвовавших в этом сражении. А так мы имеем:

ship	displacement	numGuns
California	NULL	NULL
Kirishima	NULL	NULL
South Dakota	NULL	NULL
Washington	NULL	NULL

Правильным же ответом является:

Ship	displacement	numGuns
California	32000	12
Kirishima	32000	8
South Dakota	37000	12
Washington	37000	12

Откуда же берется эта информация? Информация о классе корабля содержится в таблице Ships, то есть нужна еще одна таблица. Итак, если корабль из Outcomes имеется в Ships (Outcomes.ship = Ships.name), то нам известен его класс, и, следовательно, вся необходимая информация о нем может быть взята из таблицы Classes (Ships.class = Classes.class). Рассмотрим решение, которое выполняет нужные соединения:

Решение 3.1.2

```
1.SELECT o.ship, c.displacement, c.numGuns
2.FROM Outcomes o LEFT JOIN
3. Ships s ON o.ship = s.name LEFT JOIN
4. Classes c ON s.class=c.class
5.WHERE o.battle = 'Guadalcanal';
```

На основной базе получаем правильный результат, однако, система не принимает решение. При этом левое соединение гарантирует появление корабля в выходном наборе даже в том случае, если его класс неизвестен (корабля нет в Ships). В последнем случае будет получена строка типа:

```
1.Корабль NULL NULL
```

Ошибка заключается в пресловутом «Бисмарке». Не именно в нем, а в той ситуации, когда в Outcomes имеется головной корабль, которого нет в Ships. Предположим, что «Бисмарк» участвовал в сражении при Гвадалканале. Рассматриваемый нами запрос вернет такую строку:

```
1.Bismarck NULL NULL
```

так как этого корабля нет в Ships. Однако его характеристики нам известны, поскольку известен класс корабля (головной корабль!). Правильной строкой будет:

```
1.Bismarck 8 42000
```

Строку же

```
1.Корабль NULL NULL
```

мы получаем только в том случае, если в битве принимал участие неголовной корабль, отсутствующий в таблице Ships. Подобная ситуация могла бы еще возникать и при неизвестном классе корабля в таблице Ships, однако, она исключается ограничением **NOT NULL** на столбце class в этой таблице.

В заключение приведу еще одно решение, содержащее ту же ошибку, но не использующее внешние соединения:

Решение 3.1.3

```
1.SELECT a.ship, b.displacement, b.numguns
2.FROM Outcomes a, Ships c, Classes b
3.WHERE a.battle='Guadalcanal' AND
4. a.ship = c.name AND
5. c.class = b.class
```

```
6.UNION
7.SELECT a.ship, NULL AS displacement, NULL AS numguns
8.FROM Outcomes a
9.WHERE a.battle = 'Guadalcanal' AND
10.       a.ship NOT IN (SELECT name
11.       FROM Ships
12.       );
```

Заметим, что первое рассмотренное нами решение дало бы правильный результат для такого головного корабля. Поэтому чтобы решить эту задачу, нужно второе решение дополнить первым. Как не следует «дополнять», можно посмотреть в главе 4.

Упражнение 51

Найдите названия кораблей, имеющих наибольшее число орудий среди всех кораблей такого же водоизмещения (учесть корабли из таблицы Outcomes).

Решение 3.8.1. Не очень оптимальное решение и, к тому же, содержащее ошибку.

```
1.SELECT name
2.FROM (SELECT O.ship AS name, numGuns, displacement
3. FROM Outcomes O INNER JOIN
4. Classes C ON O.ship = C.class AND
5. O.ship NOT IN (SELECT name
6. FROM Ships
7. )
8. UNION
9. SELECT S.name AS name, numGuns, displacement
10.       FROM Ships S INNER JOIN
11.       Classes C ON S.class = C.class
12.       ) OS INNER JOIN
13.       (SELECT MAX(numGuns) AS MaxNumGuns, displacement
14.       FROM Outcomes O INNER JOIN
15.       Classes C ON O.ship = C.class AND
```

```
16.        O.ship NOT IN (SELECT name
17.        FROM Ships
18.        )
19.        GROUP BY displacement
20.        UNION
21.        SELECT MAX(numGuns) AS MaxNumGuns, displacement
22.        FROM Ships S INNER JOIN
23.        Classes C ON S.class = C.class
24.        GROUP BY displacement
25.        ) GD ON OS.numGuns = GD.MaxNumGuns AND
26.        OS.displacement = GD.displacement;
```

В предложении **FROM** данного решения соединяются два подзапроса. В первом из них определяется имя, число орудий и водоизмещение всех имеющихся в базе данных кораблей. Эти корабли собираются по двум таблицам — Ships и Outcomes (головные корабли). При этом выполняется неправильная и излишняя проверка на дубликаты:

```
1.O.ship NOT IN (SELECT name
2.  FROM Ships
3.  )
```

Почему неправильная? Да потому, что она все равно оставляет дубликаты, учитывая головной корабль столько раз, сколько раз он участвовал в сражениях. Ну, а излишней она является потому, что предложение **UNION** все равно устранит дубликаты. Это в данном случае оказалось совсем нелишним, в результате чего подзапрос, хотя и не оптимальный, дает то, что и предполагалось по алгоритму.

Второй подзапрос в соединении определяет максимальное число орудий для каждого значения водоизмещения имеющихся кораблей, при этом опять, как и ранее, эти значения определяются отдельно для кораблей из Ships и головных кораблей из Outcomes с последующим объединением.

Соединение выполняется по совпадению числа орудий и водоизмещения в строках этих подзапросов.

Логика построения решения вполне верная; не верна реализация. Чтобы доказать это, обычно прибегают к контрпримеру. Другими словами, приведем пример данных, на котором этот запрос даст неверное решение задачи. Итак,

пусть только в таблице Ships есть корабли водоизмещением 40 000 тонн с максимальным числом орудий 16, и только в таблице Outcomes имеется головной корабль водоизмещения 40 000 тонн и числом орудий 17. Тогда второй подзапрос из соединения даст нам две строки:

16	40000
17	40000

поскольку это не дубликаты, обе эти строки будут присутствовать в результирующем наборе. В результате соединения мы получим не только корабли с максимальным числом орудий для данного водоизмещения — 17, но и корабли, имеющие на вооружении 16 орудий. Узнаете ошибку? Она уже встречалась ранее: сначала нужно делать объединение, а потом группировку.

Упражнение 53

Определите среднее число орудий для классов линейных кораблей. Получить результат с точностью до двух десятичных знаков.

Автор полагал, что в этой задаче лишь одна проблема — округление. Однако как-то поступило следующее решение:

```
1.SELECT SUM(sum_g)/SUM(count_g)
2.FROM (SELECT SUM(numGuns) AS sum_g, COUNT(*) AS count_g
3. FROM Classes INNER JOIN
4. Ships ON Classes.class = Ships.class
5. WHERE type = 'bb'
6. UNION
7. SELECT SUM(numGuns) AS sum_g, COUNT(*) AS count_g
8. FROM Classes INNER JOIN
9. Outcomes ON Classes.class = Outcomes.ship
10.      WHERE type='bb'
```

```
11.        ) AS a;
```

Богатое для анализа ошибок решение. Начнем с округления. Число орудий — целое число (по типу столбца, а не по смыслу!). Поэтому и сумма будет числом целым. При делении целых чисел в SQL Server всегда получается целое число. Причем результат достигается не округлением, а отбрасыванием дробной части. Выполните, например, следующий запрос

```
1.SELECT 2/3;
```

Результатом будет 0, что подтверждает сказанное. Поэтому, чтобы внести косметические исправления данного запроса, нужно выполнить преобразование хотя бы одного операнда к вещественному типу. Как сказано в пункте 5.9, можно воспользоваться неявным преобразованием типа:

```
1.SELECT SUM(sum_g)*1.0/SUM(count_g)
```

то есть при умножении на вещественную единицу числитель становится вещественным числом.

Теперь, поскольку требуется определить среднее по классам, то, во-первых, не нужно учитывать корабли, а, во-вторых, не нужно учитывать корабли из таблицы Outcomes.

Однако чтобы проанализировать допущенные ошибки, давайте рассмотрим решение в трактовке автора этого запроса, то есть определим среднее значение по всем линейным кораблям из базы данных, а это не что иное, как задача 54. Эта задача рассматривается в пункте 3.10, но там решение содержит другую ошибку.

Итак, в подзапросе подсчитывается число орудий и количество отдельно по кораблям из таблицы Ships и головным кораблям из таблицы Outcomes. Затем в основном запросе суммируется число орудий и количество кораблей, полученных по каждой таблице, и делится одно на другое, чтобы получить среднее значение.

Рассмотрим пример. Пусть в Ships есть 2 корабля с 11 и 8 орудиями, а в Outcomes — один корабль с 11 орудиями. Итого получаем 3 корабля и 30 орудий. Среднее 30/3 = 10. Правильно? Нет, то есть правильно, но не во всех случаях. Нам же нужно написать запрос, который будет верен на любых данных. Я вижу здесь несколько контрпримеров.

Первый контрпример. А если в Outcomes не будет головного корабля, отвечающего условиям задачи? Тогда второй подзапрос даст: кораблей — 0, число орудий — NULL. В результате вычисление среднего в рассматриваемом запросе даст

```
1. (19 + NULL)/(2+0)  = NULL
```

вместо 19/2.

Второй контрпример. Пусть головной корабль класса bb есть как в таблице Ships, так и в таблице Outcomes, то есть это один и тот же корабль. Тогда в результате мы должны получить не 30/3, что нам дает представленное решение, а 19/2.

Третий контрпример. А если в предыдущей ситуации по кораблям головной корабль дважды принимал участие в сражениях? Тогда мы получим вместо тех же 19/2 — (19 + 22)/(2+2) = 41/4.

Четвертый контрпример… Придумайте сами. Вот так и формируется проверочная база сайта.

Упражнение 54

С точностью до двух десятичных знаков определите среднее число орудий всех линейных кораблей (учесть корабли из таблицы Outcomes).

Решение 3.10.1

```
1. SELECT ROUND(SUM(ng)/SUM(cnt), 2) res
```

```
2. FROM (SELECT SUM(numGuns) ng, COUNT(*) cnt
3.  FROM Classes c,Ships s
4.  WHERE c.class = s.class AND
5.  c.type = 'bb'
6.  UNION ALL
7.  SELECT SUM(numGuns) ng, COUNT(*) cnt
8.  FROM Classes c, Outcomes o
9.  WHERE c.class = o.ship AND
10.       c.type = 'BB'AND
11.       NOT EXISTS (SELECT name FROM Ships s
12.       WHERE s.name = o.ship)
13.       ) x;
```

В этом решении сделана попытка вручную посчитать среднее как сумму значений, деленную на их количество. Однако специфика арифметических операций в SQL Server состоит в том, что результат всегда приводится к типу аргумента. Поскольку число орудий — целое число (тип INTEGER для столбца numGuns), то дробная часть числа, полученного при делении, будет попросту отбрасываться, что заведомо даст неправильный результат.

***Внимание*:**

Использование функции AVG для вычисления среднего не меняет ситуацию, так как приведение типа проводится по тем же правилам. Это легко проверить, если выполнить запрос

```
1. SELECT AVG(3/2);
```

который даст 1, а не 2, если бы выполнялось округление.

Если вы будете выполнять аналогичные запросы на сайте, поставьте флажок «Без проверки» на странице с упражнениями, чтобы система не выполняла бесполезного сравнения результата с эталонным решением соответствующего упражнения.

Для получения «точного» результата деления целых чисел нужно привести операнд (хотя бы один) к вещественному типу. Это можно сделать с помощью функции приведения типа **CAST** или простым умножением на вещественную единицу, как мы и поступим:

```
1.SELECT SUM(numGuns*1.0) ng
```

Теперь поговорим об округлении, которое использует функцию T-SQL ROUND. Опять обратимся к простому примеру (округление до двух цифр после десятичной точки):

```
1.SELECT ROUND(AVG(5.0/3),2);
```

который даст нам 1.670000 в качестве результата. То есть округление выполнено правильно, но сохранены незначащие нули, количество которых соответствует числу значащих цифр, используемых по умолчанию для представления вещественного числа. Это число, естественно, зависит от реализации, поэтому в данном случае мы говорим лишь об SQL Server. Здесь уместно заметить, что при сравнении результатов с «правильным» решением значения 1.67 и 1.670000 будут считаться разными. Поэтому нужно позаботиться еще и об удалении этих нулей. Отложим этот вопрос до анализа следующего решения, так как там эта проблема, как и проблема округления, решена верно. Там же мы рассмотрим и логическую ошибку, которую содержит решение 3.10.1.

Решение 3.10.2

```
1.SELECT CAST(AVG(numGuns*1.0) AS NUMERIC(10,2))
2.FROM (SELECT numguns
3. FROM Classes c JOIN
4. Ships s ON c.class = s.class
5. WHERE type = 'bb'
6. UNION ALL
7. SELECT numguns
8. FROM Classes] c JOIN
9. Outcomes o ON c.class = o.ship
10.      WHERE type='bb' AND
11.      o.ship NOT IN(SELECT name
12.      FROM Ships
13.      )
14.      ) t;
```

Обратите внимание на приведение типа к числу с фиксированной точкой, которое и выполняет требуемое округление результата.

В подзапросе объединяются (**UNION ALL**) два запроса. Первый определяет число орудий для кораблей в таблице Ships, принадлежащих классам линейных кораблей (тип bb). Второй учитывает головные корабли соответствующих классов при условии, что их нет в таблице Ships.

Таким образом, сделана попытка учесть каждый корабль в БД только один раз. Поскольку для объединения используется **UNION ALL**, то дубликаты устраняться не будут. Это совершенно справедливо, так как многие корабли будут иметь одинаковое число орудий, а в предложении **SELECT** подзапроса выводится только этот столбец.

И все же ошибка связана именно с использованием **UNION ALL**. Поступим формально, то есть не будем домысливать предметную область, а обратимся к схеме (рис. 3.1). В таблице Ships первичным ключом является имя корабля, поэтому первый запрос в объединении даст нам по одной строке на каждый корабль известного класса. В таблице же Outcomes ключом является пара {ship, battle}, то есть уникальность обеспечивается для комбинации имени корабля и сражения, в котором он принимал участие. Отсюда следует, что один и тот же корабль может несколько раз упоминаться в таблице Outcomes, что соответствует участию данного корабля в нескольких сражениях.

В результате второй запрос в объединении даст дубликаты кораблей, если головной корабль участвовал в нескольких сражениях. Это и делает ошибочным представленное решение.

С другой стороны, и **UNION** вместо **UNION ALL** мы написать не можем по указанной выше причине.

Упражнение 55

Для каждого класса определите год, когда был спущен на воду первый корабль этого класса. Если год спуска на воду головного корабля неизвестен, определите минимальный год спуска на воду кораблей этого класса. Вывести: класс, год.

Видимо, ошибки вызваны наличием ловушек в предыдущих задачах. При решении задач со сложностью 1 делаются попытки учесть все и вся.

Сам по себе учет излишних фактов не делает решение неверным, разве что увеличивает стоимость выполнения запроса, но дело в том, что при этом допускались столь характерные ошибки, что автор решил рассмотреть несколько подобных решений.

Что же лишнего пытаются учесть при решении этой задачи? Это — головные корабли из таблицы Outcomes. Таблица Outcomes вообще не нужна для решения этой задачи. Ведь нам нужно определить год, который является атрибутом таблицы Ships. Поэтому даже если в таблице Outcomes есть головной корабль, отсутствующий в таблице Ships, то мы все равно не знаем года его спуска на воду. В случае же отсутствия в БД других кораблей этого класса наличие такого корабля все равно ничего не дает, так как результат должен выглядеть следующим образом:

Класс	NULL

поскольку в задании сказано «для каждого класса». Тем самым утверждается, что для данного класса год спуска первого корабля неизвестен. Но такую строку в результирующем наборе мы можем получить и без таблицы Outcomes, выполнив внешнее соединение таблицы Classes с таблицей Ships. Перейдем к анализу решений, в которых были допущены ошибки при учете кораблей из таблицы Outcomes.

Решение 3.11.1

```
1.SELECT C.class , launched
2.FROM Classes C ,
3. (SELECT name , class , launched
4.  FROM Ships
5.  UNION
6.  SELECT ship , ship , NULL
7.  FROM Outcomes O
8.  WHERE NOT EXISTS (SELECT *
```

```
9. FROM Ships S
10.       WHERE S.class = O.ship
11.       )
12.       UNION
13.       SELECT ship , ship , MIN(launched)
14.       FROM Ships S ,
15.       Outcomes O
16.       WHERE S.class = O.ship
17.       GROUP BY ship
18.       ) S
19.     WHERE C.class = S.name;
```

Рассмотрим подзапрос S, в котором объединяются три запроса, результирующий набор каждого из которых содержит три столбца {имя корабля, класс, год спуска на воду}. В первом из них выбираются все корабли из таблицы Ships. Во втором выбираются корабли из Outcomes, имя которых не совпадает ни с одним классом кораблей из таблицы Ships. При этом в качестве года спуска на воду используется **NULL**, что правильно, а имя корабля ассоциируется с именем класса (**SELECT** ship, ship, NULL). Последнее не является здесь ошибкой, так как в последующем будут отбираться только головные корабли:

```
1.WHERE C.class = S.name
```

Наконец, в третьем запросе определяется минимальный год для классов кораблей, у которых головной корабль имеется в таблице Outcomes.

Принципиальная ошибка заключается в наличии ситуации, когда головной корабль из Outcomes присутствует также и в таблице Ships с неизвестным годом спуска на воду. Кроме того, имеются и другие корабли того же класса с известным годом спуска на воду. Тогда первый запрос даст строку с годом **NULL**, а третий — с минимальным годом по этому классу. В итоге получим две строки, которые не являются дубликатами и, следовательно, не будут исключены использованием **UNION**.

Наконец, условие отбора только по головным кораблям исключит из рассмотрения классы, вообще не имеющие головных кораблей.

Решение 3.11.2

```
1.SELECT DISTINCT class, MIN(launched)
2.FROM Ships GROUP BY Class
3.UNION
4.SELECT DISTINCT Ship AS class, NULL
5.FROM Outcomes
6.WHERE ship IN (SELECT class
7. FROM Classes
8. ) AND
9. ship NOT IN (SELECT name
10.       FROM Ships
11.       );
```

Первый запрос в объединении считает минимальный год спуска на воду по классам кораблей из таблицы Ships. Во втором запросе выбираются те головные корабли из таблицы Outcomes (предикат **IN**), которых нет в таблице Ships (предикат **NOT IN**). Здесь, как и в решении 3.11.1, такой корабль (класс) учитывается с **NULL** в качестве года спуска на воду. Однако ошибка здесь уже другая. Суть ее заключается в ситуации, когда в Outcomes есть головной корабль, которого нет в таблице Ships (пусть это будет корабль «Бисмарк»), но в Ships есть другой корабль класса «Бисмарк» с известным годом спуска на воду. В результате мы опять получаем две строки на класс, с известным и неизвестным годом спуска на воду.

По поводу решения 3.11.2 следует отметить совершенно излишнее, и даже вредное с точки зрения производительности, задействование **DISTINCT**. В первом из объединяемых запросов группировка делает появление дубликатов невозможным. Во втором запросе, если и будут дубликаты (участие корабля в нескольких сражениях), они устраняются использованием при объединении предложения **UNION**.

Теперь рассмотрим решения, где не применяется таблица Outcomes, но, тем не менее, допускается логическая ошибка.

Решение 3.11.3 (с комментариями автора решения)

```
1./*
2.Год спуска на воду головных кораблей
3.*/
4.SELECT class, launched AS year
```

```
5.FROM Ships
6.WHERE name = class
7.UNION
8./*
9.Минимальный год спуска на воду кораблей по классам,
10.      у которых нет данных по головным кораблям в таблице
  Ships
11.      */
12.      SELECT class, MIN(launched)
13.      FROM Ships
14.      WHERE class NOT IN (SELECT class
15.       FROM Ships
16.       WHERE name = class
17.       )
18.      GROUP BY class
19.      UNION
20.      /*
21.      Выводим NULL в качестве года спуска на воду для
  классов,
22.      кораблей которых нет в Ships
23.      */
24.      SELECT class, NULL
25.      FROM classes
26.      WHERE class NOT IN(SELECT class
27.       FROM ships
28.       );
```

В этом решении в соответствии с условием задачи учтены все классы, включая те, которые не имеют кораблей в БД (последний запрос в объединении). По-видимому, допущенная здесь ошибка связана с попыткой отдельного учета головных кораблей (которые являются первыми кораблями в классе и, следовательно, имеют наименьший год спуска на воду) и классов, не имеющих головных кораблей в базе данных.

Представим ситуацию, когда год спуска головного корабля неизвестен (**NULL**), но при этом он имеется в таблице Ships. Там же находится другой корабль аналогичного класса с известным годом спуска на воду. Тогда именно этот год должен по условию задачи фигурировать в выходном наборе.

Данное же решение даст (первый запрос в объединении) **NULL**, в то время как корабль с нужным годом будет проигнорирован во втором запросе объединения в силу следующей фильтрации

```
1.WHERE class NOT IN (SELECT class
2. FROM Ships
3. WHERE name = class
4. )
```

Аналогичная ошибка содержится и в нижеследующем варианте, который предлагается проанализировать самостоятельно.

Решение 3.11.4

```
1.SELECT class,
2.  (SELECT launched
3.  FROM (SELECT launched
4.  FROM Ships sh
5.  WHERE cl.class = sh.name
6.  UNION
7.  SELECT launched
8.  FROM Ships sh
9.  WHERE launched = (SELECT MIN(launched)
10.       FROM ships sh2
11.       WHERE class = cl.class AND
12.       NOT EXISTS(SELECT launched
13.       FROM ships sh
14.       WHERE cl.class = sh.name
15.       )
16.       )
17.       ) tab
18.       ) year
19.     FROM classes cl;
```

Упражнение 56

Для каждого класса определите число кораблей этого класса, потопленных в сражениях. Вывести: класс и число потопленных кораблей.

Решение 3.12.1

```
1.SELECT aa.class, SUM(aa.sunks) Sunks
2.FROM (
3.-- 1
4.SELECT c.class, COUNT(a.ship) sunks
5.FROM Outcomes a INNER JOIN
6.Ships b ON a.ship = b.name INNER JOIN
7.Classes c ON b.class = c.class
8.WHERE a.result = 'sunk'
9.GROUP BY c.class
10.      UNION
11.      -- 2
12.      SELECT c.class, COUNT(a.ship)
13.      FROM Outcomes a INNER JOIN
14.      Classes c ON a.ship = c.class
15.      WHERE a.result = 'sunk'
16.      GROUP BY c.class
17.      UNION
18.      -- 3
19.      SELECT c.class, 0
20.      FROM Classes c
21.      ) aa
22.      GROUP BY aa.class;
```

В подзапросе предложения **FROM** объединяются три таблицы:

Класс и число потопленных кораблей, которые есть в таблице Ships.

Класс и число потопленных головных кораблей класса. Здесь уже есть «излишество», а именно: нет необходимости использовать группировку и соответственно функцию **COUNT**, так как у класса может быть только один головной корабль, да и потоплен корабль может быть только однажды.

Каждый класс с нулевым количеством потопленных кораблей. Это позволяет учесть те классы, которые не имеют потопленных кораблей и, следовательно, не попадают в предыдущие два набора записей.

Объединение с использованием **UNION** устраняет дубликаты, что, по мнению автора решения, позволяет корректно обработать ситуацию, когда потопленный головной корабль присутствует в таблице Ships. Наконец, выполняется группировка по классам с суммированием. При этом последний набор не дает вклада в окончательный результат, если в классе имеются потопленные корабли, что правильно.

Однако ошибка кроется в том, что объединяются двухатрибутные кортежи {класс, число потопленных кораблей}. Поэтому если в некотором классе (опять «Бисмарк») имеется два потопленных корабля, причем головной корабль отсутствует в Ships, то объединяться будут два одинаковых кортежа

Бисмарк	1

Тогда после устранения дубликатов мы получаем один потопленный корабль вместо двух.

Но это еще не все. Даже головной корабль мы можем посчитать дважды, если он присутствует в Ships. Это справедливо для случая, когда есть и другие корабли этого класса, потопленные в сражениях. Давайте опять возьмем для примера «Бисмарк», только теперь он присутствует также в таблице Ships. Пусть есть и еще один потопленный корабль (естественно, не головной) этого класса. Тогда первый набор даст:

Бисмарк	2

а второй:

Бисмарк	1

В результате мы получим

Бисмарк	3

хотя на самом деле корабля всего два.

Вот еще одно решение задачи, в котором не используется объединение, но содержится другая ошибка:

Решение 3.12.2

```
1.SELECT classes.class, COUNT(ship) sunked
2.FROM Classes FULL JOIN
3. Ships ON classes.class = ships.class LEFT JOIN
4. (SELECT ship
5. FROM Outcomes
6. WHERE result = 'sunk'
7. ) s ON s.ship = ships.name OR
8. s.ship = classes.class
9.GROUP BY classes.class;
```

Первое (полное) соединение:

```
1.Classes FULL JOIN
2.Ships ON classes.class = ships.class
```

будет содержать все возможные классы кораблей. Заметим, что здесь можно было ограничиться левым (**LEFT**) соединением, так как согласно связи между таблицами в Ships не может быть корабля, класс которого отсутствует в таблице Classes.

Далее выполняется левое соединение с потопленными кораблями из таблицы Outcomes по следующему предикату (множество s содержит все потопленные корабли):

```
1.ON s.ship = ships.name OR s.ship = classes.class
```

То есть мы включаем в результирующий набор корабль, если имя его совпадает с именем потопленного корабля или если класс совпадает с именем

потопленного корабля. На рассмотренных выше примерах данных этот запрос будет работать правильно, в отличие от первого рассмотренного решения. Действительно, если в классе «Бисмарк» имеется два потопленных корабля, один из которых является головным и отсутствует в Ships, то оба они будут учтены согласно рассмотренному выше предикату. Если же головной корабль присутствует в таблице Ships, то это ничего не меняет, так как предикат все равно будет выполнен.

В чем же здесь ошибка? Ошибка заключается как раз в предикате последнего соединения. Пусть в таблице Ships имеются корабли некоторого класса (например, два корабля с именами A и B класса Class_1), которые не были потоплены. И пусть в Outcomes имеется потопленный головной корабль этого же класса. Тогда соединяться будут следующие два отношения (приводим здесь только важные для анализа атрибуты):

Class	Name
Class_1	A
Class_1	B

и

Ship (отношение s) Class_1

по предикату

s.ship = classes.class

В результате будет получено отношение, содержащее корабли, которые не были потоплены, но учитываются этим решением:

Class	Name	Ship
Class_1	A	Class_1
Class_1	B	Class_1

Можно сказать иначе, а именно, потопленный головной корабль учтен здесь столько раз, сколько кораблей этого класса имеется в таблице Ships (как потопленных, так и нет). Так или иначе, но **COUNT**(ship) = 2, что неверно, так как потоплен был всего один корабль.

Кстати, из сказанного становится очевидным, как исправить данное решение; причем сделать это очень просто. Можно просто добавить 8 символов. Попробуйте.

Решение 3.12.3

```
1.SELECT class, SUM(CASE
2. WHEN result = 'sunk'
3. THEN 1 ELSE 0
4. END)
5.FROM (SELECT c.class, sh.name, o.ship, o.result
6. FROM Classes c LEFT JOIN
7. Ships sh ON c.class = sh.class LEFT JOIN
8. Outcomes o ON ISNULL(sh.name, c.class) = o.ship
9. ) t
10.      GROUP BY class;
```

Оставим без внимания подсчет количества потопленных кораблей. Ошибка не в этом, а в том, как формировался набор записей для этого подсчета.

Итак, левое соединение таблицы Classes с таблицей Ships по столбцу class позволяет нам учесть также классы, которые не имеют кораблей в таблице Ships. Это правильно, так как нам следует выводить данный класс со значением 0 в качестве количества потопленных кораблей, если таковые отсутствуют.

Далее выполняется левое соединение с таблицей Outcomes, которая и содержит информацию о результатах сражений. Предикат соединения использует специфическую для SQL Server функцию **ISNULL**, которая возвращает первый аргумент, если он не является **NULL**-значением, и второй — в противном случае:

```
1.ISNULL(sh.name, c.class) = o.ship
```

То есть имя корабля в таблице Outcomes сравнивается с именем корабля, полученным из таблицы Ships или именем класса, если имя корабля содержит **NULL**-значение. Это значение возникает в предыдущем соединении тогда, когда класс не имеет кораблей в Ships; и только в этом случае!

Опять рассмотрим случай, когда в Ships имеется корабль A некоторого класса Class_1, а в таблице Outcomes содержится как этот корабль, так и головной корабль класса Class_1 (имя совпадает с именем класса). Пусть оба эти корабля были потоплены. Тогда первое соединение даст:

Class_1	A

Второе же соединение будет искать в таблице Outcomes строки, удовлетворяющие вышеприведенному предикату. Такой строкой будет всего одна:

Class_1	A	A

так как будет выполнено сравнение только по имени корабля (A), но не по классу!

Решение 3.12.4

```
1.SELECT class, SUM(sunks) sunks
2.FROM (SELECT cl.class, 1 sunks
3.      FROM Classes cl LEFT JOIN
4.           Ships sh ON cl.class = sh.class INNER JOIN
5.           Outcomes ou ON ou.ship = sh.name OR
6.                         ou.ship = cl.class
7.           WHERE result='sunk'
8.       UNION
9.       SELECT DISTINCT class, 0 sunks
10.            FROM classes
11.          ) tab
```

```
12.      GROUP BY class;
```

В первом из объединяемых запросов предложения FROM для каждого потопленного корабля в результирующий набор попадает строка:

Класс 1

В основном запросе эти «единички» суммируются. Но поскольку **UNION** устраняет дубликаты, то для любого количества потопленных кораблей в классе мы, в конечном итоге получим либо 1 (из этого первого запроса), либо 0 из второго, который учитывает классы, не имеющие потопленных кораблей.

Если же выполнить объединение при помощи **UNION ALL**, то придем к решению 3.12.2, которое содержит аналогичную ошибку.

Решение 3.12.5

```
1.SELECT t1.class, COUNT(*) AS cnt
2.FROM (SELECT a.class, b.name
3.      FROM Classes a LEFT JOIN
4.-- соединение с Ships без головных кораблей:
5.           Ships b ON a.class = b.class AND
6.                      a.class <> b.name
7.      ) AS t1 JOIN
8.-- соединение либо по классу для головных кораблей, либо
  по имени
9.      Outcomes t2 ON t1.class = t2.ship OR
10.                        t1.name = t2.ship
11.      WHERE result = 'sunk'
12.      GROUP BY t1.class
13.      -- Выбираются классы кораблей, которые не попали в
  первый запрос.
14.      -- Это классы, не имеющие потопленных кораблей.
15.      UNION
16.      SELECT class, '0'
17.      FROM Classes
18.      WHERE class NOT IN (SELECT DISTINCT t1.class
19.                          FROM (SELECT a.class, b.name
20.                                FROM Classes a LEFT JOIN
21.                                     Ships b ON a.class
  = b.class AND
```

```
22.                                                              a.class
   <> b.name
23.                                 ) AS t1 JOIN
24.                                 Outcomes t2 ON t1.class
   = t2.ship OR
25.                                                          t1.name =
   t2.ship
26.                          WHERE result = 'sunk'
27.                          );
```

Решение 3.12.6

```
1.SELECT d.class class, (SELECT COUNT(f.result)
2.                       FROM (SELECT c.result
3.                             FROM Ships b LEFT OUTER JOIN
4.                                  Outcomes c ON (b.name
   = c.ship)
5.                             WHERE c.result = 'sunk' AND
6.                                   d.class = b.class
7.                             UNION ALL
8.                             SELECT c.result
9.                             FROM Outcomes c
10.                                   WHERE   c.result   =
   'sunk' AND
11.                                           d.class    =
   c.ship
12.                                 ) f
13.                          ) Sunks
14.      FROM Classes d;
```

Для анализа двух последних решений — 3.12.5 и 3.12.6 — рассмотрим следующие варианты данных. В таблице Ships (показаны только принципиальные для анализа столбцы):

name	class
ship1_class_1	class_1
ship2_class_1	class_1

В таблице Outcomes:

ship	result
ship1_class_1	sunk
class_1	sunk

Тогда согласно предикату соединения в решении 3.12.5

```
1.    ON t1.class = t2.ship OR
2.         t1.name = t2.ship
```

в результирующий набор дважды попадет корабль ship1_class_1 из таблицы Ships, так как для первой строки в таблице Outcomes у него совпадает имя корабля, а для второй — название класса. В результате получим 3 потопленных корабля, хотя на самом деле их только 2.

Решение задачи 3.12.6 даст нам здесь правильный результат, поскольку первый запрос в объединении (соединение по имени корабля) даст только ship1_class_1, а второй — только class_1. Однако это решение тоже не верно, что будет продемонстрировано на другом варианте данных.

В таблице Ships

name	class
ship1_class_2	class_2
class_2	class_2

В таблице Outcomes:

ship	result
ship1_class_2	sunk
class_2	sunk

Первый запрос в объединении даст нам оба потопленных корабля класса class_2, а второй — головной корабль этого класса. Поскольку при

объединении используется **UNION ALL**, то головной корабль дважды будет учтен в результирующем наборе, в результате чего мы опять получаем 3 вместо 2. Косметическое исправление **UNION ALL** на **UNION** не делает решение верным, так как здесь возникает та же ошибка, что и в решении 3.12.4, когда для любого количества потопленных кораблей класса в результат попадает только 1.

Кстати, решение 3.12.5 на этих данных тоже дает значение 3, но по другой причине, описанной выше.

Упражнение 57

Для классов, имеющих потери в виде потопленных кораблей и не менее 3 кораблей в базе данных, вывести имя класса и число потопленных кораблей.

Эта задача в чем-то подобна задаче 56, то есть здесь можно допускать те же ошибки в подсчете потопленных кораблей. Однако ситуация усугубляется еще и определением общего числа кораблей в классе.

Решение 3.13.1

```
1.SELECT c.class, SUM(outc)
2.FROM Classes c LEFT JOIN
3. Ships s ON c.class = s.class LEFT JOIN
4. (SELECT ship, 1 outc
5. FROM Outcomes
6. WHERE result = 'sunk') o ON s.name = o.ship OR
7. c.class = o.ship
8.GROUP BY c.class
9.HAVING COUNT(*) > 2 AND
10.       SUM(outc) IS NOT NULL;
```

Первое левое соединение дает все классы, повторяющиеся столько раз, сколько имеется кораблей в таблице Ships. Если некий класс не имеет кораблей в этой таблице, то он будет упомянут один раз, и это дает нам

возможность учесть головные корабли класса в таблице Outcomes, если таковые имеются.

Далее выполняется еще одно левое соединение с набором потопленных кораблей по предикату

```
1.ON s.name = o.ship OR c.class = o.ship
```

В вычисляемый столбец заносится 1, если имя потопленного корабля совпадает либо с именем корабля, либо с именем класса из полученного ранее набора. Таким образом, мы здесь и пытаемся учесть головные корабли.

Наконец, выполняется группировка по классам с отбором по числу кораблей (строк) класса и подсчитывается сумма потопленных кораблей (единиц в столбце outs). Автор решения предлагает рациональный способ вычислить в одной группировке и общее число кораблей, и количество потопленных кораблей в классе. Предикат:

```
1.SUM(outc) IS NOT NULL
```

в соответствии с условием задачи убирает из результата классы, не имеющие потопленных кораблей.

Те, кто внимательно читал анализ предыдущих задач, уже догадались, в чем дело. Правильно, проблема в предикате второго соединения. Однако не только в этом.

Рассмотрим следующий вариант данных. Пусть для некоторого класса class_N в таблице Ships имеется два корабля: ship_1 и ship_2. Кроме того, в таблице Outcomes есть потопленный корабль ship_1 и оставшийся на плаву головной — class_N.

Первое соединение даст:

Class	Ship
Class_N	ship_1
Class_N	ship_2

Выполняем второе соединение:

Class	ship	outs
Class_N	ship_1	1
Class_N	ship_2	NULL

В результате этот класс вообще не попадет в результирующий набор, так как не будет выполнено условие COUNT(*) > 2, хотя на самом деле корабля 3. Причина ошибки заключается в том, что мы выполняем соединение только по потопленным кораблям, одновременно подсчитывая общее число кораблей.

Давайте теперь немного изменим данные в примере. Пусть и головной корабль class_N тоже потоплен. Тогда результатом соединения будет:

class	ship	outs
class_N	ship_1	1
class_N	ship_2	NULL
class_N	ship_1	1
class_N	ship_2	1

Последние две строки будут получены в результате соединения со строкой потопленного головного корабля, так как предикат c.class = o.ship дает «истину». Таким образом, мы вместо одной строки для головного корабля получаем по строке на каждый корабль класса из таблицы Ships. Итого, вместо

class	outs
class_N	2

имеем

class	outs
class_N	3

Вы можете попытаться исправить это решение или использовать другой подход на базе внутреннего соединения и объединения.

Как это ни покажется удивительным, но ниже приведены три совсем разных решения, которые содержат одну и ту же ошибку, по крайней мере, они возвращают один и тот же результат на проверочной базе сайта.

Решение 3.13.2

```
1.SELECT class, SUM(sunk)
2.FROM (SELECT class, COUNT(*) AS sunk
3. FROM Ships a JOIN
4. Outcomes b ON a.name = b.ship AND
5. a.class <> b.ship
6. WHERE result = 'sunk'
7. GROUP BY class
8. UNION ALL
9. SELECT class, '1'
10.       FROM Classes a JOIN
11.       Outcomes b ON a.class = b.ship
12.       WHERE result = 'sunk'
13.       UNION ALL
14.       SELECT class, '0'
15.       FROM classes
```

```
16.        ) t
17.       -- где классы с числом кораблей больше 2:
18.       WHERE class IN (SELECT t1.class
19.        FROM (SELECT a.class
20.        FROM Classes a LEFT JOIN
21.        Ships b ON a.class = b.class
22.        ) t1 LEFT JOIN (SELECT DISTINCT ship
23.        FROM Outcomes
24.        WHERE ship NOT IN (SELECT name
25.        FROM Ships
26.        )
27.        ) t2 ON t1.class = t2.ship
28.        GROUP BY t1.class
29.        HAVING COUNT(*) > 2
30.        )
31.       GROUP BY class
32.       HAVING SUM(sunk) > 0;
```

Решение 3.13.3

```
1.SELECT a.class AS cls, a.num_sunks AS sunk
2.FROM (SELECT c.class, COUNT (o.ship) AS num_sunks
3. FROM Outcomes o LEFT JOIN
4. Ships s ON o.ship = s.name LEFT JOIN
5. Classes c ON s.class = c.class
6. WHERE o.result = 'sunk'
7. GROUP BY c.class) a,
8. (SELECT c1.class
9. FROM Ships s1, Classes c1
10.        WHERE s1.class = c1.class
11.        GROUP BY c1.class
12.        HAVING COUNT(name) >= 3
13.        ) B
14.       WHERE a.class = b.class;
```

Решение 3.13.4

```
1.SELECT class, COUNT(result) AS sunk
2.FROM (SELECT class, result, name
3. FROM Ships LEFT JOIN
4. Outcomes ON ship=name AND
```

```
5. class IS NOT NULL AND
6. result = 'sunk'
7. ) T
8.GROUP BY class
9.HAVING COUNT(class) > 2 AND
10.       COUNT(result) > 0;
```

Проанализируйте тонкости вышеприведенных решений, самым красивым из которых, безусловно, является 3.13.4. Всего лишь одно соединение, для которого сразу подсчитывается как количество потопленных, так и общее число кораблей в классе. У этих решений имеется общая ошибка, о которой шла речь выше: не учтены головные корабли, которые присутствуют в таблице Outcomes и отсутствуют в таблице Ships.

Упражнение 59

Посчитать остаток денежных средств на каждом пункте приема для базы данных с отчетностью не чаще одного раза в день. Вывод: пункт, остаток.

Решение 2.2.1

```
1.SELECT ss.point, ss.inc - dd.out
2.FROM (SELECT i.point, SUM(inc) AS inc
3. FROM Income_o i
4. GROUP BY i.point
5. ) AS ss,
6. (SELECT o.point, SUM(out) AS out
7. FROM Outcome_o o
8. GROUP BY o.point
9. ) AS dd
10.      WHERE ss.point = dd.point;
```

В предложении **FROM** в каждом из подзапросов определяется сумма соответственно прихода и расхода денежных средств на каждом из пунктов приема. Эти подзапросы соединяются по равенству номеров пунктов приема,

что позволяет построчно вычислить остаток денежных средств на каждом пункте: ss.inc— dd.out.

Казалось бы, все правильно, однако, решение 2.2.1 содержит одну ошибку. Попробуйте ее найти.

Упражнение 60

Посчитать остаток денежных средств на начало дня 15.04.2001 на каждом пункте приема для базы данных с отчетностью не чаще одного раза в день. Вывод: пункт, остаток.

Задача 2.3.1

```
1.SELECT i.point, CASE inc
2.  WHEN NULL
3.  THEN 0
4.  ELSE inc
5.  END -
6.  CASE out
7.  WHEN NULL
8.  THEN 0
9.  ELSE out
10.       END
11.      FROM (SELECT point, SUM(inc) inc
12.       FROM Income_o
13.       WHERE '20010415' > date
14.       GROUP BY point
15.       ) AS I FULL JOIN
16.       (SELECT point, SUM(out) out
17.       FROM Outcome_o
18.       WHERE '20010415' > date
19.       GROUP BY point
20.       ) AS III ON III.point = I.point;
```

Эта задача во многом аналогична предыдущей задаче №59. По сути, здесь дополнительно используется лишь отбор по дате. В связи с этим мы хотим

обратить внимание на ее представление в запросе. Дело в том, что в предикате сравнивается строка с полем типа datetime. В SQL Server имеется функция **CONVERT**, которая позволяет преобразовать строковое представление даты/времени к этому типу, используя различные форматы представления даты. Однако строковое представление в виде год месяц день, которое используется в рассматриваемом решении, всегда будет правильно преобразовываться неявно к типу **datetime** вне зависимости от настроек сервера [5].

Кто разобрался с ошибкой в предыдущей задаче, наверняка, увидел здесь попытку ее исправить. К сожалению, попытку неудачную, что, с другой стороны, дает вам возможность еще раз проверить себя.

База данных «Корабли»

Рассматривается база данных кораблей, участвовавших в морских сражениях второй мировой войны. Имеются следующие отношения:

```
1. Classes (class, type, country, numGuns, bore, displacement)
2. Ships (name, class, launched)
3. Battles (name, date)
4. Outcomes (ship, battle, result)
```

Корабли в «классах» построены по одному и тому же проекту. Классу присваивается либо имя первого корабля, построенного по данному проекту, либо названию класса дается имя проекта, которое в этом случае не совпадает с именем ни одного из кораблей. Корабль, давший название классу, называется головным.

Атрибутами отношения Classes являются имя класса (class), тип (значение bb используется для обозначения боевого или линейного корабля, а bc для боевого крейсера), страну (country), которой принадлежат корабли данного

класса, число главных орудий (numGuns), калибр орудий (bore — диаметр ствола орудия в дюймах) и водоизмещение в тоннах (displacement).

В отношение Ships записывается информация о кораблях: название корабля (name), имя его класса (class) и год спуска на воду (launched).

В отношение Battles включены название (name) и дата битвы (date), в которой участвовал корабль.

Отношение Outcomes используется для хранения информации о результатах участия кораблей в битвах, а именно, имя корабля (ship), название сражения (battle) и чем завершилось сражение для данного корабля (потоплен — sunk, поврежден — damaged или невредим — ok).

***Примечание*:**

В отношение Outcomes могут входить корабли, отсутствующие в отношении Ships.

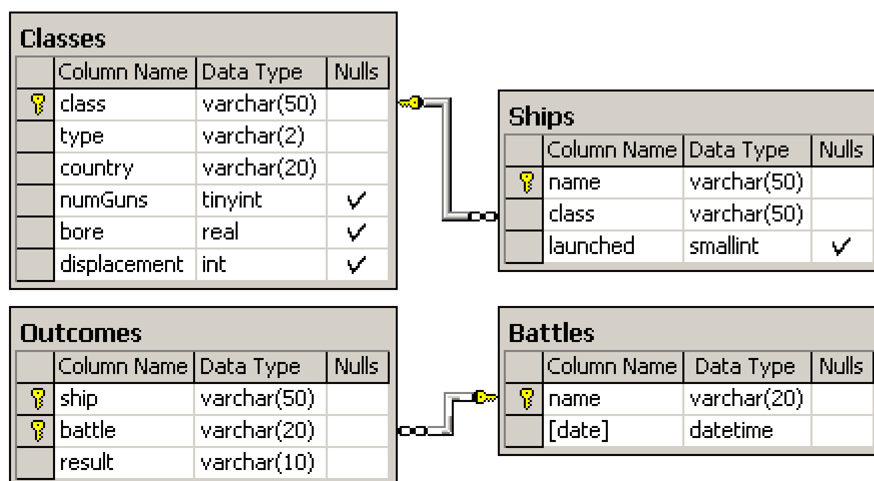


Рис. 3.1. Схема базы данных «Корабли»

Отметим несколько моментов, на которые следует обратить внимание при анализе схемы на рис. 3.1. Таблица Outcomes имеет составной первичный ключ {ship, battle}. Это ограничение не позволит ввести в базу данных дважды один и тот же корабль, принимавший участие в одном и том же сражении. Однако допустимо неоднократное присутствие одного и того же корабля в данной таблице, что означает участие корабля в нескольких битвах. Класс корабля определяется из таблицы Ships, которая имеет внешний ключ (class) к таблице Classes.

Особенностью данной схемы, которая усложняет логику запросов и служит причиной ошибок при решении задач, является то, что таблицы Outcomes и Ships никак не связаны, то есть в таблице результатов сражений могут

находиться корабли, отсутствующие в таблице Ships. На основании этого, казалось бы, можно сделать вывод о том, что для таких кораблей их класс неизвестен, а, следовательно, неизвестны и все технические характеристики. Это не совсем так. Как следует из описания предметной области, имя головного корабля совпадает с именем класса, родоначальником которого он является. Поэтому если имя корабля из таблицы Outcomes совпадает с именем класса в таблице Classes, то однозначно можно сказать, что это головной корабль, и, следовательно, все его характеристики нам известны.

Каждый знает, как улучшить эту «плохую» схему: связать таблицы Ships и Outcomes по имени корабля, при этом столбец ship в Outcomes становится внешним ключом к таблице Ships. Безусловно, это так, однако не следует забывать, что в реальной ситуации не вся информация может быть доступна. Например, имеется архивная информация о кораблях, участвовавших в том или ином сражении, без указания классов этих кораблей. При наличии обсуждаемой связи сначала будет необходимо внести такой корабль в таблицу Ships, при этом столбец class должен допускать NULL-значения.

С другой стороны, что нам мешает ввести головной корабль, который попал в таблицу Outcomes, также и в таблицу Ships? В принципе ничего, так как год спуска на воду не является обязательной информацией. По этому поводу следует заметить, что администратор базы данных и разработчик приложения, как правило, разные люди. Не всегда разработчик приложения и его пользователи имеют права на модификацию данных.

Плохая структура еще не означает, что из нее нельзя извлечь достоверную информацию, чем собственно мы и занимаемся, решая предлагаемые задачи. Что касается учебных целей, то работа с такой структурой даст значительно больше в освоении языка, чем структура «хорошая», так как заставит писать более сложные запросы и научит учитывать дополнительные обстоятельства, накладываемые схемой. Этим видимо и руководствовались авторы этой схемы данных [2]. Кроме того, запросы, написанные для «плохой» схемы, будут давать правильные результаты и после улучшения структуры (хотя и станут менее эффективными), то есть тогда, когда вся информация станет доступной, и мы сможем установить связь между таблицами Ships и Outcomes.

Наконец, стоит обратить внимание на то, что столбец launched в таблице Ships допускает NULL-значения, то есть нам может быть неизвестен год спуска на воду того или иного корабля. То же самое мы можем сказать о кораблях из Outcomes, отсутствующих в Ships.

Что ж, перейдем к решению задач. Заметим лишь, что в настоящей главе мы уже не рассматриваем совсем простые задачи (хотя они имеются и для этой схемы), которых должно было хватить вам в первой главе.

Упражнение 70

Укажите сражения, в которых участвовало, по меньшей мере, три корабля одной и той же страны.

Решение 3.7.1

```
1.SELECT AA.name AS bat
2.FROM (SELECT O.battle AS name, C.country, COUNT(O.ship)
  AS cnt
3. FROM Outcomes O, Ships S, Classes C
4. WHERE O.ship = S.name AND
5. C.class = S.class
6. GROUP BY O.battle, C.country
7. ) AA
8.WHERE AA.cnt >= 3;
```

Можно назвать этот запрос «первым приближением» к решению. Соединяются все необходимые таблицы через предложение **WHERE**, в результате чего определяется битва и страна (из таблицы Classes) для кораблей из таблицы Outcomes. Далее выполняется группировка по стране и сражению с последующим отбором по числу кораблей.

Ошибочным здесь является то, что мы никак не учитываем корабли, отсутствующие в таблице Ships, так как используются внутренние соединения. Читатель уже, наверное, вник в используемую схему и понимает, что здесь нет учитываются головные корабли, класс которых может быть определен не только через таблицу Ships, но и непосредственно с помощью таблицы Classes, а, следовательно, может быть определена и владеющая кораблем страна. Теперь рассмотрим решения, в которых была сделана попытка учесть эту особенность схемы данных.

Решение 3.7.2

```
1.SELECT bat
2.FROM  (SELECT  DISTINCT  d.battle  AS  bat,  a.country,
  COUNT(d.ship) AS s
3.      FROM Outcomes d, Ships b, Classes a
4.      WHERE d.ship = b.name AND
5.            b.class=a.class
6.      GROUP BY d.battle, a.country
7.      UNION
8.       SELECT  DISTINCT  d.battle  AS  bat,  a.country,
  COUNT(d.ship) AS s
9.      FROM Outcomes d, Classes a
10.           WHERE d.ship = a.class AND
11.                 d.ship NOT IN (SELECT name
12.                                FROM Ships
13.                                )
14.           GROUP BY d.battle, a.country
15.           ) AS t1
16.     WHERE s > 2;
```

Ошибка, характерная для начинающих, состоит в том, что сначала выполняется группировка, а потом объединение. И хотя здесь отсутствует ошибка решения 3.7.1 (во втором запросе объединения учтены головные корабли, которых нет в Ships), решение не даст нам страну, у которой в сражении участвовало 3 корабля, два из которых присутствуют в таблице Ships, а один (головной) — только в таблице Outcomes.

Одно время на сайте системой проверки принималось заведомо неправильное решение:

Решение 3.7.3

```
1.SELECT battle
2.FROM Classes c LEFT JOIN
3. Ships s ON c.class = s.class INNER JOIN
4. Outcomes o ON o.ship = s.name OR
5. c.class = o.ship
6.GROUP BY battle, country
7.HAVING COUNT(ship) > 3;
```

Обратите внимание на **HAVING COUNT**(ship) > 3. Использование правильного предиката с условием >=3 делало запрос неверным, каким он и является. Подгонка решения позволила обнаружить огрех в проверке, который и был устранен.

Итак, запрос соединяет классы с кораблями из таблицы Ships, чтобы определить страну корабля. Левое соединение (**LEFT JOIN**) используется для того, чтобы не потерять класс, если кораблей этого класса нет в таблице Ships. Такой (и не только) класс понадобится для того, чтобы учесть головные корабли из таблицы Outcomes, что и делается в следующем (внутреннем) соединении. Предикат этого соединения

```
1.ON o.ship = s.name OR c.class = o.ship
```

сформирует строку, в столбце ship которой будет находиться имя корабля, принимавшего участие в сражениях, если его имя совпадает с именем корабля известного класса в таблице Ships или если его имя совпадает с именем класса (головной корабль). Если корабль не принимал участия в сражении, то значением в столбце ship будет **NULL**. Затем выполняется группировка по паре атрибутов {battle, country} с предложением **HAVING COUNT**(ship) >= 3, что позволяет отобрать только те страны, которые участвовали в битве более чем двумя кораблями. Заметим, что функция **COUNT** корректно обработает **NULL**-значения в столбце ship.

Внимание:

О разнице в использовании COUNT(*) и COUNT(имя столбца) можно почитать в пункте 5.5.

В этом «или» предиката (1) и заключается основная ошибка этого запроса. Если один и тот же головной корабль имеется и в таблице Outcomes, и в таблице Ships, то он будет учтен дважды для одной и той же битвы. Это можно увидеть из следующего запроса:

```
1.SELECT battle, country, ship, COUNT(*) qty
2.FROM Classes c LEFT JOIN
3. Ships s ON c.class = s.class INNER JOIN
4. Outcomes o ON o.ship = s.name OR
5. c.class = o.ship
6.GROUP BY battle, country, ship;
```

Приведем здесь только одну неправильную строку результата:

Battle	country	ship	qty
Surigao Strait	USA	Tennessee	2

Явная ошибка, так как один и тот же корабль не может дважды упоминаться для одной и той же битвы (Surigao Strait), что запрещено первичным ключом на таблице Outcomes.

Отметим, что рассматриваемый запрос, как и решения 3.7.2 и 3.7.1, содержит еще одну ошибку, встречающуюся настолько часто, что она даже описана в FAQ на сайте. Эта ошибка заключается в том, что поскольку группировка выполняется по паре атрибутов {battle, country}, то битва будет выводиться неоднократно, если в ней принимало участие минимум по 3 корабля от каждой участвовавшей в битве страны.

Остается один вопрос. Почему же при трех отмеченных ошибках (>3 вместо >=3, ошибочное соединение и возможное появление дубликатов) запрос принимался системой?

Попробуем разобраться. В основной базе не было ни одной битвы, для которой бы выполнялось условие задачи. Правильное решение показывало пустой набор записей. Поэтому ошибочное увеличение числа кораблей не работало с правильным критерием (>=3), так как запрос выдавал битву Surigao Strait, хотя в ней реально принимало участие 2 корабля из USA. А вот условие >3 опять давало пустой набор.

В проверочной базе для блокировки решения с неисключенными дубликатами для одной битвы было два набора по 3 и более корабля разных стран. При этом в одном наборе головной корабль присутствовал в обеих таблицах (Outcomes и Ships). Для этого набора рассматриваемым запросом ошибочно считалось 4 корабля, а для второго правильно — 3. Поэтому условие в предикате **HAVING** – > 3 и давало только одну битву, разрешая самым неожиданным образом проблему с дубликатами.

Мир полон неожиданностей; чем больше делается ошибок, тем больше вероятность совпадения результатов.

Пустой набор результата решения этой задачи на основной базе неоднократно вызывал нарекания. Поэтому автор, попутно блокируя рассмотренное неверное решение, добавил данных и в основную базу.

Решение 3.7.4

```
1.SELECT DISTINCT battle
2.FROM (SELECT battle, country
3. FROM (SELECT battle, country
4. FROM Outcomes INNER JOIN
5. Classes ON ship = class
6. UNION
7. SELECT battle, country
8. FROM Outcomes o INNER JOIN
9. Ships s ON o.ship = s.name INNER JOIN
10.       Classes c ON s.class = c.class
11.       ) x
12.       GROUP BY battle, country
13.       HAVING COUNT(*) > 2
14.       ) y;
```

Во внутреннем подзапросе объединяются два запроса. В первом из них

```
1.SELECT battle, country
2. FROM Outcomes INNER JOIN
3. Classes ON ship = class;
```

определяется страна и сражения, в которых принимали участие головные корабли из этой страны. Во втором запросе

```
1.SELECT battle, country
2. FROM Outcomes o INNER JOIN
3. Ships s ON o.ship = s.name INNER JOIN
4. Classes c ON s.class = c.class;
```

определяется страна и сражения для тех кораблей, которые имеются в таблице Ships. Соединение с таблицей Classes необходимо, чтобы узнать страну, владеющую кораблем.

Использование для объединения предложения **UNION** устраняет дубликаты. С одной стороны, это кажется правильным, так как головной корабль может находиться как в таблице Outcomes, так и в таблице Ships. С другой стороны, после удаления дубликатов в результирующем наборе останется только одна уникальная пара {сражение, страна}, а это означает, что для любого числа кораблей из одной страны, останется лишь одна строка для каждого из сражений. В результате последующая группировка попросту излишней, как предложение **HAVING**.

Первое, что приходит в голову, — написать **UNION ALL** вместо **UNION**, то есть учесть все дубликаты. Но, как уже понятно из предыдущего обсуждения, тогда для одного головного корабля, участвующего в некотором сражении, мы получим две строки, если этот корабль присутствует еще и в таблице Ships.

Как поступить? Автор предлагает два подхода. При первом подходе мы оставляем **UNION**, но подсчитываем не страны, а корабли. Тогда устранение дубликатов будет правильным. При втором подходе автор предполагает использование **UNION ALL**, но тогда нужно в одном из объединяемых запросов проверять, чтобы учитываемый корабль не присутствовал в другой таблице, тем самым, подсчитывая его один раз.

Какой из способов предпочесть, зависит не только от наших предпочтений, но и от стоимости плана выполнения запроса. Предлагаем самостоятельно оценить планы, предварительно решив задачу двумя описанными способами.

Упражнение 71

Найти тех производителей ПК, все модели ПК которых имеются в таблице PC

Вот типичный неверный запрос

```
1.SELECT DISTINCT maker
2.FROM Product
3.WHERE model IN (SELECT model FROM PC);
```

который сопровождается следующим вопросом: "Производитель E с моделью 1260 присутствует и в таблице PC, а правильный результат его не содержит. Почему?"

Ключевым моментом формулировки является слово "ВСЕ". Давайте посмотрим на модели производителя E. Модели ПК, которые выпускает производитель E, дает следующий запрос:

```
1.SELECT model
2.FROM Product
3.WHERE maker='E' AND type='PC';
```

Результат:

model
1260
2111
2112

А теперь проверим, какие из этих моделей имеются в таблице PC:

```
1.SELECT DISTINCT model
2.FROM PC
3. WHERE model IN(1260, 2111, 2112);
```

Оказывается, что из трех моделей только одна - 1260 - имеется в таблице PC. По условию же задачи там должны находиться ВСЕ три модели производителя E.

Собственно, решение этой задачи сводится к операции реляционного деления, только для каждого производителя у нас свой делитель (его модели). В упрощенном виде операцию реляционного деления можно записать так:

```
1.A(a, b) DIVIDEBY B(b)
```

где делимое (A) представляет собой бинарное (двухатрибутное) отношение, а делитель (B) - унарное. Результатом являются такие значения из первого атрибута отношения A, для каждого из которых значения второго атрибута содержат ВСЕ значения делителя.

Операция реляционного деления не является примитивной. Это означает, что эту операцию можно выразить через другие (примитивные) реляционные операции. Избыточность реляционной алгебры, предложенной Коддом, обусловлена ориентаций на практическое применение. Язык SQL тоже избыточен, в чем нас убеждает каждая задача, которую можно решить разными способами. Несмотря на это, аналога операции реляционного деления в нем нет.

В заключение приведу представление реляционного деления, выраженного через другие операции.

```
1.A DIVIDEBY B :=
2.A[a] EXCEPT ((A[a] TIMES B) EXCEPT A) [a]
```

Здесь A[a] означает проекцию отношения A на атрибут a; TIMES - декартово произведение. "Подстрочный" перевод на язык SQL может выглядеть следующим образом:

```
1.SELECT a FROM A
2.EXCEPT
3.SELECT a FROM (
4.SELECT A.a, B.b FROM A, B
5.EXCEPT
6. SELECT a,b FROM A) X;
```

Не следует использовать этот подстрочник как руководство к действию; есть более простые способы решить задачу. Впрочем, я не настаиваю.

Упражнение 77

Определить дни, когда было выполнено максимальное число рейсов из Ростова ('Rostov'). Вывод: число рейсов, дата.

Мне казалось, что формулировка предельно понятна. Тем более, что когда эта задача еще находилась на втором рейтинговом этапе, она не вызывала никаких вопросов. Однако сказалась разница в классе :) . Отвечать на аналогичные вопросы приходилось столь часто, что мне пришлось написать это объяснение.

Вот типичный пример неправильного запроса:

```
1.SELECT MAX(superden.qty), superden.date
2.  FROM
3.    (SELECT COUNT(den.trip_no) AS qty, den.date
4.       FROM
5.          (SELECT   DISTINCT   trip_no,   date   FROM
  Pass_in_trip) AS den,
6.           Trip WHERE trip.trip_no=den.trip_no AND
7.           trip.town_from='Rostov'
8.       GROUP BY den.date) AS superden
9.    GROUP BY superden.date;
```

Подзапрос

```
1.SELECT DISTINCT trip_no, date FROM Pass_in_trip;
```

определяет совершенные рейсы. DISTINCT здесь вполне уместен, т.к. для пассажиров, летевших в одном самолете, комбинация {trip_no, date} совпадает. Подзапрос соединяется с таблицей Trip, чтобы отобрать только ростовские рейсы: trip.town_from='Rostov'.

Группировка по дате позволяет подсчитать распределение количества ростовских рейсов по дням. Пока все верно, но последний шаг лишён смысла. Зачем еще одна группировка по дате, если все уже сгруппировано, т.е. для каждой даты в результирующем наборе и так есть только одна строка?

Кажется, что таким образом автор решения пытался найти максимум, но получил тот же самый набор. Пусть распределение количества по датам будет таким:

2007-08-19	2
2007-08-20	2
2007-08-21	3

По условию задачи мы должны получить лишь одну строку:

2007-08-21	3

т.к. максимальное число полётов – 3 – достигается 2007-08-21, но в результате последней группировки по дате мы получим те же самые 3 строки.

Надеюсь, что теперь понятно, как следует решать эту задачу, и поддержке сайта не придется больше отвечать на письма по этому поводу.

Упражнение 78

Для каждого сражения определить первый и последний день месяца, в котором оно

состоялось.Вывод: сражение, первый день месяца, последний день месяца.

Характерной ошибкой в этой задаче является неправильное определение високосного года.

Следует иметь в виду, что если год делится нацело на 100, но при этом не делится на 400, то такой год не является високосным. Следовательно, високосным, например, не является 1900 год, в то время как 2000 год – високосный.

База данных «Аэрофлот»

Схема БД состоит из четырех таблиц:

```
1.Company (ID_comp, name)
2.Trip(trip_no, ID_comp, plane, town_from, town_to,
  time_out, time_in)
3.Passenger(ID_psg, name)
4.Pass_in_trip(trip_no, date, ID_psg, place)
```

Таблица Company содержит идентификатор и название компании, осуществляющей перевозку пассажиров. Таблица Trip содержит информацию о рейсах: номер рейса, идентификатор компании, тип самолета, город отправления, город прибытия, время отправления и время прибытия. Таблица Passenger содержит идентификатор и имя пассажира. Таблица Pass_in_trip содержит информацию о полетах: номер рейса, дата вылета (день), идентификатор пассажира и место, на котором он сидел во время полета. При этом следует иметь в виду, что

- рейсы выполняются ежедневно, а длительность полета любого рейса менее суток;

- время и дата учитывается относительно одного часового пояса;

- время отправления и прибытия указывается с точностью до минуты;

- среди пассажиров могут быть однофамильцы (одинаковые значения поля name, например, Bruce Willis);

- номер места в салоне – это число с буквой; число определяет номер ряда, буква (a – d) – место в ряду слева направо в алфавитном порядке;

- связи и ограничения показаны на схеме данных.

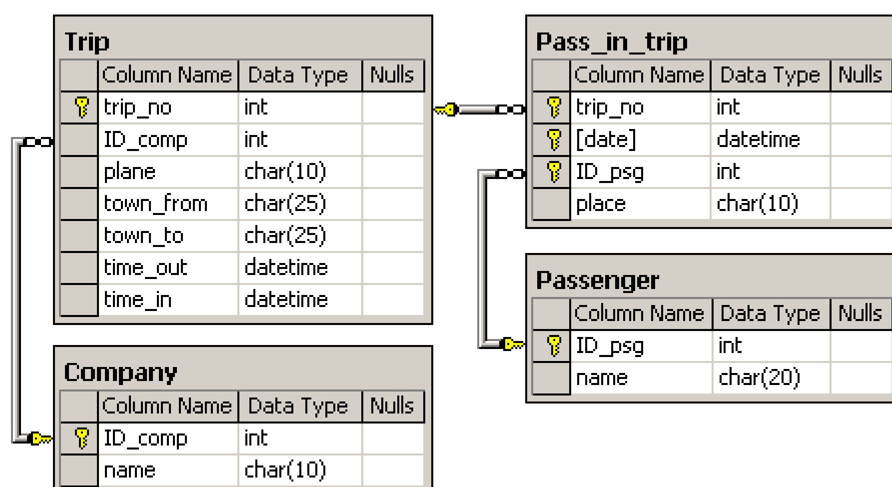


Рис. Схема базы данных «Аэропорт»

Нередко задают такой вопрос: "Почему в таблице Trip днём отправления/прибытия является 1900-01-01?"

В таблице Trip содержится только время отправления/прибытия, поскольку, согласно описанию предметной области, рейсы выполняются ежедневно. Присутствие даты объясняется тем, что в ранних версиях SQL Server не было отдельных типов данных для даты (DATE) и времени (TIME), которые появились только в версии 2008 года. Поэтому использовался тип **DATETIME**, соответствующий стандартному TIMESTAMP, включающему все составляющие метки времени.

Что же касается конкретно даты **1900-01-01**, то эта дата соответствует началу отсчета времени, т.е. нулю. Выполните запрос:

```
1. SELECT CAST(0 AS DATETIME);
```

и вы получите

1900-01-01 00:00:00.000

Т.е. если ввести в столбец типа DATETIME только время, то датой этого значения станет 1900-01-01. В этом можно убедиться, явно приведя значение времени к типу DATETIME, например:

```
1.SELECT CAST('13:44:00' AS DATETIME);
```

1900-01-01 13:44:00.000

Упражнение 93

Для каждой компании, перевозившей пассажиров, подсчитать время, которое провели в полете самолеты с пассажирами. Вывод: название компании, время в минутах.

Проблемы, возникающие при решении этой задачи, можно проиллюстрировать таким сообщением одного из участников. Вот что он пишет:

"Если выполнить запрос

```
1.SELECT Trip.time_out, Trip.time_in
2.       FROM Trip
3.       WHERE Trip.id_comp=2;
```

time_out	time_in
1900-01-01 09:35:00.000	1900-01-01 11:23:00.000
1900-01-01 17:55:00.000	1900-01-01 20:01:00.000

то получается, что компания Аэрофлот (id_comp=2) произвела два полёта, первый продолжительностью в 1 час 48 минут, второй - продолжительностью 2 часа 6 минут. Итоговая продолжительность полётов получается 108 + 126= 234 минуты, а никак не 216 минут, как указано в "правильном результате".

Непонимание вызвано недостаточным изучением схемы БД и её описанием. Таблица Trip представляет собой расписание полетов, которые выполняются ежедневно. А вот в таблице Pass_in_trip содержится информация о полетах с пассажирами. Давайте посмотрим, какие рейсы компании с id_comp=2 были выполнены:

```
1.SELECT pt.trip_no, date, time_out, time_in
2.        FROM pass_in_trip pt
3.        JOIN
4.         (SELECT trip_no,time_out,time_in FROM trip WHERE
  id_comp=2) t
5.                 ON t.trip_no=pt.trip_no
6.        GROUP BY pt.trip_no, date, time_out, time_in;
```

Вот результат вышеприведенного запроса:

trip_no	date	time_out	time_in
1145	2003-04-05 00:00:00.000	1900-01-01 09:35:00.000	1900-01-01 11:23:00.000
1145	2003-04-25 00:00:00.000	1900-01-01 09:35:00.000	1900-01-01 11:23:00.000

Итак, первый рейс был выполнен дважды, а второй - ни разу, т.е. 108*2 = 216.

Упражнение 121

Найдите названия всех тех кораблей из базы данных, о которых можно определенно сказать, что они были спущены на воду до 1941 г.

Несмотря на коэффициент сложности 2, решение этой задачи вызывало, по-видимому, наибольшие затруднения. Причем связано это не со сложностью в построении запроса, а с логикой решения задачи.

Эта задача заменила более простую задачу, которая звучала так: «Найдите названия всех кораблей из базы данных, спущенных на воду до 1918 г». Казалось бы, какая разница? Разве следующее решение не является правильным?

Решение 3.5.1

```
1.SELECT name AS shipName
2.FROM Ships
3.WHERE launched < 1941;
```

Нет. Как справедливо было замечено посетителями сайта, здесь никак не учитываются даты сражений. Действительно, если год спуска на воду корабля неизвестен, и при этом он участвовал в сражении, которое произошло до 1941 года, то такой корабль следует включать в результат. В первоначальной формулировке этой задачи, где фигурировал 1918 год, об этом можно было не заботиться, так как формально база данных содержит информацию о сражениях второй мировой войны, которая началась в 1939 году.

Решение 3.5.2

```
1.SELECT name
2.FROM Ships
3.WHERE launched < 1941
4.UNION
5.SELECT ship
6.FROM Outcomes, Battles
7.WHERE name = battle AND
8.      DATEPART(YEAR, date) < 1941
9.UNION
10.     SELECT ship
11.     FROM Outcomes
12.     WHERE ship IN (SELECT class
13.                    FROM Ships
14.                    WHERE launched < 1941
15.                    );
```

Решение 3.5.2 учитывает:

1. Корабли из таблицы Ships с известным годом спуска на воду до 1941 года.
2. Корабли, которые принимали участие в сражениях до 1941 года (естественно, такие корабли должны были быть спущены на воду до сражения, в котором они принимали участие).
3. Корабли из таблицы Outcomes, имена которых совпадает с именем класса какого-нибудь корабля из Ships, спущенного на воду до 1941 года.

Следует заметить, что возможные дубликаты устраняются использованием объединения посредством **UNION**.

Последний вариант учитывает и те случаи, когда головной корабль принимал участие в сражениях только после 1941 года, так как более ранние сражения учтены предыдущим запросом. Осталось выяснить, зачем это нужно. Ответ на этот вопрос следует искать в «висящих» головных кораблях. Итак, корабль из Outcomes, имя которого совпадает с именем одного из классов (головной корабль), отсутствует в таблице Ships или присутствует, но с неизвестным годом спуска на воду. Пусть в таблице Ships есть корабль того же класса с известным годом спуска на воду. Если этот год окажется меньше 1941, то головной корабль следует включать в результирующий набор наряду с упомянутым кораблем. Это следует из того факта, что головной корабль — это

первый корабль в своем классе и, следовательно, должен был быть спущен на воду не позднее любого другого корабля своего класса.

***Примечание*:**

Во втором из объединяемых запросов решения 3.5.2 используется специфическая для SQL Server функция DATEPART. Она необходима, так как из даты сражения (поле date имеет темпоральный тип данных — datetime) нужно извлечь год сражения; в противном случае предикат.

```
1.date < 1941
```

будет давать сравнение с датой 1905 года, в результате преобразования целого числа к типу дата-время (как число дней прошедших от начала 1900 года, что является точкой отсчета дат для настроек по умолчанию в SQL Server 2000). В этом легко убедиться, если выполнить запрос:

```
1.SELECT CAST(1941 AS DATETIME);
```

результатом которого будет

```
1.1905-04-26 00:00:00.000
```

Чтобы оставаться в рамках стандарта, можно было бы использовать следующий предикат:

```
1.date < '1941'
```

Тогда неявное преобразование типа дало бы нужный результат. Опять проверим:

```
1.SELECT CAST('1941' AS DATETIME);
```

что дает

```
1.1941-01-01 00:00:00.000
```

Естественно, правильным будет и предикат, содержащий полную дату начала 1941 года (1 января):

```
1.date < '19410101'
```

Однако вернемся к нашему решению. Оно неверно. Как было сказано, здесь не учитывается ситуация, когда головной корабль присутствует только в Ships, и год его спуска на воду неизвестен. Учесть эту ситуацию можно, добавив к объединению еще один запрос.

Решение 3.5.3

```
1.SELECT name
2.FROM Ships
3.WHERE launched < 1941
4.UNION
5.SELECT ship
6.FROM Outcomes, Battles
7.WHERE name = battle AND
8.DATEPART(YEAR, date) < 1941
9.UNION
10.      SELECT ship
11.      FROM Outcomes
12.      WHERE ship IN (SELECT class
13.      FROM Ships
14.      WHERE launched < 1941
15.      )
16.      UNION
17.      SELECT name
18.      FROM Ships
```

```
19.      WHERE name IN (SELECT class
20.      FROM Ships
21.      WHERE launched < 1941
22.      );
```

Сколько интересной информации можно извлечь из этой задачи, всего лишь поменяв год!

Однако и это еще не все. Предлагаем самостоятельно найти дополнительные корабли, отвечающие условиям задачи. Проверить себя вы сможете, заглянув в ПиР.

Упражнение 124

Среди пассажиров, которые пользовались услугами не менее двух авиакомпаний, найти тех, кто совершил одинаковое количество полётов самолетами каждой из этих авиакомпаний. Вывести имена таких пассажиров.

Эта задача порождает массу ошибочных решений, которые я разделяю на две группы. К первой группе относятся решения, связанные с неверным прочтением формулировки. Например, пытаются найти двух пассажиров, которые летали бы одинаково двумя или большим числом компаний.

Поясню, что рассматривать следует отдельного пассажира и подсчитываеть число полетов, которое он сделал каждой из авиакомпаний, которыми летал.

Что дальше? Рассмотрим теперь пример из второй группы ошибочных решений:

```
1.SELECT DISTINCT name
2. FROM (SELECT id_psg, id_comp, COUNT(pt.trip_no) AS CNT
3.                FROM pass_in_trip pt JOIN trip t ON
  pt.trip_no=t.trip_no
4.               GROUP BY id_comp,id_psg)a,
5.             (SELECT id_psg, id_comp, COUNT(pt.trip_no) AS
  CNT
6.               FROM pass_in_trip pt JOIN trip t ON
  pt.trip_no=t.trip_no
7.              GROUP BY id_comp,id_psg)b,
8.            passenger p
9. WHERE a.id_psg=b.id_psg AND a.id_comp<>b.id_comp AND
  a.cnt=b.cnt
10.       AND p.id_psg=b.id_psg;
```

Сразу отметим ошибочное DISTINCT name, которое устраняет возможных однофамильцев. Однако не это здесь главное. В предложении FROM соединяются два одинаковых запроса

```
1.SELECT id_psg, id_comp, COUNT(pt.trip_no) AS CNT
2.FROM pass_in_trip pt JOIN trip t ON pt.trip_no=t.trip_no
3.GROUP BY id_comp,id_psg;
```

которые, как и сказано выше, подсчитывают для каждого пассажира число полетов, которое он совершил самолетами каждой из компаний.

Соединяются эти запросы по следующим условиям:

- пассажир тот же самый;
- компании разные;
- число полетов совпадает.

Итак, если пассажир совершил, скажем, компанией Aeroflot 3 полета, и также 3 полета он совершил самолетами компании Don_avia, то такой пассажир удовлетворяет условиям соединения и будет выведен в результатах запроса. Если пассажир летал всего двумя компаниями, то это - правильный результат. А если компании три?

Если в результате рассмотренного выше подзапроса мы получим

Bruce Willis	Don_avia	2
Bruce Willis	Aeroflot	2
Bruce Willis	Dale_avia	1

то пассажир Bruce Willis не отвечает условиям задачи, хотя рассматриваемый запрос выведет его, поскольку в запросе будут соединены первые две строки.

Итак, число полетов пассажира каждой из компаний, которыми он летал, должно находиться в пропорции 1:1:...1.

База данных «Окраска»

Схема базы данных состоит из трех таблиц:

```
1.utQ (Q_ID int, Q_NAME varchar(35));
2.utV (V_ID int, V_NAME varchar(35), V_COLOR char(1));
3.utB (B_Q_ID int, B_V_ID int, B_VOL tinyint, B_DATETIME
  datetime).
```

Таблица **utQ** содержит идентификатор и название квадрата, цвет которого первоначально черный.

Таблица **utV** содержит идентификатор, название и цвет баллончика с краской.

Таблица **utB** содержит информацию об окраске квадрата баллончиком: идентификатор квадрата, идентификатор баллончика, количество краски и время окраски.

При этом следует иметь в виду, что:

- баллончик с краской может быть одного из трех цветов - красный V_COLOR='R', зеленый V_COLOR='G', синий V_COLOR='B' (латинские буквы).
- объем баллончика равен 255 и первоначально он полный;
- цвет квадрата определяется по правилу RGB, т.е. R=0,G=0,B=0 - черный, R=255, G=255, B=255 - белый;
- запись в таблице закрасок utB уменьшает количество краски в баллончике на величину B_VOL и соответственно увеличивает количество краски в квадрате на эту же величину;
- значение 0 < B_VOL <= 255;
- количество краски одного цвета в квадрате не превышает 255, а количество краски в баллончике не может быть меньше нуля;
- время окраски B_DATETIME дано с точностью до секунды, т.е. не содержит миллисекунд.

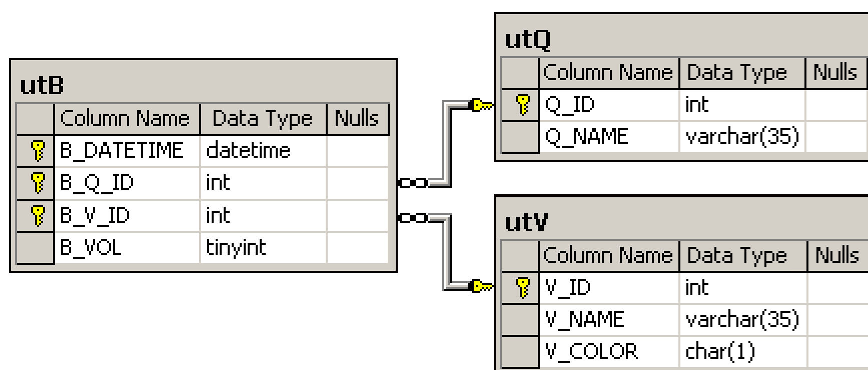


Рис. Схема базы данных "Окраска"

Некоторые пояснения к схеме.

Идентификаторы черных квадратов отсутствуют в таблице utB. Это следует из того, что B_VOL не допускает NULL-значений и строго больше нуля. Таким образом, каждая запись в таблице utB описывает факт окраски одного квадрата одним баллоном, черный же квадрат (R=0,G=0,B=0) не подвергался окраске вовсе.

Ограничения схемы допускают одновременную окраску одного квадрата несколькими баллонами, как и одновременную окраску одним баллоном

нескольких квадратов. Однако запрещена одновременная окраска одного квадрата одним и тем же баллоном.

Ошибки в задачах DML

В этой главе я разберу некоторые ошибки, которые допускаются в задачах DML. Номера задач, как и их формулировки, не приводятся, т.к. пока задачи DML не разделяются на учебные и рейтинговые, ввиду их малочисленности.

Проблема. Требуется определить максимальную скорость имеющихся CD-ROM.

Ошибка состоит в использовании

```
1.SELECT MAX( cd ) FROM ...
```

Дело в том, что скорость CD-ROM хранится в текстовом представлении (например, '12x'). При сравнении текстовых констант '4x' оказывается больше, чем '24x'. А если потребуется посчитать СРЕДНЮЮ скорость?!

Проблема. Требуется округлить среднее значение года спуска на воду кораблей.

Типичной ошибкой здесь является, например, такой прием:

```
1.round(AVG(launched), 0)
```

Дело в том, что среднее значение приводится к типу аргумента. В данном случае оно приводится к целому числу, т.к. столбец launched имеет тип int. При этом SQL Server не округляет результат, а отбрасывает дробную часть. Это легко проверить, выполнив запрос:

```
1. SELECT     AVG(launched)     FROM     (VALUES(9),(10),(10))
   X(launched);
```

Математически 29/3 - это почти 10. Однако получаем 9. В результате функция ROUND здесь совершенно лишняя, т.к. округлять уже нечего.

Так как же здесь следует поступить? Очень просто. Нужно привести аргумент к вещественному типу, по нему считать среднее, а уже затем округлять.

Упражнение 128

Определить лидера по сумме выплат в соревновании между каждой парой пунктов с одинаковыми номерами из двух разных таблиц - outcome и outcome_o - на каждый день, когда осуществлялся прием вторсырья хотя бы на одном из них.
Вывод: Номер пункта, дата, текст:-
"once a day", если сумма выплат больше у фирмы с отчетностью один раз в день;-
"more than once a day", если - у фирмы с отчетностью несколько раз в день;-
"both", если сумма выплат одинакова.

Основная сложность в понимании условия этой задачи заключается в том, как выполнять сравнение, если в некий день в одной таблице есть строка (строки), а в другой - нет.

Как написано в описании предметной области, таблицы с суффиксом "_o" и без него - это разные схемы. Т.е. мы можем считать, что они описывают деятельность разных фирм.

Поскольку сравниваются пункты с одинаковыми номерами, то из результата следует исключить вариант, когда в одной таблице есть

пункт с номером, совсем отсутствующим в другой таблице. Поскольку пункта с таким номером на одной из фирм не существует, то мы не можем сравнивать что-то с тем, чего нет. Например, если в забеге выступал один спортсмен, нельзя утверждать, что он был лучше второго.

Совершенно другую ситуацию мы имеем, когда в определенный день в одной из таблиц строка для некоторого пункта отсутствует, но сам пункт существует (т.е. имеются соответствующие ему записи за другие дни). Такую ситуацию мы можем трактовать так, что пункт просто не работал или работал вхолостую. В этом случае победа аналогичному пункту другой фирмы присуждается закономерно. Возвращаясь к примеру о бегунах, мы можем утверждать, что спортсмен, пришедший к финишу первым был лучше того, который сошёл с дистанции.

Если при проверке на сайте вашего решения вы получили сообщение о большем числе строк, которые вернул ваш запрос, то причина может заключаться как раз в сравнении с несуществующим пунктом.

Упражнение (-2)

Для каждой страны определить год, когда на воду было спущено максимальное количество ее кораблей. В случае, если окажется несколько таких лет, взять минимальный из них. Вывод: страна, количество кораблей, год

Решение 3.6.1. Вот характерное решение для начинающих:

```
1.SELECT country, MAX(x), MIN(launched)
```

```
2.FROM (SELECT country, COUNT(*) x , launched
3. FROM Ships b, Classes a
4. WHERE a.class = b.class
5. GROUP BY country, launched
6. ) s
7.WHERE launched = ANY(SELECT MIN(launched)
8. FROM Ships bb, Classes aa
9. WHERE bb.class = aa.class
10.       GROUP BY country, launched
11.       )
12.      GROUP BY country;
```

Подзапрос в предложении **FROM** определяет количество строк для каждой уникальной пары значений {страна, год спуска на воду}. На языке предметной области это означает, что определяется число кораблей, спущенных на воду каждой страной в каждом году. Пусть результатом выполнения подзапроса s будет следующая таблица:

country	x	launched
Gt.Britain	6	1916
Gt.Britain	1	1917
Japan	1	1913
Japan	2	1914
Japan	2	1915
Japan	1	1916
Japan	1	1941
Japan	1	1942
USA	1	1920

USA	1	1921
USA	3	1941
USA	2	1943
USA	2	1944

Далее (в предложении **WHERE**) отбираются только те строки, у которых год спуска на воду совпадает хотя бы с одним годом, определяемым следующим подзапросом:

```
1.SELECT MIN(launched)
2.FROM Ships bb, Classes aa
3.WHERE bb.class = aa.class
4.GROUP BY country, launched;
```

Что же дает этот подзапрос? Все годы из каждой строки приведенной выше таблицы, так как группировка опять таки выполняется по стране и году. В результате используемый подзапрос никак не ограничивает выборку, и, следовательно, является ненужным. Видимо, подразумевалось удовлетворение условия задачи по минимальному году. Однако речь идет о минимальном годе из тех лет, когда на воду было спущено максимальное количество кораблей данной страны.

Чтобы исправить эту ошибку, недостаточно убрать из группировки столбец launched:

```
1.SELECT MIN(launched)
2.FROM Ships aa, Classes bb
3.WHERE bb.class = aa.class
4.GROUP BY country;
```

в результате чего будут получены минимальные годы по каждой стране:

1916
1913
1920

Тогда строки, удовлетворяющие предикату, будут выглядеть так:

country	x	launched
Gt.Britain	6	1916
Japan	1	1913
Japan	1	1916
USA	1	1920

Таким образом, мы уже потеряли правильные строки для Японии и США. Следует обратить внимание еще и на то, что мы получили строку для Японии:

Japan	1	1916

только на основании того, что год 1916 совпал с минимальным годом для США. Дальнейший код не имеет смысла. Однако и там имеется принципиальная ошибка. В основном запросе

```
1. SELECT country, MAX(x), MIN(launched)
2. …
3. GROUP BY country
```

выполняется группировка по стране с определением двух агрегатных показателей — максимума по количеству кораблей и минимума по году.

Использование в данном случае группировки ошибочно. Действительно, нужные нам строки уже находятся в таблице, получаемой в предложении FROM (например, строка {Gt.Britain, 6, 1916}). Зачем же здесь группировка, когда требуется лишь критерий, который поставит фильтр, отсекающий лишние строки. В результате же группировки образуется одна строка, содержащая статистические показатели для всей группы. При этом максимум и минимум в общем случае достигаются в разных строках группы. Это хорошо видно из таблицы на примере кораблей США, когда минимальному году отвечает далеко не максимальное значение (x=1), а максимальное значение (x=3) достигается совсем в другом году (1941). Поэтому такая группировка может дать правильный результат (в смысле условия задачи) только в том случае, если все значения x для страны совпадают.

Все в этом решении поставлено с ног на голову. Тем не менее, выяснив причины ошибок и заблуждений, попытаемся исправить его без радикальной переработки.

Чтобы все же связать год со страной, можно использовать коррелирующий подзапрос в предложении **WHERE** *(AND aa.country = s.country)*:

```
1.WHERE launched = ANY(SELECT MIN(launched)
2. FROM Ships bb, Classes aa
3. WHERE aa.class = bb.class AND
4. aa.country = s.country
5. GROUP BY country
6. )
```

Это правильно, но пока ничего не меняет, разве что исключит неправильную строку:

Japan	1	1916

Чтобы двигаться дальше, нужно вычислять минимальный год среди лет с максимальным количеством кораблей для каждой страны. Здесь первичным является максимальное количество кораблей. Ведь выбирая лишь

минимальный год, мы можем потерять правильные строки. Поэтому в предикате нужно оценивать не год, а количество кораблей:

```
1.WHERE x >= ALL(SELECT COUNT(launched)
2. FROM Ships bb, Classes aa
3. WHERE bb.class = aa.class AND
4. s.country=aa.country
5. GROUP BY country, launched
6. )
```

Обратите внимание на предикат >= **ALL** — (больше или равно), который дает нам максимальное значение x. Перепишем весь запрос с учетом сказанного о группировке в основном запросе:

Решение 3.6.2

```
1.SELECT country, x, launched
2.FROM (SELECT country, COUNT(*) x , launched
3. FROM Ships b, Classes a
4. WHERE a.class = b.class
5. GROUP BY country, launched
6. ) s
7.WHERE x >= ALL(SELECT COUNT(launched)
8. FROM Ships bb, Classes aa
9. WHERE bb.class = aa.class AND
10.      s.country=aa.country
11.      GROUP BY country, launched
12.      );
```

Все? Не совсем. Если максимум для какой-нибудь страны достигается в разные годы, то мы получим по строке на каждый год. Нам же по условиям задачи требуется в таком случае вывести минимальный год. Как было отмечено выше, это как раз тот самый случай, когда группировка приемлема по смыслу (все значения x для страны одинаковы — максимальны):

Решение 3.6.3. Использование для решения задачи соединения вместо коррелирующего подзапроса.

```
1.SELECT  a.country,  a.numShips  AS  Qty,  MIN(launched)  AS
  Year
2.FROM (SELECT country, COUNT(*) AS numShips, launched
3. FROM Classes INNER JOIN
4. Ships ON Classes.class = Ships.class
5. GROUP BY country, launched
6. ) AS a INNER JOIN
7. (SELECT a.country, MAX(a.numShips) AS Qty
8. FROM (SELECT country, COUNT(*) AS numShips
9. FROM Classes INNER JOIN
10.       Ships ON Classes.class = Ships.class
11.       GROUP BY country, launched
12.       ) AS a
13.       GROUP BY country) AS b
14.       ON a.country = b.country AND a.numShips = b.Qty
15.      GROUP BY a.country, a.numShips;
```

В предложении **FROM** выполняется внутреннее эквисоединение по стране и числу кораблей двух подзапросов. В первом подзапросе определяется страна и число кораблей, спущенных на воду в этой стране в каждом году. Второй подзапрос содержит аналогичный запрос в предложении **FROM**, выбирая из него только ту пару {страна, число кораблей}, которая содержит максимальное число кораблей, спущенное на воду в течение одного года.

В результате этого соединения пара {страна, максимальное число кораблей} дополняется годом, в котором такое число кораблей было спущено на воду. Наконец, выполняется аналогичная решению 3.6.2 группировка, чтобы определить минимальный год, если максимум достигался несколько раз для одной и той же страны.

Решение 3.6.4.

Использование предложения **HAVING**.

```
1.SELECT country, QTY, MIN(launched)
2.FROM (SELECT country, launched, COUNT(name) QTY
3. FROM Classes c JOIN
```

```
4.  Ships s ON c.class = s.class
5.  GROUP BY country,launched
6.  HAVING COUNT(name) = (SELECT MAX(qty)
7.  FROM (SELECT country,launched,COUNT(name) qty
8.  FROM Classes c1 JOIN
9.  Ships s1 ON c1.class = s1.class
10.       WHERE country = c.country
11.       GROUP BY country,launched
12.       )e
13.       )
14.       )T
15.      GROUP BY t.qty, t.country;
```

В подзапросе предложения **FROM** сначала определяются строки {страна, год, число кораблей}. Затем в предикате предложения **HAVING** отбираются только те строки, у которых число кораблей совпадает с максимальным числом кораблей данной страны. Обратите внимание на то, что подзапрос в этом предикате является коррелирующим:

```
1.WHERE country = c.country
```

Именно поэтому **MAX**(qty) относится именно к стране из основного запроса, а не представляет собой глобальный максимум, что было бы в противном случае. Наконец, находится минимальный год для каждого сочетания {страна, максимальное число кораблей}.

Можно переходить к следующей задаче? Нет, еще рано. Все рассмотренные варианты решений содержат одну и ту же ошибку, которую автор предлагает найти самостоятельно.

Упражнение 3 (рейтинговый этап)

Для таблицы Product получить результирующий набор в виде таблицы со

столбцами maker, pc, laptop и printer, в которой для каждого производителя требуется указать, производит он (yes) или нет (no) соответствующий тип продукции. В первом случае (yes) указать в скобках без пробела количество имеющихся в наличии (т.е. находящихся в таблицах PC, Laptop и Printer) различных по номерам моделей соответствующего типа.

Если производитель выпускает модели некоторого типа, но ни одной из них нет в наличии, то, согласно формулировке, результат должен быть

yes(0)

а не

no

Многие находят «ошибку» в тестовом решении. Претензии сводятся к следующему запросу:

```
1. SELECT COUNT(*)
2.    FROM Product
3.    WHERE Maker = 'E' AND type='PC';
```

который дает 3 модели ПК для производителя E, в то время как «правильный ответ» дает только одну модель компьютера для этого производителя. Вернемся к формулировке, в которой сказано:

«...указать в скобках без пробела количество имеющихся в наличии (т.е. находящихся в таблицах PC, Laptop и Printer) различных по номерам моделей соответствующего типа...»

Это в нашем случае означает, что в скобках требуется указать число **различных** моделей ПК производителя E в таблице PC. На языке SQL это можно записать так:

```
1. SELECT COUNT(DISTINCT pc.model)
2.    FROM Product pr
3.    JOIN PC ON pr.model=pc.model
4.    WHERE Maker = 'E';
```

т.е.

1

Подсказки и решения

Упражнение 2 (подсказки и решения)

```
1. SELECT DISTINCT maker
2. FROM Product
3. WHERE type = 'Printer';
```

Упражнение 6 (подсказки и решения)

Соединение таблиц (а здесь необходимо внутреннее соединение) можно выполнить двумя способами:

1. Через предложение WHERE (единственная возможность до появления стандарта SQL-92)

```
1. SELECT DISTINCT Product.maker, Laptop.speed
2. FROM Product, Laptop
3. WHERE Product.model = Laptop.model
4.  AND Laptop.hd >= 10;
```

2. С помощью явной операции соединения JOIN

```
1.SELECT DISTINCT Product.maker,Laptop.speed
2.FROM Product JOIN
3. Laptop ON Product.model = Laptop.model
4.WHERE Laptop.hd >= 10;
```

Хотя оптимизатор SQL Server построит одинаковый план выполнения для обоих запросов, предпочтительным является второй вариант, который позволяет отделить условия соединения таблиц от условий фильтрации строк.

Упражнение 7 (подсказки и решения)

Вот решение, которое использует соединение вместо объединения:

```
1.SELECT DISTINCT a.model,
2. ISNULL(b.price, 0)+ISNULL(c.price, 0)+ISNULL(d.price,
  0) price
3.FROM (((Product a LEFT JOIN
4. PC b ON a.model = b.model
5. ) LEFT JOIN
6. Laptop c ON a.model = c.model
7. ) LEFT JOIN
8. Printer d ON a.model = d.model
9. )
10.      WHERE a.maker = 'B';
```

Здесь применяется три внешних (левых) соединения таблицы Product с каждой из продукционных таблиц.

Отсутствующие значения цены будут заполнены **NULL**-значениями. Например, для модели 1232 персонального компьютера цена блокнота и принтера будут **NULL**. Поэтому только один из трех ценовых столбцов будет содержать значение для каждой строки результирующей выборки. Чтобы не определять, какой это столбец, в списке столбцов предложения **SELECT** используется конструкция

```
1. ISNULL(b.price, 0)+ISNULL(c.price, 0)+ISNULL(d.price, 0),
```

складывающая все три цены, заменяя предварительно NULL-значение нулем. Последнее необходимо, так как сложение с NULL-значением даст NULL. Использование в запросе нестандартной функции **ISNULL**(price, 0) не принципиально, так как не менее эффективно всю конструкцию можно заменить стандартным **COALESCE**, даже без суммирования:

```
1. COALESCE(b.price, c.price, d.price, 0)
```

И все же представленное решение имеет один недостаток. Представьте, что у производителя B есть модель, скажем 1133, которой нет в продукционной таблице. Тогда результатом выполнения запроса будет строка:

1133	0

с ценой $0. Такой результат дает неправильную информацию, так как продукции по такой цене нет. Чтобы согласовать данное решение с решением на основе объединения, которое не выводит строки с нулевой ценой, нужно добавить еще и условие отбора по цене. Сделайте это самостоятельно и проверьте правильность своего решения.

Упражнение 8 (подсказки и решения)

Сначала пара «естественных» решений, которые отличаются лишь предикатом, проверяющим отсутствие у поставщика модели портативного компьютера.

Решение 4.4.1. Предикат NOT IN

```
1. SELECT DISTINCT maker
2. FROM Product
3. WHERE type = 'PC' AND
4.       maker NOT IN (SELECT maker
5.                     FROM Product
6.                     WHERE type = 'Laptop'
7.                     ) ;
```

Решение 4.4.2. Предикат EXISTS (что обычно для этого предиката, подзапрос является коррелирующим)

```
1. SELECT DISTINCT maker
2. FROM Product AS pc_product
3. WHERE type = 'pc' AND
4.       NOT EXISTS (SELECT maker
5.                   FROM Product
6.                   WHERE type = 'laptop' AND
7.                         maker = pc_product.maker
8.                   );
```

3. Теперь приведу несколько оригинальных решений.

Решение 4.4.3. Использование коррелирующих запросов с группировкой

```
1. SELECT DISTINCT maker
2. FROM Product AS p
3. WHERE (SELECT COUNT(1)
4.        FROM Product pt
```

```
5.        WHERE pt.type = 'PC' AND
6.              pt.maker = p.maker
7.        ) > 0 AND
8.       (SELECT COUNT(1)
9.        FROM Product pt
10.             WHERE pt.type = 'Laptop' AND
11.                   pt.maker = p.maker
12.             ) = 0;
```

В подзапросах проверяется, что число моделей ПК поставщика из основного запроса больше нуля, в то время как число моделей портативных компьютеров этого же поставщика равно нулю.

Следует обратить внимание на аргумент функции **COUNT**(1). Стандарт определяет два типа аргументов этой функции: «*» и выражение. Применение «*» приводит к подсчету числа строк, отвечающих запросу. Использование выражения дает число строк, для которых выражение имеет значение, то есть не является **NULL**. В качестве выражения обычно служит имя столбца, поэтому задействование константы может вызвать удивление у тех, кто еще недостаточно хорошо знаком с языком. Поскольку константа (в рассматриваемом запросе 1) не может быть **NULL**, то такое выражение вполне эквивалентно COUNT(*).

На примере этой несложной задачи можно продемонстрировать многообразие решений, которое обусловлено гибкостью языка SQL.

Решение 4.4.4. Внешнее самосоединение

```
1.SELECT DISTINCT p.maker
2.FROM Product p LEFT JOIN
3. Product p1 ON p.maker = p1.maker AND
4. p1.type = 'Laptop'
5.WHERE p.type = 'PC' AND
6.  p1.maker IS NULL;
```

Левое соединение таблицы Product с собой при условии, что производитель один и тот же, а тип продукции из второй таблицы есть блокнот. Тогда в столбце p1.maker будет находиться **NULL**, если у поставщика нет моделей портативных компьютеров, что и используется в предикате

предложения **WHERE** наряду с условием, что в той же строке типом продукции является ПК.

Решение 4.4.5. Группировка

```
1.SELECT maker
2.FROM (SELECT DISTINCT maker, type
3. FROM Product
4. WHERE type IN ('PC', 'Laptop')
5. ) AS a
6.GROUP BY maker
7.HAVING COUNT(*) = 1 AND
8.  MAX(type) = 'PC';
```

В подзапросе выбираются уникальные пары {поставщик, тип}, если типом является ПК или портативный компьютер. Затем выполняется группировка по поставщику, при этом сгруппированные строки должны отвечать следующим условиям:

COUNT(*) = 1 — то есть поставщик должен выпускать только один тип продукции из оставшихся (поскольку мы уже отсекли принтеры, то остается либо ПК, либо портативный компьютер);

MAX(type) = 'PC' — этим типом продукции является ПК. Поскольку в предложении **HAVING** не могут присутствовать ссылки на столбцы без агрегатных функций, то используется **MAX**(type), хотя с тем же успехом можно было написать и MIN(type).

При таком обилии подходов естественен вопрос об эффективности, то есть какой из представленных запросов будет выполняться быстрее. Лидером здесь, как по числу операций, так и по оценке времени исполнения, является решение 4.4.5. Наихудшие показатели у третьего варианта. Остальные примерно в два раза по времени уступают лидеру.

Примечание:

Оценку времени, а также процедурный план выполнения запроса в текстовом представлении можно получить в Query Analyzer (SQL Server), выполнив сначала команду. SET SHOWPLAN_ALL ON; а затем выполняя интересующие нас запросы. Чтобы вернуться к обычному режиму выполнения запросов, нужно в том же подключении дать команду SET SHOWPLAN_ALL OFF;

Если у вас не установлен SQL Server, вы можете получить план выполнения запроса непосредственно на сайте: http://www.sql-ex.ru/perfcon.php.

Упражнение 10 (подсказки и решения)

Решить задачу без использования подзапроса можно. Правда, для этого используются нестандартные средства. Метод основывается на конструкции **TOP** N (SQL Server), которая позволяет отобрать из отсортированного набора первые N строк. Аналогичные конструкции имеются в диалектах SQL многих СУБД. В комитете по разработке стандартов было даже зафиксировано предложение о включение подобной конструкции в стандарт языка. Так что не исключено, что к моменту выхода этой книги в свет данная конструкция уже будет стандартизована. А вот и решение:

```
1.SELECT TOP 1 WITH TIES model, price
2.FROM Printer
3.ORDER BY price DESC;
```

Итак, выполняется сортировка по убыванию цены. В результирующий набор попадает одна (первая — TOP 1) строка. Однако остается проблема, когда несколько принтеров из таблицы будут иметь одинаковую максимальную цену. Проблема решается при помощи предложения **WITH TIES**, которое включит в результирующий набор не только N строк (в нашем случае одну), но и все ниже идущие строки, у которых значения в полях сортировки (у нас — price) совпадают со значениями N-ой строки (у нас — первой).

В **PostgreSQL/MySQL** для ограничения количества строк, возвращаемых запросом, используется конструкция (следующая после ORDER BY)

```
1. LIMIT N [OFFSET M]
```

где
N - число первых строк, возвращаемых запросом в порядке, заданном сортировкой;
M - число строк, пропускаемых перед началом вывода.

Если предложение **OFFSET** отсутствует, то выводиться будут N строк, начиная с первой, в противном случае - N строк, начиная с M+1.

Конструкций, подобных WITH TIES, в этих СУБД нет. Поэтому для решения рассматриваемой задачи способом через сортировку так или иначе придется использовать подзапрос:

```
1. SELECT model, price
2.     FROM Printer WHERE price =
3.       (SELECT price FROM Printer ORDER BY price DESC LIMIT
   1);
```

Заметим, что в отличие от MySQL, в PostgreSQL предложение **OFFSET** может использоваться и при отсутствии предложения LIMIT. В этом случае возвращаться будут все строки запроса кроме первых M. Так, например, чтобы вывести все строки кроме первой строки с максимальной ценой, можно написать:

```
1. SELECT model, price
2.     FROM Printer
3.     ORDER BY price DESC OFFSET 1;
```

model	price
1434	290.00

1433	270.00
1408	270.00
1401	150.00

При использовании SQL Server эту же задачу можно решить так:

```
1.SELECT model, price
2.FROM Printer WHERE code NOT IN(
3.SELECT TOP 1 code
4.FROM Printer
5.ORDER BY price DESC);
```

Т.е. мы выбираем все строки, кроме той, которая идёт первой при сортировке цены по убыванию.

А теперь попробуйте решить эту задачу под MySQL.

При решении последней задачи следует обратить внимание на неточность ее постановки. В результате чего представленное решение на одних и тех же данных может давать разные результаты при разных запусках. В частности, у нас имеется две строки с максимальной ценой, и какая из них будет выводиться зависит от порядка, в котором СУБД будет извлекать строки. А этот порядок может меняться в зависимости от выбранного оптимизатором плана. Чтобы сделать постановку и результат! однозначными, следует указать в условии задачи однозначный порядок сортировки. Эту однозначность всегда обеспечит включение первичного ключа в конец списка столбцов сортировки, например:

```
1.ORDER BY price DESC, code
```

или

```
1.ORDER BY price DESC, model, code
```

Упражнение 11 (подсказки и решения)

Наличие стандартной агрегатной функции **AVG** решает все проблемы:

```
1. SELECT AVG(speed)
2. FROM PC;
```

Однако на форуме сайта были замечены попытки выделиться из общей «серой» массы. Вот вполне эквивалентное решение, которое, правда, добавляет лишнюю операцию в процедурный план:

```
1. SELECT SUM(speed)/COUNT(speed)
2. FROM PC;
```

Справедливость следующего решения зависит от имеющихся ограничений:

```
1. SELECT SUM(speed)/SUM(1)
2. FROM PC;
```

А именно, если **SUM**(speed) игнорирует строки с **NULL**-значением скорости, то **SUM**(1) подсчитает общее количество строк в таблице PC, что, по существу, эквивалентно использованию **COUNT**(*). В результате в числителе будет подсчитана сумма скоростей всех ПК за исключением строк с неизвестной скоростью, которая будет делиться на общее число строк. Поэтому все приведенные решения будут эквивалентны, только если **NULL**-значения недопустимы, то есть имеется ограничение **NOT NULL** на столбце speed.

Упражнение 15 (подсказки и решения)

Неверное решение 1.11.1 этой задачи легко поправить, если различать ПК не по номеру модели, а, как и положено, по первичному ключу — столбцу code:

```
1.SELECT DISTINCT t.hd
2.FROM PC t
3.WHERE EXISTS (SELECT *
4.                FROM PC
5.                WHERE pc.hd = t.hd AND
6.                      pc.code <> t.code
7.                );
```

Поскольку достаточно лишь двух ПК с одинаковыми жесткими дисками, можно использовать самосоединение таблицы PC по аналогичным условиям:

```
1.SELECT DISTINCT pc1.hd
2.FROM PC pc1, PC pc2
3.WHERE pc1.hd = pc2.hd AND
4.      pc1.code <> pc2.code;
```

Однако наиболее оптимальным будет решение с группировкой и условиями отбора в предложении **HAVING**:

```
1.SELECT
2.PC.hd FROM PC
3.GROUP BY hd
4.HAVING COUNT(hd) > 1;
```

Для полноты картины приведем еще одно решение, которое использует подзапрос с группировкой и которое по эффективности также уступает вышеприведенному.

```
1. SELECT DISTINCT hd
2. FROM PC
3. WHERE (SELECT COUNT(hd)
4.         FROM PC pc2
5.         WHERE pc2.hd = pc.hd
6.         ) > 1;
```

Причина низкой эффективности рассмотренных решений с подзапросами заключается в том, что все они используют коррелирующий подзапрос, то есть подзапрос, который будет выполняться для каждой строки основного запроса. Запрос с соединением имеет самую низкую производительность. Это вполне понятно, так как операции соединения весьма затратные, несмотря на достаточно эффективные алгоритмы их реализации.

Упражнение 16 (подсказки и решения)

Избыточность решения 1.12.2 можно устранить, если вообще убрать подзапрос, а соединение выполнить между таблицами P и L. При этом запрос получится не только компактным, но и легко читаемым и, что не менее важно, более эффективным.

Еще один пример тяжело читаемого запроса, который был бы правильным, будь номер модели (*model*) числовым.

```
1. SELECT MAX(model1), MIN(model2), MAX(speed), MAX(ram)
```

```
2.FROM (SELECT pc1.model AS model1, pc2.model AS model2,
  pc1.speed, pc2.ram,
3.              CASE WHEN CAST(pc1.model AS NUMERIC(6,2)) >
4.                        CAST(pc2.model AS NUMERIC(6,2))
5.                   THEN pc1.model+pc2.model
6.                   ELSE pc2.model+pc1.model
7.              END AS sm
8.       FROM PC pc1, PC pc2
9.       WHERE pc1.speed = pc2.speed AND
10.                 pc1.ram = pc2.ram AND
11.                 pc1.model <> pc2.model
12.           ) a
13.      GROUP BY a.sm;
```

Однако тип данных VARCHAR(50) подразумевает наличие произвольных символов, наличие которых имеет место в проверочной базе данных (скажем, T-64). На этих данных преобразование типа

```
1.CAST(pc1.model AS NUMERIC(6,2))
```

будет вызывать ошибку.

Я считаю, что это хороший пример того, как не следует писать запросы. Как же следует их писать? Загляните на форум задачи, когда её решите, там вы найдете лучшие образцы.

Упражнение 17 (подсказки и решения)

Итак, избавляемся от декартового произведения. Для этого убираем таблицу PC из предложения **FROM**, а таблицы Product и Laptop соединяем по столбцу model:

```
1.SELECT DISTINCT type, Laptop.model, speed
2.FROM Laptop, Product
3.WHERE Product.model = Laptop.model AND
4.       Laptop.speed < (SELECT MIN(speed) FROM PC);
```

Условие *p.type = 'laptop'* уже излишне, так как внутреннее соединение будет содержать модели только данного типа. Оказывается, что можно избавиться и от соединения, поскольку таблица Product используется только для того, чтобы в результирующем наборе вывести тип продукта. Но тип продукта заранее известен — это портативный компьютер, поэтому мы может просто применить выражение (константу) для указания типа, убрав соединение:

```
1.SELECT DISTINCT 'Laptop' AS type, model, speed
2.FROM Laptop
3.WHERE speed < (SELECT MIN(speed) FROM PC);
```

Заметим, что данное решение будет справедливо только в том случае, если в таблице Laptop будут находиться изделия с типом Laptop. Для нашей базы данных это условие выполняется, т. к. имеется всего три типа продукции и соответственно три таблицы. Поэтому нарушение данного условия может быть связано только с изменением структуры, что, впрочем, тоже следует иметь в виду при разработке приложений со встроенными SQL-запросами.

Упражнение 18 (подсказки и решения)

В следующем решении:

```
1.SELECT c.maker, a.priceA price
2.FROM (SELECT MIN(price) priceA
3. FROM Printer
4. WHERE Color ='y'
5. ) a INNER JOIN
```

```
6.  Printer b ON a.priceA = b.price INNER JOIN
7.  Product c ON b.model = c.model;
```

в подзапросе определяются минимальная цена на цветные принтеры, затем выполняется соединение по этой цене с таблицей принтеров, чтобы найти все принтеры с такой ценой. Наконец, соединение с таблицей Product дает производителей найденных принтеров.

Конечно, соединение по цене можно заменить простым сравнением

```
1.WHERE price = (SELECT MIN(price) priceA
2. FROM Printer
3. WHERE Color ='y'
4. )
```

Однако ошибка не в этом, а в том, что отыскиваются любые принтеры, у которых цена совпадает с минимальной ценой на цветные принтеры, что, собственно говоря, и имеет место. В результате получаем:

Maker	Price
D	270.0
A	270.0

Правильный же ответ дает только одну строку:

Maker	Price
D	270.0

Вторая строка не является цветным принтером, в этом легко убедиться, если добавить в предложение SELECT дополнительные столбцы:

```
1. SELECT c.maker, a.priceA price, color, b.type
```

что дает

maker	price	color	type
D	270.0	y	Jet
A	270.0	n	Matrix

Вторая ошибка заключается в возможном наличии дубликатов, т. к. для одного и того же производителя может иметься несколько принтеров по одной и той же цене.

Упражнение 23 (подсказки и решения)

Для решения этой задачи используется обычно два подхода: соединение с последующим выбором нужной строки и проверка на попадание производителя в два списка. Следующее неправильное решение, реализует первый подход:

Решение 4.11.1

```
1. SELECT DISTINCT a.maker
2. FROM Product a LEFT JOIN
3.       PC b ON a.model = b.model AND
```

```
4.                b.speed> =750 LEFT JOIN
5.        Laptop c ON a.model = c.model  AND c.speed> =750
6.WHERE NOT (b.model IS NULL AND
7.              c.model IS NULL
8.              );
```

Это еще один вариант на тему «чего-то одного». Действительно, модель уникальна, т. е. она представляет собой либо ПК, либо портативный компьютер. В результате внешних соединений получаются строки типа:

```
1.maker model (PC) NULL
```

или

```
1.maker NULL model (laptop)
```

Однако здесь не может быть строк:

```
1.maker model (PC) model (laptop)
```

поскольку соединение выполняется по номеру модели.

В результате в списке имеются производители, которые выпускают только один вид продукции с требуемыми характеристиками. Однако подправить это решение просто, добавив группировку по производителю и посчитав модели. Предлагаем вам сделать это самостоятельно.

Решение 4.11.2

Для демонстрации второго подхода рассмотрим следующий вариант:

```
1.SELECT DISTINCT t.maker
2.FROM Product t
3.WHERE (t.model IN (SELECT model
4.                   FROM PC
5.                   WHERE speed >= 750
6.                   ) OR
7.        t.model IN (SELECT model
```

```
8.                     FROM Laptop
9.                     WHERE speed >= 750
10.                          )
11.              ) AND
12.              EXISTS (SELECT *
13.                      FROM Product
14.                      WHERE Product.maker = t.maker AND
15.                            Product.type='PC'
16.                      ) AND
17.              EXISTS (SELECT *
18.                      FROM Product
19.                      WHERE Product.maker = t.maker AND
20.                      Product.type='Laptop'
21.                      );
```

который можно прочитать следующим образом: найти производителя, который выпускает ПК со скоростью не менее 750 МГц или портативный компьютер со скоростью не менее 750 МГц; при этом данный производитель должен производить ПК и портативные компьютеры. Несомненный прогресс этого решения по сравнению с решением 1.15.1 состоит в том, что выводятся производители, как ПК, так и портативных компьютеров. Однако это решение допускает вариант, когда производитель выпускает только ПК со скоростью более 750 МГц, в то время как все его портативные компьютеры имеют скорость менее 750 МГц, и наоборот.

Упражнение 26 (подсказки и решения)

Ниже приводится еще пара неверных решений этой задачи, которые содержат легко исправимую ошибку.

Решение 4.12.1

```
1.SELECT AVG(price)
2.FROM (SELECT price
3. FROM PC
```

```
4. WHERE model IN (SELECT model
5. FROM product
6. WHERE maker='A' AND
7. type='PC'
8. )
9. UNION
10.       SELECT price
11.       FROM Laptop
12.       WHERE model IN (SELECT model
13.       FROM product
14.       WHERE maker='A' AND
15.       type='Laptop'
16.       )
17.       ) AS prod;
```

Решение 4.12.2

```
1.SELECT AVG(price)
2.FROM (SELECT price, model
3. FROM pc
4. WHERE model IN (SELECT model
5. FROM product
6. WHERE maker='A' AND
7. type='PC'
8. )
9. UNION
10.       SELECT price, model
11.       FROM Laptop
12.       WHERE model IN (SELECT model
13.       FROM product
14.       WHERE maker='A' AND
15.       type='Laptop'
16.       )
17.       ) AS prod;
```

Первое из этих решений дает на основной базе результат 772.5, а второе — 773.0 при правильном значении 734.5454545454545.

В запросе 4.12.1 выбираются цены на все модели производителя A из таблицы PC. Затем они объединяются с ценами на все модели производителя A из таблицы Laptop. Наконец, вычисляется среднее значение. Что же тут

неправильного? Ошибка состоит в том, как объединяются цены. Оператор **UNION** исключает дубликаты, поэтому из нескольких одинаковых цен (если таковые имеются) будет оставаться только одна. Как результат, среднее будет посчитано по неверному количеству.

В запросе 4.12.2 выбирается не только цена, но и номер модели. То есть объединение выполняется по паре атрибутов. Это решение было бы правильным, если бы в соответствующей таблице не было одинаковых моделей с одинаковыми ценами. Последнее было бы гарантировано, если бы пара атрибутов {price, model} являлась первичным ключом. Однако согласно нашей схеме это не так. Сама по себе такая ситуация не является нереальной. Представим себе, что одна модель комплектуется большим диском, чем другая модель с тем же номером, а памяти, наоборот, имеет меньше. Тогда цены у них вполне может быть одинаковы. Естественно, может быть несколько идентичных моделей.

В результате объединения будут исключены дубликаты пар {price, model} и, как следствие, получен неверный результат.

Надеемся, теперь вполне очевидно, как следует решать эту задачу.

Упражнение 27 (подсказки и решения)

Смотрите упражнение номер 2

Упражнение 30 (подсказки и решения)

Чтобы решить проблему удвоения, утроения результатов, достаточно сначала выполнить соответствующую группировку по каждой таблице, а уже затем их соединять. Тогда эта задача сводится

к задаче 29, решение которой не вызывало проблем и потому не приводится в книге.

Это не единственный способ решить задачу. Загляните на форум сайта, чтобы познакомиться с другими вариантами.

Упражнение 46 (подсказки и решения)

Обещанный в пункте 3.1 пример автоматического соединения двух решений, каждое из которых правильно учитывает один из двух моментов, вызывающих характерные ошибки, может выглядеть так (смотрите комментарии)

```
1. -- Корабли, участвующие в битве при Гвадалканале и которые
   есть в Ships
2. -- Обратите внимание на использование коррелирующего
   подзапроса в
3. -- предложении WHERE, который решает проблему устранения
   дубликатов при
4. -- декартовом произведения
5. SELECT a.ship, displacement, numGuns
6. FROM (SELECT ship
7.  FROM Outcomes
8.  WHERE battle = 'Guadalcanal'
9.  ) AS a, Classes
10.      WHERE class IN (SELECT class
11.       FROM Ships
12.       WHERE name = a.ship
13.       )
14.      UNION
15.      -- Аналогичный по логике запрос, который выбирает
   те головные корабли из
16.      -- Outcomes, которые сражались при Гвадалканале.
17.      SELECT a.ship, displacement, numGuns
18.      FROM (SELECT ship
19.       FROM Outcomes
20.       WHERE battle = 'Guadalcanal'
21.       ) AS a, Classes
22.      WHERE class IN (SELECT ship
```

```
23.         FROM Outcomes
24.         WHERE ship = a.ship
25.         )
26.        UNION
27.        --По сути, это решение 3.1.1
28.        SELECT a.ship, displacement, numGuns
29.        FROM (SELECT ship
30.         FROM Outcomes
31.         WHERE battle = 'Guadalcanal'
32.         ) AS a LEFT JOIN
33.         Classes ON a.ship = class;
```

В результате получим лишние строки, характерным примером которых являются такие:

ship	displacement	numGuns
California	32000	12
California	NULL	NULL

Можно еще утяжелить этот запрос (и сделать его менее эффективным), добавив код для исключения ошибочной строки. Критерием здесь может служить присутствие **NULL**-значения, например, в столбце displacement, если есть другая строка с тем же именем корабля. Однако мы советуем обойтись без этого и решить задачу иначе. Тем более, что это возможно, в чем легко убедиться, зайдя на форум сайта, посвященный этой задаче.

В заключение приведем почти правильное решение:

```
1.SELECT name, displacement, numGuns
2.FROM Classes, Ships
3.WHERE Classes.class = Ships.class AND
4. name IN (SELECT Ship
5. FROM Outcomes
6. WHERE battle = 'Guadalcanal'
```

```
7. )
8.UNION
9.SELECT class, displacement, numGuns
10.      FROM Classes
11.      WHERE class IN(SELECT ship
12.       FROM Outcomes
13.       WHERE battle = 'Guadalcanal'
14.       );
```

Первый запрос из объединения в этом решении находит информацию о кораблях, которые есть в таблице Ships и которые принимали участие в сражение при Гвадалканале.

Второй запрос находит нужные нам головные корабли в Outcomes. Возможные дубликаты (когда головной корабль имеется также и в таблице Ships) исключаются использованием предложения **UNION**.

Так что же здесь неверно? Если до сих пор непонятно, вернитесь к обсуждению задачи в пункте 3.1.

Упражнение 37 (подсказки и решения)

Рассмотрим следующее решение задачи, которое свободно от ошибок, проанализированных в пункте 3.2:

```
1.SELECT t1.class
2.FROM (SELECT a.class AS class, COUNT(b.name) AS coun
3.      FROM Classes a LEFT JOIN
4.           Ships b ON b.class = a.class
5.      GROUP BY a.class
6.      UNION ALL
7.      SELECT a1.class AS class, COUNT(ship) AS coun
8.      FROM Classes a1 LEFT JOIN
9.           Outcomes d ON d.ship = a1.class
10.           WHERE d.ship NOT IN (SELECT b.name
11.                                FROM Ships b
```

```
12.                                          )
13.               GROUP BY a1.class
14.               ) t1
15.         GROUP BY t1.class
16.         HAVING SUM(t1.coun) = 1;
```

Действительно, в подзапросе объединяются два запроса, первый из которых подсчитывает для каждого класса корабли из таблицы Ships, а второй подсчитывает только те головные корабли, которых нет в Ships. Затем основной запрос для каждого класса эти количества суммирует и оставляет только те классы, в которых содержится только один корабль.

Обратите внимание на использование **UNION ALL**. Это необходимо, т. к. в противном случае будут устранены дубликаты пары {класс, количество кораблей}, в результате чего будет выводиться класс, для которого имеется один неголовной корабль в Ships и головной в Outcomes. Как раз эта характерная ошибка отмечалась нами в пункте 3.2.

Что же осталось исправить, раз это решение не принимается системой? Причина состоит в том, что головной корабль класса может принимать участие в нескольких сражениях, и тогда второй из объединяемых запросов столько раз учтет один и тот же головной корабль, сколько раз тот участвовал в сражениях.

Упражнение 39 (подсказки и решения)

Рассмотрим решение, которое учитывает даты сражений, но все же является не вполне верным:

```
1.SELECT t.name
2.FROM (SELECT o.ship AS name, battle
3.      FROM Outcomes o
4.      ) t, Battles b
5.WHERE t.battle = b.name
6.GROUP BY t.name
7.HAVING (SELECT result
```

```
8.         FROM Outcomes, Battles
9.         WHERE ship = t.name AND
10.                    battle = name AND
11.                    date = MIN(b.date)
12.              ) = 'damaged' AND
13.             (SELECT result
14.              FROM Outcomes, Battles
15.              WHERE ship = t.name AND
16.                    battle = name AND
17.                    date = MAX(b.date)
18.              ) IN ('damaged', 'ok', 'sunk') AND
19.              COUNT(t.name) > 1;
```

В этом решении участвовавшие в сражениях корабли группируются по именам, после чего остаются только те которые отвечают следующим условиям:

- в сражении с минимальной датой корабль должен быть поврежден;
- в сражении с максимальной датой результат сражения может быть любым;
- число сражений должно быть больше одного.

Логическая ошибка, допущенная в этом запросе, заключается в том, что если корабль участвовал более чем в двух сражениях, то в первом своем сражении (сражении с минимальной датой) корабль может и не быть поврежден. Более точно, результат его сражения должен быть ok, чтобы представленное выше решение перестало давать правильный результат. Действительно, потоплен корабль быть не может, иначе бы он не участвовал в более поздних сражениях. Если бы он был поврежден, то запрос его бы справедливо учитывал. А вот если последовательность результатов будет следующей: ok, damaged и любой из трех возможных, то представленное решение его бы не выводило, хотя такой корабль и отвечает условиям задачи.

Упражнение 51 (подсказки и решения)

Здесь мы хотим привести одно интересное решение, которое использует только соединения:

```
1.SELECT DISTINCT CASE
2. WHEN sh.name IS NOT NULL
3. THEN sh.name
4. ELSE ou.ship
5. END name
6.FROM Ships sh FULL JOIN
7. Outcomes ou ON ou.ship = sh.name LEFT OUTER JOIN
8. Classes cl ON ou.ship = cl.class OR
9. sh.class = cl.class LEFT OUTER JOIN
10.       Classes cl2 ON cl.displacement = cl2.displacement
  AND
11.       cl.numGuns < cl2.numGuns
12.      WHERE cl.class IS NOT NULL AND
13.       cl2.class IS NULL;
```

Достаточно сложная логика этого решения будет, безусловно, полезна для обучения на данном этапе. Начнем, как обычно, с предложения **FROM**. Полное соединение (**FULL JOIN**) дает нам все корабли из базы данных. При этом возможны следующие варианты:

- корабль присутствует только в таблице Ships;
- корабль присутствует в обеих таблицах;
- корабль присутствует только в таблице Outcomes.

Этим трем случаям соответствуют следующие строки результирующего набора (показаны только значимые в данном случае столбцы):

Ship	name
NULL	ship_1
ship_2	ship_2
ship_3	NULL

Затем выполняется левое соединение с таблицей классов по предикату:

```
1.ou.ship = cl.class OR
2.sh.class = cl.class
```

То есть либо имя корабля из Outcomes должно совпадать с именем класса (висящие головные корабли), либо класс корабля из таблицы Ships. Результат соединения можно проиллюстрировать следующей таблицей:

ship	name	sh.class	cl.class
NULL	ship_1	class_1	class_1
ship_2	ship_2	class_1	class_1
ship_3	NULL	NULL	NULL
class_1	NULL	class_1	class_1

Третья строка таблицы соответствует случаю, когда класс корабля неизвестен (он не является головным!), а четвертая описывает случай головного корабля, отсутствующего в таблице Ships. Забегая немного вперед, заметим, что третья строка не может нам дать никакой информации о водоизмещении и числе орудий такого корабля, поэтому она отфильтровывается следующим предикатом в предложении **WHERE** рассматриваемого решения:

```
1.cl.class IS NOT NULL
```

Наконец, последнее левое соединение опять выполняется с таблицей классов, но уже по другому предикату:

```
1.cl.displacement = cl2.displacement AND
2.cl.numGuns < cl2.numGuns
```

Первое условие предиката очевидно — мы соединяем строки по равенству значений водоизмещения, так как нам нужно находить максимум в группе кораблей, имеющих одинаковое водоизмещение. Чтобы понять смысл второго условия, опять обратимся к примеру. Дополним нашу таблицу необходимыми столбцами и рассмотрим результат соединения по данному предикату на примере, скажем, первой строки предыдущей таблицы при следующих значениях числа орудий у классов кораблей водоизмещением 30 000 тонн:

class_1	16
class_2	10
class_3	14

ship	Name	sh.class	cl.class	displacement	cl2.class	cl.numGuns	cl2.numGuns
NULL	ship_1	class_1	class_1	30000	NULL	16	NULL
NULL	ship_2	class_2	class_2	30000	class_1	10	16
NULL	ship_2	class_2	class_2	30000	class_3	10	14
NULL	ship_3	class_3	class_3	30000	class_1	14	16

Таким образом, корабли класса, имеющего максимальное число орудий в свой группе по водоизмещению, будут иметь NULL-значение в столбцах, относящихся к таблице cl2 (при левом соединении!), в том числе в столбце cl2.class, так как предикат не удовлетворяется. Именно этот критерий и используется в предложении **WHERE** для отбора записей, отвечающих условиям задачи (*cl2.class IS NULL*).

Наконец, оператор **CASE** формирует имя корабля в окончательном результирующем наборе, выбирая из двух вариантов — столбцы ship или name — тот, в котором находится не **NULL**-значение.

Если изложенное выше вам не вполне понятно, почитайте о внешних соединениях в главе 5 (пункт 5.6.1).

Интересное решение, но учитывающее не все возможные варианты данных. Оно не будет давать верный результат, если имеется класс, число орудий на кораблях которого, нам не известно. Обратите внимание на схему данных: столбец numGuns допускает NULL-значения! Предлагаем вам самостоятельно проанализировать причину ошибки и исправить рассмотренное решение.

Упражнение 53 (подсказки и решения)

Вот решение, в котором сделана попытка выполнить округление:

```
1. SELECT  CAST(ROUND(AVG(numguns),  2)  AS  DECIMAL(10,2))
   Avg_numGuns
2. FROM Classes
3. WHERE type = 'bb';
```

Попытка неудачная, так как округление, как и последующее приведение к типу **DECIMAL** (впрочем, совершенно излишнее в данном контексте), уже применяется к значению целого типа. В результате получить удается не более чем два нуля после десятичной точки.

Проблема заключается в том, что **AVG**(numguns) применяется к аргументу целого типа, поэтому и результат приводится к целому, причем дробная часть не округляется, а отбрасывается, что является особенностью SQL Server, и не является общим правилом.

Упражнение 54 (подсказки и решения)

Чтобы написать **UNION** в решении 3.10.2, недостаточно выполнять объединение по одному столбцу numGuns. Список столбцов должен быть таким, чтобы он однозначно определял корабль. Тогда действительно исключаться будут дубликаты, а не полезная информация.

Ниже представлено решение, которое использует этот прием, но содержит незначительную ошибку, поиск которой предоставляем читателям.

```
1.SELECT CAST(AVG(numguns*1.0) AS NUMERIC (6,2))
2.FROM (SELECT ship, type, numguns
3.      FROM Outcomes RIGHT JOIN
4.           Classes ON ship = class
5.      UNION
6.      SELECT name, type, numguns
7.      FROM Ships s JOIN
8.           Classes c ON c.class = s.class
9.      ) AS al_sh
10.     WHERE type = 'bb';
```

Упражнение 55 (подсказки и решения)

Здесь мы приводим решение, которое верно (хотя и излишне) учитывает корабли из таблицы Outcomes.

```
1.SELECT t1.class, MIN(b.launched)
2.FROM (SELECT name, class
3.      FROM Ships
4.      UNION
5.      SELECT ship, class
6.      FROM Outcomes JOIN
```

```
7.              Classes ON ship = class AND
8.                          ship NOT IN(SELECT name
9.                                      FROM Ships
10.                                           )
11.              ) t1 LEFT JOIN
12.              Ships b ON t1.class = b.class
13.        GROUP BY t1.class;
```

Единственная ошибка этого решения заключается в том, что не учтена ситуация, когда имеется класс, для которого нет кораблей в текущем состоянии базы данных. Заметим, что это допускается схемой, имеющей связь «один-ко-многим» между таблицами Classes и Ships.

Упражнение 56 (подсказки и решения)

Ниже приведено решение, в котором правильно проведен учет потопленных кораблей. Незначительную ошибку в этом решении предлагаем найти самостоятельно. В случае затруднения вернитесь к решению 3.12.3.

```
1.SELECT class, SUM(r) sunks
2.FROM (SELECT name, class, CASE
3. WHEN result = 'sunk'
4. THEN 1 ELSE 0
5. END r
6. FROM Ships AS s LEFT JOIN
7. Outcomes AS o ON o.ship = s.name
8. UNION
9. SELECT ship, class, CASE
10.       WHEN result = 'sunk'
11.       THEN 1 ELSE 0
12.       END r
13.       FROM Classes c JOIN
14.       (SELECT *
15.       FROM Outcomes
16.       WHERE NOT Ship IN (SELECT name
```

```
17.        FROM Ships)
18.        ) AS ot ON ot.ship = c.class
19.        ) AS b GROUP BY class;
```

Упражнение 57 (подсказки и решения)

Ниже представлено решение, в котором правильно определяется число потопленных кораблей (хотя, на наш взгляд, весьма громоздко), однако неправильно подсчитывается общее число кораблей в классе.

```
1. SELECT f.class, SUM(count_out) AS cnt
2. FROM (SELECT t.class, SUM(cnt) AS count_out
3.       FROM (SELECT c.class, ship, COUNT(*) CNT
4.             FROM Classes c LEFT JOIN
5.                  Ships s ON c.class = s.class INNER JOIN
6.                  Outcomes o ON o.ship = s.name AND
7.                                result = 'sunk'
8.             GROUP BY c.class,ship
9.             ) AS t
10.            GROUP BY t.class
11.            UNION ALL
12.            SELECT t.class, SUM(cnt) AS count_out
13.            FROM (SELECT c.class, ship, COUNT(*) cnt
14.                  FROM Classes c INNER JOIN
15.                       Outcomes o ON c.class = o.ship AND
16.                                     o.result = 'sunk' AND
17.                                     NOT EXISTS (SELECT *
18.                                                 FROM Ships
19.                                                  WHERE  o.ship  =
   name
20.                                                 )
21.                  GROUP BY c.class,ship
22.                  ) AS t
23.            GROUP BY t.class
24.            ) AS f
25.        GROUP BY f.class
26.        HAVING 2 < (SELECT SUM(cnt)
27.                     FROM (SELECT COUNT(c.class) AS cnt
28.                           FROM Classes c, Ships s
29.                           WHERE c.class = s.class AND
30.                                 c.class = f.class
```

```
31.                         UNION
32.                         SELECT COUNT(c.class) AS cnt
33.                         FROM Classes c, Outcomes o
34.                         WHERE c.class = o.ship AND
35.                               c.class = f.class AND
36.                               NOT EXISTS (SELECT *
37.                                           FROM Ships
38.                                           WHERE o.ship = name
39.                                           )
40.                         ) AS k
41.                   );
```

Подсчет общего числа кораблей в классе выполняется здесь в предложении **HAVING** основного запроса. В подзапросе этого предложения для каждого класса из основного запроса выполняется объединение числа кораблей из таблицы Ships с числом кораблей (головных) из таблицы Outcomes при условии, что такие корабли не были учтены ранее (их нет в таблице Ships).

Видно, что поскольку объединяются одноатрибутные отношения посредством оператора **UNION**, то если у нас имеется по одному кораблю и в одном, и другом наборе, мы заведомо получаем неверный результат в результате устранения дубликатов. Однако здесь это как бы не должно являться ошибкой, так как мы отбираем классы, имеющие в сумме более двух кораблей. А других возможных вариантов быть не должно, поскольку головной корабль если и есть, то он только один (несмотря на излишнее использование **COUNT** во втором запросе). И все же ошибка кроется именно здесь. Дело в том, что если головной корабль принимал участие более чем в одном сражении, то мы его учитываем по числу сражений, разумеется, если его нет в таблице Ships.

Исправить это решение несложно, предлагаем вам сделать это самостоятельно. Однако можно написать и более простой (а также более эффективный) запрос.

Упражнение 59 (подсказки и решения)

В решении 2.2.1 не учитывается тот факт, что возможна ситуация, когда на некотором пункте

приема может быть только приход, но не быть расхода, то есть в таблице Outcome_o может не быть ни одной строки, относящейся к данному пункту приема. С точки зрения предметной области эта ситуация возможна для вновь открытого пункта приема, когда факт приема денежных средств уже зафиксирован, а факт расхода — еще нет. Тогда выражение в предложении SELECT

```
1.ss.inc - dd.out
```

для такой точки приема станет эквивалентно выражению

```
1.ss.inc - NULL
```

что даст в результате NULL, а не ss.inc, как это должно быть по условиям задачи.

Исправить решение очень легко, переписав ошибочное выражение в виде

```
1.(COALESCE (ss.inc, 0) - COALESCE (dd.out, 0) )
```

что соответствует стандарту, или с помощью функции SQL Server **ISNULL**:

```
1.(ISNULL(ss.inc, 0) - ISNULL(dd.out, 0) )
```

Упражнение 60 (подсказки и решения)

В решении 2.3.1 используется полное внешнее соединение (**FULL JOIN**) подзапросов, чтобы учесть возможные варианты, когда в результате выполнения этих подзапросов для какого-нибудь пункта приема либо сумма прихода, либо сумма расхода будет **NULL**-значением (другими словами, не было расхода и/или прихода). Если, скажем, полученный приход составляет 1000, а расход — 800, то будут учтены все возможные варианты:

Приход	Расход
1000	800
NULL	800
1000	NULL

Варианта NULL NULL быть не может, так как это бы означало, что пункта приема просто не существовало (на данный момент времени).

В предложении **SELECT** используется конструкция, которая должна заменить **NULL** нулем в выражении вычисления остатка. Логика совершенно правильная, однако конструкция применена неверно:

```
1.CASE inc
2.     WHEN NULL
3.     THEN 0
4.     ELSE inc
5. END
```

Ошибка заключается в том, что здесь фактически задействована простая операция сравнения с NULL-значением, а именно,

```
1.CASE
2.     WHEN inc = NULL
3.     THEN 0
4.     ELSE inc
5. END
```

Сравнение же с NULL-значением всегда дает UNKNOWN. Поэтому условие **WHEN** не выполняется, в результате чего выполняется ветвь **ELSE**, всегда возвращая значение inc, даже в том случае, когда inc есть NULL.

Упражнение 70 (подсказки и решения)

Здесь мы хотим рассмотреть вполне прозрачное решение, содержащее две незначительные ошибки. Вот оно (см. комментарии):

```
1.SELECT q.battle
2.FROM (
3.--Определяем корабли из Ships участвовавшие в битвах
4. SELECT Outcomes.battle, Outcomes.ship, Classes.country
5. FROM Classes INNER JOIN
6. Ships ON Classes.class = Ships.class INNER JOIN
7. Outcomes ON Ships.name = Outcomes.ship
8. UNION
9.--Определяем головные корабли из Outcomes
10.       SELECT       Outcomes.battle,       Outcomes.ship,
  Classes.country
11.       FROM Outcomes INNER JOIN
12.       Classes ON Outcomes.ship = Classes.class
13.       ) AS q
14.      GROUP BY q.battle
15.      HAVING COUNT(q.country) >= 3;
```

Надеемся, что вы без труда их найдете и исправите.

Упражнение 121 (подсказки и решения)

Беда с этими головными кораблями! Вот как иногда неправильно решается вопрос о головных кораблях, спущенных на воду ранее 1941 года:

```
1.SELECT class
2.FROM Classes
3.WHERE EXISTS (SELECT 1
4. FROM Ships
5. WHERE launched < 1941 AND
6. Ships.class = Classes.class
7. );
```

То есть здесь класс отождествляется с наличием головного корабля в БД, а именно, разыскивается класс, который имеет в БД корабль, спущенный на воду ранее 1941 года. Однако из описания предметной области следует, что не всегда есть корабль, имя которого совпадает с именем класса. Поэтому искать неучтенные в Ships головные корабли следует исключительно в таблице Outcomes.

Наконец, о неучтенном варианте в решении 5.3.3. Итак, возможна следующая ситуация. Имеется головной корабль с неизвестным годом спуска на воду. Более того, он может участвовать только в сражениях после 1941 года. Пусть для всех других корабли того же класса год спуска на воду тоже неизвестен (это допускается схемой данных). Однако, если хотя бы один из этих кораблей участвовал в сражении до 1941 года, то нам следует такой корабль включить в результирующий набор вместе с головным, так как головной корабль (если он есть!) должен быть спущен на воду ранее любого другого корабля аналогичного класса.

Вот решение, которое, казалось бы, учитывает все оговоренные моменты:

```
1.-- Корабли, спущенные на воду до 1941 года
2.SELECT name
3.FROM Ships
4.WHERE launched < 1941
5.
6.UNION
7.-- Корабли, принимавшие участие в сражениях до 1941 года
8.SELECT ship
9.FROM Outcomes JOIN
```

```
10.        Battles ON Battles.name = Outcomes.battle
11.       WHERE date < '19410101'
12.
13.       UNION
14.       -- Головные корабли из Outcomes, в классе которых
  есть другие корабли,
15.       -- спущенные на воду до 1941 года
16.       SELECT ship
17.       FROM Outcomes
18.       WHERE ship IN (SELECT class
19.        FROM Ships
20.        WHERE launched < 1941
21.        )
22.
23.       UNION
24.       -- Головные корабли из Outcomes при условии, что
  хотя бы один из кораблей
25.       -- того же класса, участвовал в сражении до 1941
  года
26.       SELECT ship
27.       FROM Outcomes
28.       WHERE Ship IN (SELECT class
29.        FROM Ships JOIN
30.        Outcomes ON Ships.name = Outcomes.ship JOIN
31.        Battles ON Battles.name = Outcomes.battle
32.        WHERE date < '19410101'
33.        );
```

Однако система все равно сообщает об ошибке…

Как уже было отмечено в пункте 3.5, головные корабли с неизвестным годом спуска на воду могут находиться не только в таблице Outcomes, но и в таблице Ships. Более того, такие корабли не будут учтены рассматриваемым запросом, если их нет в таблице Outcomes, то есть они либо не участвовали в сражениях, либо информация об их участии неизвестна.

Так что здесь чистая логика, и никаких подвохов.

Упражнение (-2) (подсказки и решения)

При решении этой задачи часто допускается весьма характерная и, на наш взгляд, принципиальная ошибка, которая имеется и в решении 3.6.2. Однако, чтобы не приводить здесь окончательный вариант и позволить читателю самому его построить, объясним суть ошибки, упростив формулировку задачи:

Определить год, когда на воду было спущено максимальное количество кораблей. Вывод: количество кораблей, год

Определить распределение количества кораблей по годам можно так:

```
1. SELECT launched [year], COUNT(*) cnt
2. FROM Ships
3. GROUP BY launched;
```

Примечание:

В SQL Server имена столбцов заключаются в квадратные скобки ([]), чтобы избежать неоднозначности. Например, неоднозначность возникает при использовании пробелов в именах, когда первое слово может быть истолковано как имя столбца, а второе — как его псевдоним (alias). Хорошим стилем признается отказ от пробелов в именах, однако, вполне оправданным является их употребление для формирования заголовков отчета.

В нашем случае ([year]) квадратные скобки применяются во избежание путаницы с функцией year(), которая возвращает год из аргумента, представленного типом дата-время.

Теперь нам нужно оставить из всех строк, возвращаемых этим запросом, только те, у которых количество (cnt) максимально, то есть:

```
1. cnt >= ALL(SELECT COUNT(*) cnt
```

```
2.  FROM Ships
3.  GROUP BY launched
4.  )
```

Окончательно получим:

Решение 4.19.1

```
1.SELECT *
2.FROM (SELECT launched [year], COUNT(*) cnt
3.  FROM Ships
4.  GROUP BY launched
5.  ) x
6.WHERE cnt >= ALL(SELECT COUNT(*) cnt
7.  FROM Ships
8.  GROUP BY launched
9.  );
```

Тем не менее, здесь кроется ошибка. Эта ошибка не связана с формальным построением решения. Оно не вызывает сомнения. Как это обычно происходит при решении задач на сайте, ошибка заключается в неточном учете особенностей модели предметной области, а именно, ее ограничений. В данном случае допускается, что в базе данных могут быть корабли с неизвестным годом спуска на воду, так как, во-первых, столбец launched допускает **NULL**-значения и, во-вторых, для головного корабля, который присутствует только в таблице Outcomes, год спуска на воду неизвестен.

Строить корабли — это вам не кроликов разводить. Корабли строятся годами. Поэтому, если для ряда кораблей год спуска на воду неизвестен (**NULL**), то велика вероятность того, что число таких кораблей будет больше, чем количество кораблей, спущенных на воду в любом реальном году. Особенность группировки заключается в том (и это оговорено в стандарте), что **NULL**-значения трактуются как равные. Следовательно, все корабли с неизвестным годом спуска на воду, будут просуммированы с годом **NULL**. Полагаем, что результат не должен включать такую строку по той причине, что неизвестный год не означает один и тот же. С этим можно, конечно, поспорить. Однако все споры сведутся к допустимости использования специфического значения **NULL** в реляционной модели. Дискуссии по этому поводу ведутся со времен создания этой модели Коддом Е.Ф., которому и принадлежит идея **NULL**-значения. Однако, насколько нам известно, достойной альтернативы предложено не было.

Возвращаясь к нашей задаче, мы, в знак безграничного уважения к Кодду, внесем в решение следующее изменение:

Решение 4.19.2

```
1.SELECT * FROM (SELECT launched [year], COUNT(*) cnt
2.               FROM  Ships
3.               WHERE launched IS NOT NULL
4.               GROUP BY launched
5.               ) x
6.WHERE cnt >= ALL(SELECT COUNT(*) cnt
7.                 FROM  Ships
8.                 WHERE launched IS NOT NULL
9.                 GROUP BY launched
10.                      );
```

В подзапросе предложения WHERE проверку на NULL-значения можно не выполнять, если использовать вместо функции COUNT(*) функцию COUNT(launched), поскольку в этом случае будут подсчитаны только корабли с известным годом спуска на воду:

```
1.WHERE cnt >= ALL(SELECT COUNT(launched) cnt
2.                 FROM Ships
3.                 GROUP BY launched
4.                 )
```

Для всех же кораблей с неизвестным годом спуска на воду будет получена строка со значением 0, так как если в наборе нет ни одной записи, то функция COUNT возвращает именно это значение. Последнее не должно нас смущать, поскольку количество кораблей в основном запросе больше нуля, если есть хотя бы один корабль с известным годом спуска на воду. Аналогичным образом можно поступить и в основном запросе, что позволит получить более краткую форму решения:

Решение 4.19.3

```
1. SELECT * FROM (SELECT launched [year], COUNT(launched)
   cnt
2.                FROM Ships
3.                GROUP BY launched
4.                ) x
5. WHERE cnt >= ALL(SELECT COUNT(launched) cnt
6.                  FROM Ships
7.                  GROUP BY launched
8.                  );
```

Справедливости ради следует отметить, что стоимость решения 4.19.3 по его плану выполнения в SQL Server 2000 будет незначительно уступать (в третьей значащей цифре) стоимости решения 4.19.2.

Примечание:

Стоимость любого запроса к учебным базам данных, а также процедурный план его выполнения можно посмотреть на странице сайта.

Язык манипуляции данными в SQL

…язык учителя дзэн передает идеи, а не чувства или намерения. И поэтому он играет не ту роль, какую обычно играет язык; поскольку выбор фраз исходит от учителя, то чудо свершается в той области, которая ему присуща, и ученик раскрывается сам себе, понимает себя, и таким образом обычная фраза становится ключом.
Х. Кортасар. Игра в классики.

Данная книга ориентирована на практическое применение языка **SQL**, то есть в первую очередь на использование оператора **SELECT**, реализующего выборку данных из реляционных СУБД, и операторов **INSERT**, **UPDATE** и **DELETE**, которые служат для модификации данных. Эти операторы и составляют тот подъязык **SQL**, который называется языком манипуляции данными (или **DML** — Data

Manipulation Language). В этой «теоретической» части мы старались придерживаться тех синтаксических конструкций рассматриваемых операторов, которые, во-первых, полностью соответствуют стандарту **SQL**-92, и, во-вторых, поддерживаются практически всеми коммерческими СУБД. Естественно, нельзя было обойти вниманием и некоторые особенности реализации. Это обусловлено тем, что имеющиеся в каждой реализации расширения стандарта позволяют зачастую более компактно написать запрос, а также тем, что некоторые моменты не стандартизируются и, как правило, опускаются на уровень реализации. Поскольку **сайт** работает с MS SQL Server 2005, то особенности реализации, которые оговариваются отдельно, относятся по большей части именно к этому продукту. В частности, это касается функций работы со строками и значениями типа даты/времени, которые рассматриваются в **главе 7** (**часть III**). Следует отметить, что практически все рассматриваемые здесь задачи, составляющие первый обучающий этап тестирования на сайте, можно решить, пользуясь исключительно стандартными средствами. Это могут подтвердить многочисленные посетители сайта, которые в своей профессиональной деятельности используют различные СУБД от FoxPro до Oracle.

Описание синтаксиса операторов языка сопровождается многочисленными примерами запросов, которые адресуются к учебным базам данных, краткое описание которых приводится в **Приложении*1**. Кроме того, многие параграфы сопровождаются перечнем упражнений, рекомендуемых к решению для закрепления соответствующего материала. Формулировки всех заданий, которые составляют первый этап тестирования на сайте и которые выборочно разбираются в книге, приведены в **Приложении*2**.

Простой оператор SELECT

Оператор **SELECT** осуществляет выборку из базы данных и имеет наиболее сложную структуру среди всех операторов языка SQL. Практически любой пользователь баз данных в состоянии написать простейший оператор **SELECT** типа

```
1. SELECT * FROM PC;
```

который осуществляет выборку всех записей из объекта БД табличного типа с именем PC. При этом столбцы и строки результирующего набора не упорядочены. Чтобы упорядочить поля результирующего набора, их следует перечислить через запятую в нужном порядке после слова SELECT:

```
1.SELECT price, speed, hd, ram, cd, model, code
2.FROM PC;
```

Ниже приводится результат выполнения этого запроса.

price	speed	hd	ram	Cd	model	code
600	500	5	64	12x	1232	1
850	750	14	128	40x	1121	2
600	500	5	64	12x	1233	3
850	600	14	128	40x	1121	4
850	600	8	128	40x	1121	5
950	750	20	128	50x	1233	6
400	500	10	32	12x	1232	7
350	450	8	64	24x	1232	8
350	450	10	32	24x	1232	9
350	500	10	32	12x	1260	10
980	900	40	128	40x	1233	11

Вертикальную проекцию таблицы PC можно получить, если перечислить только необходимые поля. Например, чтобы получить информацию только о частоте процессора и объеме оперативной памяти компьютеров, следует выполнить запрос:

```
1. SELECT speed, ram
2. FROM PC;
```

который вернет следующие данные:

speed	ram
500	64
750	128
500	64
600	128
600	128
750	128
500	32
450	64
450	32
500	32
900	128

Следует отметить, что вертикальная выборка может содержать дубликаты строк в том случае, если она не содержит потенциального ключа, однозначно определяющего запись. В таблице PC потенциальным ключом является поле code. Поскольку это поле отсутствует в запросе, в приведенном выше результирующем наборе имеются дубликаты строк (например, строки 1 и 3).

Если требуется получить только уникальные строки (скажем, нас интересуют только различные комбинации скорости процессора и объема памяти, а не характеристики всех имеющихся компьютеров), то можно использовать ключевое слово **DISTINCT**:

```
1.SELECT DISTINCT speed, ram
2.FROM PC;
```

что даст следующий результат:

speed	ram
450	32
450	64
500	32
500	64
600	128
750	128
900	128

Помимо **DISTINCT** может применяться также ключевое слово **ALL** (все строки), которое принимается по умолчанию.

Чтобы упорядочить строки результирующего набора, можно выполнить сортировку по любому количеству полей, указанных в предложении **SELECT**. Для этого используется предложение **ORDER BY** список полей, являющееся всегда последним предложением в операторе **SELECT**. При этом в списке полей могут указываться как имена полей, так и их порядковые позиции в списке предложения **SELECT**. Так,

если требуется упорядочить результирующий набор по объему оперативной памяти в порядке убывания, можно записать:

```
1.SELECT DISTINCT speed, ram
2.FROM PC
3.ORDER BY ram DESC;
```

или

```
1.SELECT DISTINCT speed, ram
2.FROM PC
3.ORDER BY 2 DESC;
```

Результат, приведенный ниже, будет, естественно, одним и тем же.

speed	ram
600	128
750	128
900	128
450	64
500	64
450	32
500	32

Сортировку можно проводить по возрастанию (параметр **ASC** принимается по умолчанию) или по убыванию (параметр **DESC**).

Примечание:

Не рекомендуется в приложениях использовать запросы с сортировкой по номерам столбцов. Это связано с тем, что со временем структура таблицы может измениться, например, в результате добавления/удаления столбцов. Как следствие, запрос типа

```
1. SELECT *
2. FROM PC
3. ORDER BY 3;
```

может давать совсем другую последовательность или вообще вызывать ошибку, ссылаясь на отсутствующий столбец.

Сортировка по двум полям

```
1. SELECT DISTINCT speed, ram
2. FROM PC
3. ORDER BY ram DESC, speed DESC;
```

даст следующий результат:

speed	ram
900	128
750	128
600	128
500	64
450	64
500	32

450	32

Горизонтальную выборку реализует предложение **WHERE** предикат, которое записывается после предложения **FROM**. При этом в результирующий набор попадут только те строки из источника записей, для каждой из которых значение предиката равно **TRUE**. То есть предикат проверяется для каждой записи. Например, запрос «получить информацию о частоте процессора и объеме оперативной памяти для компьютеров с ценой ниже $500» можно сформулировать следующим образом:

```
1.SELECT DISTINCT speed, ram
2.FROM PC
3.WHERE price < 500
4.ORDER BY 2 DESC;
```

speed	Ram
450	64
450	32
500	32

В последнем запросе был применен предикат сравнения с использованием операции сравнения «<» (меньше чем). Кроме этой операции сравнения могут использоваться: «=» (равно), «>» (больше), «>=» (больше или равно), «<=» (меньше или равно) и «<>» (не равно). Выражения в предикатах сравнения могут содержать константы и любые поля из таблиц, указанных в предложении **FROM**. Символьные строки и константы типа дата/время записываются в апострофах.

Примеры простых предикатов сравнения:

предикат	описание
price < 1000	Цена меньше 1000
type = ‘laptop’	Типом продукции является портативный компьютер
cd = ‘24x’	24-скоростной CD-ROM
color <> ’y’	Не цветной принтер
ram - 128 > 0	Объем оперативной памяти свыше 128 Мбайт
Price <= speed*2	Цена не превышает удвоенной частоты процессора

Сортировку можно выполнять даже по столбцам, отсутствующим в списке **SELECT**. Естественно, эти столбцы должны присутствовать на выходе предложения FROM. Например, чтобы вывести список моделей PC, упорядоченный по убыванию цены, можно написать

```
1.SELECT model FROM PC
2.ORDER BY price DESC;
```

Обратите внимание, что сама цена (price) не выводится запросом. Исключением является неоднозначная ситуация, возникающая при исключении дубликатов. Так запрос

```
1.SELECT DISTINCT model FROM PC
2.ORDER BY price DESC;
```

уже вызовет ошибку:

ORDER BY items must appear in the select list if SELECT DISTINCT is specified.

(Элементы ORDER BY должны входить в список выбора, если указывается SELECT DISTINCT.)

По той же причине не будет работать запрос с группировкой

```
1.SELECT model FROM PC
2.GROUP BY model
3.ORDER BY price DESC;
```

Column "PC.price" is invalid in the ORDER BY clause because it is not contained in either an aggregate function or the GROUP BY clause.

(Столбец "PC.price" недопустим в предложении ORDER BY, так как он не содержится в агрегатной функции или предложении GROUP BY.)

Однако если неоднозначность устранить (выполнить сортировку по какому-либо агрегатному значению для группы), то можно "подправить" запрос:

```
1.SELECT model FROM PC
2.GROUP BY model
3.ORDER BY MAX(price) DESC;
```

Примечание:

Все приведенные здесь запросы (в том числе ошибочные) будут работать под MySQL, которая сама устраняет неоднозначность. Спросите как? Загляните в документацию MySQL. :-)

Сортировка по дням рождения

Такая сортировка может потребоваться, например, для установления очередности празднования дней рождения сотрудников. Особенность подобной сортировки заключается в том, что год не учитывается вовсе, а даты упорядочиваются сначала по месяцу, а потом по дню месяца.

Возьмём в качестве примера таблицу Battles, а именно, столбец *date*. Ясно, что сортировка просто

по дате (*date*) не даст желаемого результата из-за года (так, например, 15-е ноября будет предшествовать 20-му октября):

```
1.SELECT date
2.FROM Battles
3.ORDER BY date;
```

date
1941-05-25 00:00:00.000
1942-11-15 00:00:00.000
1943-12-26 00:00:00.000
1944-10-25 00:00:00.000
1962-10-20 00:00:00.000
1962-10-25 00:00:00.000

Для выполнения требуемой сортировки можно предложить два варианта (**SQL Server**).

1. Использование функции CONVERT

При этом способе мы преобразуем дату к текстовому представлению в формате "mm-dd"

```
1.SELECT CONVERT(CHAR(5), date, 110) "mm-dd"
2.FROM Battles;
```

по которому и выполним сортировку:

```
1. SELECT date
2. FROM Battles
3. ORDER BY CONVERT(CHAR(5),date,110);
```

date
1941-05-25 00:00:00.000
1962-10-20 00:00:00.000
1962-10-25 00:00:00.000
1944-10-25 00:00:00.000
1942-11-15 00:00:00.000
1943-12-26 00:00:00.000

2. Использование функций MONTH и DAY

Здесь мы используем встроенные функции, которые возвращают компоненты даты - месяц (**MONTH**) и день (**DAY**) соответственно. По этим компонентам выполним сортировку:

```
1. SELECT date
2. FROM Battles
3. ORDER BY MONTH(date), DAY(date);
```

Что касается производительности, то вы можете выбрать любой вариант, т.к. оптимизатор строит для них идентичные планы.

В заключение представим последний запрос в более наглядном виде, добавив в него еще и "виновника торжества":

```
1. SELECT DAY(date) BD_day, DATENAME(mm, date) BD_month,
   name
```

```
2.FROM Battles
3.ORDER BY MONTH(date), DAY(date);
```

Предикаты I

Предикаты представляют собой выражения, принимающие истинностное значение. Они могут представлять собой как одно выражение, так и любую комбинацию из неограниченного количества выражений, построенную с помощью булевых операторов **AND**, **OR** или **NOT**. Кроме того, в этих комбинациях может использоваться SQL-оператор **IS**, а также круглые скобки для конкретизации порядка выполнения операций.

Предикат в языке SQL может принимать одно из трех значений TRUE (истина), FALSE (ложь) или **UNKNOWN** (неизвестно). Исключение составляют следующие предикаты: **IS NULL** (отсутствие значения), **EXISTS** (существование), **UNIQUE** (уникальность) и **MATCH** (совпадение), которые не могут принимать значение **UNKNOWN**.

Правила комбинирования всех трех истинностных значений легче запомнить, обозначив **TRUE** как 1, **FALSE** как 0 и **UNKNOWN** как 1/2 (где-то между истинным и ложным значениями) [**2**].

AND с двумя истинностными значениями дает минимум этих значений. Например, **TRUE AND UNKNOWN** будет равно **UNKNOWN**.

OR с двумя истинностными значениями дает максимум этих значений. Например, **FALSE OR UNKNOWN** будет равно **UNKNOWN**.

Отрицание истинностного значения равно 1 минус данное истинностное значение.

Например, **NOT UNKNOWN** будет равно **UNKNOWN**.

Логические операторы при отсутствии скобок, как и арифметические операторы, выполняются в соответствии с их старшинством.

Одноместная операция NOT имеет наивысший приоритет. В этом легко убедиться, если выполнить следующие два запроса.

```
1. -- модели, не являющиеся ПК
2. -- второй предикат ничего не меняет, т.к. он добавляет
   условие,
3. -- уже учтенное в первом предикате
4. SELECT maker, model, type
5. FROM Product
6. WHERE NOT type='PC' OR type='Printer';
```

```
1. -- модели производителя A, которые не являются ПК
2. SELECT maker, model, type
3. FROM Product
4. WHERE NOT type='PC' AND maker='A';
```

Поменять порядок выполнения логических операторов можно при помощи скобок:

```
1. -- модели, не являющиеся ПК или принтером, т.е. модели
   ноутбуков в нашем случае
2. SELECT maker, model, type
3. FROM Product
4. WHERE NOT (type='PC' OR type='Printer');
```

```
1. -- модели, которые не являются ПК, выпускаемыми
   производителем A
2. SELECT maker, model, type
```

```
3. FROM Product
4. WHERE NOT (type='PC' AND maker='A');
```

Следующий приоритет имеет оператор AND. Сравните результаты следующих запросов.

```
1. -- модели ПК, выпускаемые производителем A, и любые модели
   производителя B
2. SELECT maker, model, type
3. FROM Product
4. WHERE type='PC' AND maker='A' OR maker='B';
```

```
1. -- модели ПК, выпускаемые производителем A или
   производителем B
2. SELECT maker, model, type
3. FROM Product
4. WHERE type='PC' AND (maker='A' OR maker='B');
```

***Примечание*:**

Если вы не уверены, что точно помните порядок выполнения логических операторов, ставьте скобки.

Предикат в предложении WHERE выполняет реляционную операцию ограничения, т.е. строки, появляющиеся на выходе предложения FROM ограничиваются теми, для которых предикат дает значение TRUE.

Если *cond1* и *cond2* являются простыми условиями, то ограничение по предикату

cond1 AND cond2

эквивалентно пересечению ограничений по каждому из предикатов.

Ограничение по предикату

cond1 OR cond2

эквивалентно объединению ограничений по каждому из предикатов, а ограничение по предикату

NOT cond1

эквивалентно взятию разности, когда от исходного отношения вычитается ограничение по предикату *cond1*.

Обратимся к примерам.

Получить информацию о моделях ПК производителя A.

Здесь

cond1: maker = 'A' ,

cond2: type = 'pc'.

cond1 AND cond2

```
1. SELECT * FROM product
2. WHERE maker = 'A' AND type = 'pc';
```

Пересечение

```
1. SELECT * FROM product
2. WHERE maker = 'A'
3. INTERSECT
4. SELECT * FROM product
5. WHERE type = 'pc';
```

Получить информацию о моделях производителей A и B.

Здесь

cond1: maker = 'A' ,

cond2: maker = 'B'.

cond1 OR cond2

```
1. SELECT * FROM product
2. WHERE maker = 'A' OR maker = 'B';
```

Объединение

```
1. SELECT * FROM product
2. WHERE maker = 'A'
3. UNION
4. SELECT * FROM product
5. WHERE maker = 'B';
```

В свою очередь, условия *condX* могут не быть простыми. Например,

Получить информацию о моделях ПК производителей A и B.

Решение

```
1. SELECT * FROM product
2. WHERE (maker = 'A' OR maker = 'B') AND type = 'pc';
```

можно выразить через пересечение

```
1. SELECT * FROM product
2. WHERE maker = 'A' OR maker = 'B'
3. INTERSECT
4. SELECT * FROM product
5. WHERE  type = 'pc';
```

а его эквивалентную форму

```
1. SELECT * FROM product
2. WHERE (maker = 'A' AND type = 'pc')
```

```
3.     OR (maker = 'B' AND type = 'pc');
```

через объединение

```
1. SELECT * FROM product
2. WHERE maker = 'A' AND type = 'pc'
3. UNION
4. SELECT * FROM product
5. WHERE maker = 'B' AND type = 'pc';
```

Найти модели, которые не являются ПК

Здесь

cond1: type = 'pc'

NOT cond1

```
1. SELECT * FROM product
2. WHERE NOT type = 'pc';
```

Разность

```
1. SELECT * FROM product
2. EXCEPT
3. SELECT * FROM product WHERE type = 'pc';
```

Несколько слов о производительности

Если на столбцах, по которым выполняется ограничение нет индексов, при выполнении запроса будет выполнено сканирование таблицы. В первых вариантах решений такое сканирование будет выполнено один раз, в то время как в решениях на основе объединения, пересечения и разности запросов таблица сканируется дважды, плюс будет выполнена операция, сравнивающая наборы строк, возвращаемые каждым из запросов (например, Nested Loops). Это делает запрос менее производительным, хотя, возможно, существуют

оптимизаторы, способные построить один и тот же план в двух сравниваемых нами случаях.

Предикаты сравнения

Предикат сравнения представляет собой два выражения, соединяемых оператором сравнения. Имеется шесть традиционных операторов сравнения: =, >, <, >=, <=, <>.

Данные типа **NUMERIC** (числа) сравниваются в соответствии с их алгебраическим значением.

Данные типа **CHARACTER STRING** (символьные строки) сравниваются в соответствии с их алфавитной последовательностью. Если a1a2…an и в1 в…bn — две последовательности символов, то первая «меньше» второй, если a1 < в1, или a1 = в1 и a2 < в2 и т. д. Считается также, что a1a2…an < в1в2…bm, если n < m и a1a2…an = в1в2…bn, то есть если первая строка является префиксом второй. Например, ‘folder’ < ‘for’, так как первые две буквы этих строк совпадают, а третья буква строки ‘folder’ предшествует третьей букве строки ‘for’. Также справедливо неравенство ‘bar’ < ‘barber’, поскольку первая строка является префиксом второй.

Данные типа **DATETIME** (дата/время) сравниваются в хронологическом порядке. Данные типа **INTERVAL** (временной интервал) преобразуются в соответствующие типы, а затем сравниваются как обычные числовые значения типа **NUMERIC**.

Пример 5.2.1

Получить информацию о компьютерах, имеющих частоту процессора не менее 500 МГц и цену ниже $800:

```
1. SELECT *
2. FROM PC
```

```
3. WHERE speed >= 500 AND
4.  price < 800;
```

Запрос возвращает следующие данные

code	model	speed	ram	hd	cd	price
1	1232	500	64	5	12x	600
3	1233	500	64	5	12x	600
7	1232	500	32	10	12x	400
10	1260	500	32	10	12x	350

Пример 5.2.2

Получить информацию обо всех принтерах, которые не являются матричными и стоят меньше $300:

```
1. SELECT *
2. FROM printer
3. WHERE NOT (type = 'matrix') AND
4.  price < 300;
```

Результат выполнения запроса:

code	model	color	type	price
2	1433	y	Jet	270
3	1434	y	Jet	290

Предикат BETWEEN

Синтаксис:

```
1.BETWEEN::=
2.<Проверяемое выражение> [NOT] BETWEEN
3. <Начальное выражение> AND <Конечное выражение>
```

Предикат **BETWEEN** проверяет, попадают ли значения проверяемого выражения в диапазон, задаваемый пограничными выражениями, соединяемыми служебным словом **AND**. Естественно, как и для предиката сравнения, выражения в предикате **BETWEEN** должны быть совместимы по типам.

Предикат

```
1.exp1 BETWEEN exp2 AND exp3
```

равносилен предикату

```
1.exp1 >= exp2 AND exp1 <= exp3
```

А предикат

```
1.exp1 NOT BETWEEN exp2 AND exp3
```

равносилен предикату

```
1.NOT (exp1 BETWEEN exp2 AND exp3)
```

Если значение предиката *exp1 BETWEEN exp2 AND exp3* равно TRUE, в общем случае это отнюдь не означает, что значение предиката *exp1 BETWEEN exp3 AND exp2* тоже будет TRUE, так как первый будет интерпретироваться как предикат:

```
1.exp1 >= exp2 AND exp1 <= exp3
```

а второй как:

```
1.exp1 >= exp3 AND exp1 <= exp2
```

Пример 5.2.3

Найти модель и частоту процессора компьютеров стоимостью от $400 до $600:

```
1.SELECT model, speed
2.FROM PC
3.WHERE price BETWEEN 400 AND 600;
```

model	speed
1232	500
1233	500
1232	500

Предикат IN

Синтаксис:

```
1.IN::=
2. <Проверяемое выражение> [NOT] IN (<подзапрос>)
3. | (<выражение для вычисления значения>,...)
```

Предикат **IN** определяет, будет ли значение проверяемого выражения обнаружено в наборе значений, который либо явно определен, либо получен с помощью табличного подзапроса. Здесь табличный подзапрос это обычный оператор **SELECT**, который создает одну или несколько строк для одного столбца, совместимого по типу данных со значением проверяемого выражения. Если целевой объект эквивалентен хотя бы одному из указанных в предложении **IN** значений, истинностное значение предиката **IN** будет равно **TRUE**. Если для каждого значения X в предложении **IN** целевой объект <> X, истинностное значение будет равно **FALSE**. Если подзапрос выполняется, и результат не содержит ни одной строки (пустая таблица), предикат принимает значение **FALSE**. Когда не соблюдается ни одно из упомянутых выше условий, значение предиката равно **UNKNOWN**.

Пример 5.2.4

Найти модель, частоту процессора и объем жесткого диска тех компьютеров, которые комплектуются накопителями 10 или 20 Гбайт:

```
1.SELECT model, speed, hd
2.FROM PC
3.WHERE hd IN (10, 20);
```

model	speed	hd
1233	750	20
1232	500	10
1232	450	10
1260	500	10
1233	800	20

Пример 5.2.5

Найти модель, частоту процессора и объем жесткого диска компьютеров, которые комплектуются накопителями 10 Гбайт или 20 Гбайт и выпускаются производителем A:

```
1.SELECT model, speed, hd
2.FROM PC
3.WHERE hd IN (10, 20) AND
4. model IN (SELECT model
5. FROM product
6. WHERE maker = 'A'
7. );
```

model	speed	hd
1233	750	20

1232	500	10
1232	450	10
1233	800	20

Переименование столбцов и вычисления в результирующем наборе

Имена столбцов, указанные в предложении **SELECT**, можно переименовать. Это делает результаты более читабельными, поскольку имена полей в таблицах часто сокращают с целью упрощения набора. Ключевое слово AS, используемое для переименования, согласно стандарту можно и опустить, так как оно неявно подразумевается.

Например, запрос:

```
1.SELECT ram AS Mb, hd Gb
2.FROM PC
3.WHERE cd = '24x';
```

переименует столбец ram в Mb (мегабайты), а столбец hd в Gb (гигабайты). Этот запрос возвратит объемы оперативной памяти и жесткого диска для тех компьютеров, которые имеют 24-скоростной CD-ROM:

Mb	Gb
64	8

32	10

Переименование особенно желательно при использовании в предложении **SELECT** выражений для вычисления значения. Эти выражения позволяют получать данные, которые не находятся непосредственно в таблицах. Если выражение содержит имена столбцов таблицы, указанной в предложении **FROM**, то выражение подсчитывается для каждой строки выходных данных. Так, например, чтобы вывести объем оперативной памяти в килобайтах, можно написать:

```
1.SELECT ram*1024 AS Kb, hd Gb
2.FROM PC
3.WHERE cd = '24x';
```

Теперь будет получен следующий результат:

Kb	Gb
65536	8
32768	10

Иногда бывает необходимо выводить поясняющую информацию рядом с соответствующим значением. Это можно сделать, добавив строковое выражение как дополнительный столбец. Например, запрос:

```
1.SELECT ram, 'Mb' AS ram_units, hd, 'Gb' AS hd_units
2.FROM PC
3.WHERE cd = '24x';
```

даст следующий результат:

ram	ram_units	hd	hd_units
64	Mb	8	Gb

32	Mb	10	Gb

Если же явно не указать имя для выражения, то будет принят способ именования по умолчанию, который зависит от используемой СУБД. Так, в MS Access будут использованы имена типа выражение1 и т. д., а выходной столбец в MS SQL Server вообще не будет иметь заголовка.

Согласно стандарту могут использоваться имена с ограничителями (delimited identifier), при этом в качестве ограничителя применяется символ двойной кавычки ("). Такой прием допускает присутствие в именах специальных символов и зарезервированных слов. Например, запрос

```
1. SELECT 'SELECT' "SELECT";
```

выведет значение выражения (в данном случае символьную константу 'SELECT') в столбце с именем SELECT. Т.е. мы используем зарезервированное слово в качестве имени столбца. Без этого компилятор (SQL Server) не сможет корректно выполнить разбор подобного запроса

```
1. SELECT 'SELECT' SELECT;
```

и выдаст такую ошибку:

Incorrect syntax near 'SELECT'.

(некорректный синтаксис возле 'SELECT')

Помимо стандартного ограничителя, различные СУБД допускают использование своих собственных. Например, в SQL Server наш запрос можно написать так:

```
1. SELECT 'SELECT' [SELECT];
```

В то же время, стандартный ограничитель используется параллельно, но не везде он принят настройками по умолчанию. В MSSQL настройку, отвечающую за имена с ограничителями, можно изменить с помощью оператора

```
1. SET QUOTED_IDENTIFIER { ON | OFF }
```

При этом стандартное поведение (ON) принято по умолчанию.

MySQL

Запрос

```
1. SELECT 'SELECT' "SELECT";
```

даст в результате выполнения

```
SELECTSELECT
```

Это я объясняю тем, что настройки по умолчанию допускают использование символа (") в качестве символа (') для ограничителей строковой константы. Поэтому две строки просто сливаются в одну. Однако если написать так

```
1. SELECT 'SELECT' AS "SELECT";
```

или так

```
1. SELECT 'SELECT' `SELECT`;
```

то мы получим требуемый результат.

Чтобы запретить использование двойной кавычки в качестве одинарной, можно поменять настройки на стандартные. Следующий оператор изменит настройку, о которой идет речь:

```
1. SET GLOBAL sql_mode='ANSI_QUOTES';
```

а этот все настройки сделает стандартными:

```
1. SET GLOBAL sql_mode='ANSI';
```

После этого запрос

```
1. SELECT 'SELECT' "SELECT";
```

даст

Oracle и PostgreSQL

Эти СУБД ведут себя стандартно. Следует лишь отметить, что поскольку Oracle требует присутствия предложения FROM в запросе, наш оператор следует написать так:

```
1. SELECT 'SELECT' "SELECT" FROM dual;
```

Предикат LIKE

Синтаксис:

```
1. LIKE::=
2.  <Выражение для вычисления значения строки>
3.  [NOT] LIKE <Выражение для вычисления значения строки>
4.  [ESCAPE <символ>]
```

Предикат **LIKE** сравнивает строку, указанную в первом выражении, для вычисления значения строки, называемого проверяемым значением, с образцом, который определен во втором выражении для вычисления значения строки. В образце разрешается использовать два трафаретных символа:

- символ подчеркивания (_), который можно применять вместо любого единичного символа в проверяемом значении;
- символ процента (%) заменяет последовательность любых символов (число символов в последовательности может быть от 0 и более) в проверяемом значении.

Если проверяемое значение соответствует образцу с учетом трафаретных символов, то значение предиката равно **TRUE**. Ниже приводится несколько примеров написания образцов.

Образец	Описание
'abc%'	Любые строки, которые начинаются с букв «abc»
'abc_'	Строки длиной строго 4 символа, причем первыми символами строки должны быть «abc»
'%z'	Любая последовательность символов, которая обязательно заканчивается символом «z»
'%Rostov%'	Любая последовательность символов, содержащая слово «Rostov» в любой позиции строки
'% % %'	Текст, содержащий не менее 2-х пробелов, например, "World Wide Web"

Пример 5.4.1

Найти все корабли, имена классов которых заканчиваются на букву 'о'

```
1.SELECT *
2.FROM Ships
3.WHERE class LIKE '%o' ;
```

Результатом выполнения запроса будет следующая таблица:

name	class	launched
Haruna	Kongo	1916
Hiei	Kongo	1914
Kirishima	Kongo	1915
Kongo	Kongo	1913
Musashi	Yamato	1942
Yamato	Yamato	1941

Пример 5.4.2

Найти все корабли, имена классов которых заканчиваются на букву 'о', но не на 'go'

```
1.SELECT *
2.FROM Ships
3.WHERE class NOT LIKE '%go' AND
4. class LIKE '%o' ;
```

Name	Class	launched
Musashi	Yamato	1942
Yamato	Yamato	1941

Если искомая строка содержит трафаретный символ, то следует задать управляющий символ в предложении **ESCAPE**. Этот управляющий символ должен использоваться в образце перед трафаретным символом, сообщая о том, что последний следует трактовать как обычный символ. Например, если в некотором поле следует отыскать все значения, содержащие символ «_», то шаблон '%_%' приведет к тому, что будут возвращены все записи из таблицы. В данном случае шаблон следует записать следующим образом:

```
1.'%#_%' ESCAPE '#'
```

Для проверки значения на соответствие строке «25%» можно воспользоваться таким предикатом:

```
1.LIKE '25|%' ESCAPE '|'
```

Истинностное значение предиката **LIKE** присваивается в соответствии со следующими правилами:

- если либо проверяемое значение, либо образец, либо управляющий символ есть **NULL**, истинностное значение равно **UNKNOWN**;
- в противном случае, если проверяемое значение и образец имеют нулевую длину, истинностное значение равно **TRUE**;
- в противном случае, если проверяемое значение соответствует шаблону, то предикат **LIKE** равен **TRUE**;
- если не соблюдается ни одно из перечисленных выше условий, предикат **LIKE** равен **FALSE**.

Предикат LIKE и регулярные выражения

Предикат LIKE в его стандартной редакции не поддерживает регулярных выражений, хотя ряд реализаций (в частности, Oracle) допускает их использование, расширяя возможности стандарта.

В SQL Server 2005/2008 использование регулярных выражений возможно через CLR, т.е. посредством языков Visual Studio, которые могут использоваться для написания хранимых процедур и функций.

Однако в Transact-SQL, помимо стандартных символов-шаблонов ("%" и "_"), существует еще пара символов, которые делают этот предикат LIKE более гибким инструментом. Этими символами являются:

- [ ] - одиночный символ из набора символов (например, [zxy]) или диапазона ([a-z]), указанных в квадратных скобках. При этом можно перечислить сразу несколько диапазонов (например, [0-9a-z]);
- ^ - который в сочетании с квадратными скобками исключает из поискового образца символы из набора или диапазона.

Поясним использование этих символов на примерах.

```
1. SELECT * FROM
2. (
3. SELECT '5%' name    UNION ALL
4. SELECT '55'             UNION ALL
5. SELECT '5%%'          UNION ALL
6. SELECT '3%%'          UNION ALL
7. SELECT 'a5%%'        UNION ALL
8. SELECT 'abc'             UNION ALL
9. SELECT 'abc 5% cde' UNION ALL
10.      SELECT '5c2e'             UNION ALL
11.      SELECT 'C2H5OH'       UNION ALL
12.      SELECT 'C25OH'         UNION ALL
13.      SELECT 'C54OH'
14.      ) x
15.       /* 1 */
16.       --WHERE name LIKE'5%' -- начинается с 5
17.       /* 2 */
18.       --WHERE name LIKE '5[%]' -- 5%
19.       /* 3 */
20.       --WHERE name LIKE '5|%' ESCAPE '|'-- 5%
21.       /* 4 */
22.       --WHERE name LIKE '%5|%%' ESCAPE '|' -- 5% в любом
   месте строки
```

```
23.        /* 5 */
24.        --WHERE  name  LIKE  '[0-9][a-zA-Z]%'  --  первая
   цифра, вторая буква
25.        /* 6 */
26.        --WHERE name LIKE '[a-z][0-9]%' -- первая буква,
   вторая цифра
27.        /* 7 */
28.        --WHERE name LIKE '[^0-9]%' -- начинается не на
   цифру.
29.        /* 8 */
30.        --WHERE name LIKE '%[02468]%' -- содержит четную
   цифру.
31.        /* 9 */
32.        --WHERE name LIKE '%[02468][13579]%' -- комбинация
   четная-нечетная.
```

В данном запросе генерируются некоторые данные, для поиска в которых используется предикат LIKE. Девять примеров - это соответственно 9 закомментированных предложений WHERE. Для проверки результатов выполнения запроса на сайте предварительно снимите комментарий с одной из строк, начинающихся с WHERE. Тем, кто не может использовать сайт, приведу здесь результаты выполнения этих примеров.

1. Все строки, которые начинаются с 5:

name
5%
55
5%%
5c2e

2. Поиск строки '5%'. Символ в скобках воспринимается как обычный единичный символ:

name
5%

3. Другое решение, аналогичное второму, но использующее ESCAPE-символ, указывающий, что знак % следует воспринимать как обычный символ.

4. Поиск подстроки '5%', находящейся в любом месте строки:

name
5%
5%%
a5%%
abc 5% cde

5. Поиск строки, у которой первый символ является цифрой, а второй - буквой:

name
5c2e

6. Поиск строки, у которой первый символ является буквой, а второй - цифрой. Вариант для регистронезависимого сравнения:

name
a5%%
C2H5OH
C25OH
C54OH

7. Поиск строки, которая начинается не с цифры:

name
a5%%
abc
abc 5% cde
C2H5OH
C25OH
C54OH

8. Поиск строки, которая содержит четную цифру:

name
5c2e
C2H5OH
C25OH
C54OH

9. Поиск строки, которая содержит подряд идущие четную и нечетную цифры:

name
C25OH

Использование значения NULL в условиях поиска

Предикат:

```
1. IS [NOT] NULL
```

позволяет проверить отсутствие (наличие) значения в полях таблицы. Использование в этих случаях обычных предикатов сравнения может привести к неверным результатам, так как сравнение со значением **NULL** дает результат **UNKNOWN** (неизвестно).

Так, если требуется найти записи в таблице PC, для которых в столбце price отсутствует значение (например, при поиске ошибок ввода), можно воспользоваться следующим оператором:

```
1. SELECT *
2. FROM PC
3. WHERE price IS NULL;
```

Характерной ошибкой является написание предиката в виде:

```
1. WHERE price = NULL
```

Этому предикату не соответствует ни одной строки, поэтому результирующий набор записей будет пуст, даже если имеются изделия с неизвестной ценой. Это происходит потому, что сравнение с **NULL**-значением согласно предикату сравнения оценивается как **UNKNOWN**. А строка попадает в результирующий набор только в том случае, если предикат в предложении **WHERE** есть **TRUE**. Это же справедливо и для предиката в предложении **HAVING**.

Аналогичной, но не такой очевидной ошибкой является сравнение с **NULL** в предложении CASE (см. **пункт 5.10**). Чтобы продемонстрировать эту ошибку, рассмотрим такую задачу: «Определить год спуска на воду кораблей из таблицы Outcomes. Если последний неизвестен, указать 1900».

Поскольку год спуска на воду (launched) находится в таблице Ships, нужно выполнить левое соединение (см. **пункт 5.6**):

```
1.SELECT ship, launched
2.FROM Outcomes o LEFT JOIN
3. Ships s ON o.ship = s.name;
```

Для кораблей, отсутствующих в Ships, столбец *launched* будет содержать **NULL**-значение. Теперь попробуем заменить это значение значением 1900 с помощью оператора **CASE** (см. **пункт 5.10**):

```
1.SELECT ship, CASE launched
2. WHEN NULL
3. THEN 1900
4. ELSE launched
5. END 'year'
6.FROM Outcomes o LEFT JOIN
7. Ships s ON o.ship=s.name;
```

Однако ничего не изменилось. Почему? Потому что использованный оператор **CASE** эквивалентен следующему:

```
1.CASE
2. WHEN launched = NULL
3. THEN 1900
4. ELSE launched
5. END 'year'
```

А здесь мы получаем сравнение с **NULL**-значением, и в результате — **UNKNOWN**, что приводит к использованию ветви ELSE, и все остается, как и было. Правильным будет следующее написание:

```
1.CASE
2. WHEN launched IS NULL THEN 1900
3. ELSE launched
4.END 'year'
```

то есть проверка именно на присутствие **NULL**-значения.

Получение итоговых значений

Как узнать количество моделей ПК, выпускаемых тем или иным поставщиком? Как определить среднее значение цены на компьютеры, имеющие одинаковые технические характеристики? На эти и многие другие вопросы, связанные с некоторой статистической информацией, можно получить ответы при помощи итоговых (агрегатных) функций. Стандартом предусмотрены следующие агрегатные функции:

название	описание
COUNT(*)	Возвращает количество строк источника записей
COUNT	Возвращает количество значений в указанном столбце
SUM	Возвращает сумму значений в указанном столбце
AVG	Возвращает среднее значение в указанном столбце
MIN	Возвращает минимальное значение в указанном столбце
MAX	Возвращает максимальное значение в указанном столбце

Все эти функции возвращают единственное значение. При этом функции **COUNT**, **MIN** и **MAX** применимы к данным любого типа, в то время как **SUM** и **AVG** используются только для данных числового типа. Разница

между функцией **COUNT**(*) и **COUNT**(имя столбца | выражение) состоит в том, что вторая (как и остальные агрегатные функции) при подсчете не учитывает **NULL**-значения.

Пример 5.5.1

Найти минимальную и максимальную цену на персональные компьютеры:

```
1.SELECT MIN(price) AS Min_price,
2.    MAX(price) AS Max_price
3.FROM PC;
```

Результатом будет единственная строка, содержащая агрегатные значения:

Min_price	Max_price
350.0	980.0

Пример 5.5.2

Найти имеющееся в наличии количество компьютеров, выпущенных производителем А

```
1.SELECT COUNT(*) AS Qty
2.FROM PC
3.WHERE model IN(SELECT model
4. FROM Product
5. WHERE maker = 'A'
6. );
```

В результате получим

Qty
8

Пример 5.5.3

Если же нас интересует количество различных моделей, выпускаемых производителем А, то запрос можно сформулировать следующим образом (пользуясь тем фактом, что в таблице Product номер модели - столбец model - является первичным ключом и, следовательно, не допускает повторений):

```
1.SELECT COUNT(model) AS Qty_model
2.FROM Product
3.WHERE maker = 'A';
```

Qty_model
7

Пример 5.5.4

Найти количество имеющихся различных моделей ПК, выпускаемых производителем А.

Запрос похож на предыдущий, в котором требовалось определить общее число моделей, выпускаемых производителем А. Здесь же требуется найти число различных моделей в таблице PC (то есть имеющихся в продаже).

Для того чтобы при получении статистических показателей использовались только уникальные значения, при аргументе агрегатных функций можно применить параметр **DISTINCT**. Другой параметр **- ALL** - задействуется по умолчанию и предполагает подсчет всех возвращаемых (не **NULL**) значений в столбце. Оператор

```
1.SELECT COUNT(DISTINCT model) AS Qty
2.FROM PC
3.WHERE model IN (SELECT model
4. FROM Product
5. WHERE maker = 'A'
6. );
```

даст следующий результат

Qty
2

Если же нам требуется получить количество моделей ПК, производимых каждым производителем, то потребуется использовать предложение **GROUP BY**, синтаксически следующего после предложения **WHERE**.

Предложение GROUP BY

Предложение **GROUP BY** используется для определения групп выходных строк, к которым могут применяться агрегатные функции (**COUNT**, **MIN**, **MAX**, **AVG** и **SUM**). Если это предложение отсутствует, и используются агрегатные функции, то все столбцы с именами, упомянутыми в **SELECT**, должны быть включены в агрегатные функции, и эти функции будут применяться ко всему набору строк, которые удовлетворяют предикату запроса. В противном случае все столбцы списка **SELECT**, не вошедшие в агрегатные функции, должны быть указаны в предложении **GROUP BY**. В результате чего все выходные строки запроса разбиваются на группы, характеризуемые одинаковыми комбинациями значений в этих столбцах. После чего к каждой группе будут применены агрегатные функции. Следует иметь в виду, что для **GROUP BY** все значения **NULL** трактуются как равные, то есть при группировке по полю, содержащему **NULL**-значения, все такие строки попадут в одну группу.

Если при наличии предложения **GROUP BY**, в предложении **SELECT** отсутствуют агрегатные функции, то запрос просто вернет по одной строке из каждой группы. Эту возможность, наряду с ключевым словом **DISTINCT**, можно использовать для

исключения дубликатов строк в результирующем наборе.

Рассмотрим простой пример:

```
1.SELECT model, COUNT(model) AS Qty_model,
2.     AVG(price) AS Avg_price
3.FROM PC
4.GROUP BY model;
```

В этом запросе для каждой модели ПК определяется их количество и средняя стоимость. Все строки с одинаковыми значениями model (номер модели) образуют группу, и на выходе **SELECT** вычисляются количество значений и средняя цена для каждой группы. Результатом выполнения запроса будет следующая таблица

model	Qty_model	Avg_price
1121	3	850
1232	4	425
1233	3	843,333333333333
1260	1	350

Если бы в **SELECT** присутствовал столбец с датой, то можно было бы вычислять эти показатели для каждой конкретной даты. Для этого нужно добавить дату в качестве группирующего столбца, и тогда агрегатные функции вычислялись бы для каждой комбинации значений {модель, дата}.

Существует несколько определенных правил выполнения агрегатных функций.

- Если в результате выполнения запроса не получено ни одной строки (или ни одной строки для данной группы), то исходные данные для вычисления любой из агрегатных функций отсутствуют. В этом случае

результатом выполнения функций **COUNT** будет нуль, а результатом всех других функций — **NULL**. Данное свойство может дать не всегда очевидный результат. Рассмотрим, например, такой запрос:

```
1.SELECT 1 a WHERE
2.EXISTS(SELECT  MAX(price)
3.         FROM PC
4.         WHERE price<0);
```

Подзапрос в предикате EXISTS возвращает одну строку с NULL в качестве значения столбца. Поэтому, несмотря на то, что ПК с отрицательными ценами нет в базе данных, запрос в примере вернет 1.

- Аргумент агрегатной функции не может сам содержать агрегатные функции (функция от функции). То есть в простом запросе (без подзапросов) нельзя, скажем, получить максимум средних значений.
- Результат выполнения функции **COUNT** есть целое число (INTEGER). Другие агрегатные функции наследуют типы данных обрабатываемых значений.
- Если при выполнении функции **SUM** будет получен результат, превышающий максимально возможное значение для используемого типа данных, возникает ошибка.

Итак, агрегатные функции, включенные в предложение **SELECT** запроса, не содержащего предложения **GROUP BY**, исполняются над всеми результирующими строками этого запроса. Если же запрос содержит предложение **GROUP BY**, каждый набор строк, который имеет одинаковые значения столбца или группы столбцов, заданных в предложении **GROUP BY**, составляют группу, и агрегатные функции выполняются для каждой группы отдельно.

Предложение HAVING

Если предложение **WHERE** определяет предикат для фильтрации строк, то

предложение **HAVING** применяется после группировки для определения аналогичного предиката, фильтрующего группы по значениям агрегатных функций. Это предложение необходимо для проверки значений, которые получены с помощью агрегатной функции не из отдельных строк источника записей, определенного в предложении **FROM**, а из групп таких строк. Поэтому такая проверка не может содержаться в предложении **WHERE**.

Пример 5.5.5

Получить количество ПК и среднюю цену для каждой модели, средняя цена которой менее $800

```
1.SELECT model, COUNT(model) AS Qty_model,
2.   AVG(price) AS Avg_price
3.FROM PC
4.GROUP BY model
5.HAVING AVG(price) < 800;
```

В результате выполнения запроса получим:

model	Qty_model	Avg_price
1232	4	425
1260	1	350

Заметим, что в предложении **HAVING** нельзя использовать псевдоним (Avg_price), используемый для именования значений агрегатной функции в предложении **SELECT**. Дело в том, что предложение **SELECT**, формирующее выходной набор запроса, выполняется предпоследним перед предложением **ORDER BY**. Ниже приведен порядок обработки предложений в операторе **SELECT**:

1. FROM

2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. ORDER BY

Этот порядок не соответствует синтаксическому порядку общего представления оператора **SELECT**, который ближе к естественному языку:

```
1.SELECT [DISTINCT | ALL]{*
2. | [<выражение для столбца> [[AS] <псевдоним>]] [,…]}
3.FROM <имя таблицы> [[AS] <псевдоним>] [,…]
4.[WHERE <предикат>]
5.[[GROUP BY <список столбцов>]
6.[HAVING <условие на агрегатные значения>] ]
7.[ORDER BY <список столбцов>]
```

Следует отметить, что предложение HAVING может использоваться и без предложения GROUP BY. При отсутствии предложения GROUP BY агрегатные функции применяются ко всему выходному набору строк запроса, т.е. в результате мы получим всего одну строку, если выходной набор не пуст.

Таким образом, если условие на агрегатные значения в предложение HAVING будет истинным, то эта строка будет выводиться, в противном случае мы не получим ни одной строки. Рассмотрим такой пример.

Пример 5.5.6

Найти максимальную, минимальную и среднюю цену на персональные компьютеры.

Решение этой задачи дает следующий запрос:

```
1.SELECT MIN(price) AS min_price,
2.MAX(price) AS max_price, AVG(price) avg_price
3.FROM PC;
```

результатам которого будет

min_price	max_price	avg_price
350.00	980.00	675.00

Если же мы добавим в условие ограничение, скажем, на среднюю цену:

Найти максимальную, минимальную и среднюю цену на персональные компьютеры при условии, что средняя цена не превышает $600:

```
1.SELECT MIN(price) AS min_price,
2.MAX(price) AS max_price, AVG(price) avg_price
3.FROM PC
4.HAVING AVG(price) <= 600;
```

то в результате получим пустой результирующий набор, т.к. 675.00 > 600.

Получение итоговых данных с помощью оператора ROLLUP

Посчитаем сумму прихода на каждый из пунктов по таблице Income. Это несложно сделать при помощи запроса

```
1.SELECT point, SUM(inc) Qty
2.FROM Income GROUP BY point;
```

Пусть наряду с этим нам требуется вывести сумму по всем пунктам, т.е. результат должен выглядеть так:

point	Qty
1	66500.00
2	13000.00
3	3100.00
ALL	82600.00

Для решения подобной задачи в операторе SELECT имеется спецификация **ROLLUP**. С её помощью достичь требуемого результата не составляет труда:

```
1.SELECT CASE WHEN point IS NULL THEN 'ALL' ELSE CAST(point
  AS varchar) END point,
2.SUM(inc) Qty
3.FROM Income GROUP BY point WITH ROLLUP;
```

Поскольку значения столбца должны быть одного типа, номер пункта приёма приводится к символьному типу.

Последний запрос можно переписать в иной (стандартной) синтаксической форме:

```
1.SELECT CASE WHEN point IS NULL THEN 'ALL' ELSE CAST(point
  AS varchar) END point,
2.SUM(inc) Qty
3.FROM Income GROUP BY ROLLUP(point);
```

Вместо ROLLUP в нашем запросе можно также использовать оператор CUBE:

```
1. SELECT CASE WHEN point IS NULL THEN 'ALL' ELSE CAST(point
   AS varchar) END point,
2.     SUM(inc) Qty
3.     FROM Income
4.     GROUP BY point WITH CUBE;
```

Подробно о различиях между этими двумя операторами вы можете почитать в статье Бена Ричардсона.

Если СУБД не поддерживает конструкцию ROLLUP, можно использовать либо **UNION**, либо внешнее соединение (**FULL JOIN**), что позволяет объединить два запроса в один.

Ниже приводятся эти решения.

UNION

```
1. SELECT CAST(point AS varchar) point, SUM(inc) Qty
2. FROM Income GROUP BY point
3. UNION ALL
4. SELECT 'ALL', SUM(inc)
5. FROM Income;
```

FULL JOIN

```
1. SELECT          coalesce(X.point,Y.point)          point,
   coalesce(X.Qty,Y.Qty) Qty FROM
2. (SELECT CAST(point AS varchar) point, SUM(inc) Qty
3. FROM Income GROUP BY point) X
4. FULL JOIN
5. (SELECT 'ALL' point, SUM(inc) Qty
6. FROM Income) Y ON 1 = 2;
```

В последнем решении следует обратить внимание на то, что соединение выполняется по заведомо ложному предикату, т.к. нам нужны строки из обеих таблиц, которые бы не конкатенировались друг с другом.

Комбинация детализированных и агрегированных данных

Пусть, помимо модели и цены принтера, требуется еще вывести максимальную и минимальную цену по всему множеству принтеров. Для новичка подобная задача зачастую представляет определенную сложность. Эта сложность состоит в дилемме: группировка или агрегация по всему множеству. Если использовать группировку, то максимум и минимум будут получены не по всей выборке, а для каждого подмножества, определяемого группировкой (в данном примере для каждой группы с одинаковой комбинацией значений {модель, цена}). Поскольку эта комбинация уникальна в таблице Printer учебной базы данных, то мы получим 3 совпадающих значения цены:

```
1. SELECT model, price, MIN(price) min_price, MAX(price)
   max_price
2. FROM printer
3. GROUP BY model, price;
```

Если же не использовать группировку, то мы можем получить только минимальное и максимальное значение, поскольку стандартный синтаксис запрещает (ввиду неоднозначности трактовки результата) использовать наряду с агрегатами детализированные данные, по которым не выполняется группировка:

```
1. SELECT MIN(price) min_price, MAX(price) max_price
2. FROM printer;
```

Проблема разрешается довольно просто, причем не единственным способом. Так можно использовать подзапросы в предложении SELECT для каждого агрегатного значения. Это возможно, поскольку подзапрос вернет одно значение, а не набор:

Решение 1

```
1.SELECT model, price,
2.(SELECT MIN(price) FROM Printer) min_price,
3.(SELECT MAX(price) FROM Printer) max_price
4.FROM printer;
```

Более эффективным приемом будет использование подзапроса для вычисления агрегатов в предложении FROM, наряду с декартовым произведением. Бояться декартового произведения в этом случае не нужно, т.к. подзапрос вернет только одну строку, которая и будет соединяться с каждой строкой детализированных данных.

Решение 2

```
1.SELECT model, price, min_price, max_price
2.FROM printer CROSS JOIN
3.(SELECT MIN(price) min_price, MAX(price) max_price
4.FROM printer) X;
```

Почему мы утверждаем, что второй запрос будет эффективней? Дело в том, что в *решении 1* подзапрос будет вычисляться дважды, а не один раз, как это делается во втором решении. Кроме того, если оптимизатор недостаточно "умный", подзапросы в первом решении будут вычисляться для каждой строки детализированных данных. Проверьте планы выполнения для своей СУБД.

Рассмотрим теперь задачу, когда агрегат зависит от текущей строки детализированных данных, например, такую.

Вывести номер модели и цену принтера, а также максимальную и минимальную цену на принтеры того же типа.

Попытаемся адаптировать для этой задачи рассмотренные выше подходы. В *решении 1* подзапросы следует сделать коррелирующими :

Решение 1M

```
1. SELECT model, price, type,
2. (SELECT MIN(price) FROM Printer P1 WHERE P1.type=P.type)
   min_price,
3. (SELECT MAX(price) FROM Printer P1 WHERE P1.type=P.type)
   max_price
4. FROM printer P;
```

В *решении 2* мы можем воспользоваться нестандартным соединением **CROSS APPLY** (SQL Server), использующим коррелирующий подзапрос в предложении FROM.

Решение 2M

```
1. SELECT model, price, P.type, min_price, max_price
2. FROM Printer P CROSS APPLY
3. (SELECT MIN(price) min_price, MAX(price) max_price
4. FROM Printer P1
5. WHERE P1.type=P.type) X;
```

Для наглядности в *решения 1M* и *2M* добавлен столбец *type*.

Сортировка и NULL-значения

Если столбец, по которому выполняется сортировка, допускает **NULL**-значения, то при использовании **SQL Server** следует иметь в виду, что при сортировке по возрастанию NULL-значения будут идти в начале списка, а при сортировке по убыванию - в конце.

Поскольку в доступных в учебнике базах NULL-значения отсутствуют в представленных данных (коль скоро они согласованы с данными в открытых базах, используемых на сайте sql-ex.ru), я создал копию

таблицы PC с именем PC_, в которую добавил строку, содержащую NULL в столбце *price*:

```
1.INSERT INTO PC_
2.VALUES(13,2112,600,64,8,'24x',NULL);
```

Следует отметить, что это не противоречит схеме данных.

Теперь вы сами можете убедиться в сказанном, выполнив пару приведенных ниже запросов.

```
1.SELECT * FROM PC_ ORDER BY price;
```

```
1. SELECT * FROM PC_ ORDER BY price DESC;
```

Почему это важно? Дело в том, что при поиске экстремальных значений часто используют метод, основанный на сортировке. Рассмотрим, например, такую задачу.

Найти модели ПК, имеющих минимальную цену.

Иногда эту задачу решают следующим образом:

```
1.SELECT TOP 1 WITH ties model
2.FROM PC_
3.ORDER BY price;
```

Конструкция **WITH TIES** используется для того, чтобы вывести все модели с наименьшей ценой, если их окажется несколько. Однако в результате мы получим модель 2112, цена которой неизвестна, в то время как должны получить модели 1232 и 1260, имеющих действительно минимальные цены. Мы их и получим, если исключим из рассмотрения модели с неизвестными ценами:

```
1.SELECT TOP 1 WITH ties model
2.FROM PC_
3.WHERE price IS NOT NULL
4.ORDER BY price;
```

Но тут появляется еще одна проблема, связанная с дубликатами. Поскольку есть два ПК модели 1232 с минимальной ценой, то обе они будут выводиться в результирующем наборе. DISTINCT без указания в списке столбцов предложения SELECT тех, по которым выполняется сортировка, использовать мы не можем, о чем и сообщает ошибка, если мы попытаемся это сделать

```
1.SELECT DISTINCT TOP 1 WITH ties model
2.FROM PC_
3.WHERE price IS NOT NULL
4.ORDER BY price;
```

(ORDER BY items must appear in the select list if SELECT DISTINCT is specified.)

Чтобы получить решение в требуемом виде, мы можем добавить *price* в список выводимых столбцов, а потом использовать полученный запрос в качестве подзапроса. Итак,

```
1.SELECT model FROM (
2.SELECT DISTINCT TOP 1 WITH ties model, price
3.FROM PC_
4.WHERE price IS NOT NULL
5.ORDER BY price
6.) X;
```

Примечание:

При использовании <u>агрегатных функций</u> проблемы с NULL-значениями не возникает, т.к. они автоматически исключаются из рассмотрения. Хотя при этом тоже придется использовать подзапрос:

```
1. SELECT DISTINCT model FROM PC_
2. WHERE price = (SELECT MIN(price) FROM PC_);
```

Заметим также, что это стандартное решение будет работать под любыми СУБД, т.к. не использует специфических особенностей диалекта.

А как, кстати, обстоят дела с использованием метода на основе сортировки в других СУБД?

В **MySQL** мы можем использовать DISTINCT без обязательного указания в списке SELECT столбцов, по которым выполняется сортировка. Однако здесь нет аналога конструкции WITH TIES, чтобы решить задачу максимально просто.

Поэтому в методе, основанном на сортировке, нам придется использовать подзапрос, чтобы вывести все модели с минимальной ценой:

```
1. SELECT DISTINCT model FROM PC_
2. WHERE price = (SELECT price FROM PC_ WHERE price IS NOT
   NULL ORDER BY price LIMIT 1);
```

Такое же решение будет работать и в **PostgreSQL**, однако он имеет одну особенность, о которой полезно знать. А именно, при сортировке можно указать, где будут выводиться NULL-значения - в начале или в конце результирующего набора. Нам для решения задачи требуется, чтобы NULL выводились в конце отсортированного списка. Тогда не придется выполнять лишнюю операцию по отфильтровыванию NULL-значений:

```
1. SELECT DISTINCT model FROM PC_
2. WHERE price = (SELECT price FROM PC_ ORDER BY price nulls
   last LIMIT 1);
```

Кстати, при сортировке по возрастанию NULL-значения в PostgreSQL идут в конце результирующего рабора. Поэтому конструкция **NULLS LAST**, которую мы использовали выше, можно опустить при решении нашей задачи:

```
1. SELECT DISTINCT model FROM PC_
2. WHERE price = (SELECT price FROM PC_ ORDER BY price LIMIT
   1);
```

Для того чтобы NULL-значения шли в начале результирующего набора при выполнении сортировки, нужно написать **NULLS FIRST**.

К слову, мы можем смоделировать в MySQL использование конструкций NULLS FIRST/LAST. Для этого воспользуемся тем фактом, что значения логического типа в этой СУБД представляют собой TINYINT(1). Конкретно это означает, что 0 соответствует истинностному значению FALSE (ложь), а ненулевое значение эквивалентно TRUE (истина). При этом логическое выражение, оцениваемое как TRUE будет представлено единицей, т.е.

```
1. SELECT a IS NULL AS a, b IS NULL AS b FROM (SELECT NULL
   AS a, 1 AS b) x;
```

даст нам

a	b
1	0

Учитывая то, что 0 при сортировке по возрастанию идет раньше, чем 1, мы можем решение для PostgreSQL адаптировать для MySQL:

```
1. SELECT DISTINCT model FROM PC_
2. WHERE price = (SELECT price FROM PC_ ORDER BY price IS
   NULL, price LIMIT 1);
```

Oracle, как и PostgreSQL, при сортировке по возрастанию помещает NULL-значения в конец результирующего набора. Здесь также имеют место конструкции NULLS FIRST/LAST, но отсутствует аналог LIMIT/TOP N для ограничения количества выводимых строк.

Чтобы смоделировать в Oracle использованный выше подход к решению задачи, можно воспользоваться встроенной функцией **ROWNUM**. Эта функция нумерует строки, но делает это она после выполнения предложений FROM и WHERE, т.е. перед предложениями SELECT и ORDER BY. Такое поведение иллюстрирует результат следующего запроса:

```
1. SELECT code, model,price, ROWNUM rn FROM PC_ ORDER BY
   price;
```

CODE	MODEL	PRICE	RN
10	1260	350	10
9	1232	350	9
8	1232	350	8
7	1232	400	7
3	1233	600	3
1	1232	600	1
5	1121	850	5
2	1121	850	2
4	1121	850	4
6	1233	950	6
12	1233	970	12
11	1233	980	11

13	2112	NULL	13

Как видно, номер строки не соответствует порядку сортировки. Нетрудно убедиться в том, что нумерация выполнена в соответствии со столбцом code. Это объясняется тем, что оптимизатор использует индекс по этому столбцу при выполнении запроса.

Итак, чтобы найти минимальную цену на основе сортировки, придется использовать подзапрос:

```
1.SELECT price FROM(
2.SELECT model,price FROM PC_ ORDER BY price
3.) X
4.WHERE ROWNUM = 1;
```

Теперь, как и в случае MySQL и PostgreSQL, будем использовать этот запрос для получения моделей, которые продаются по цене, найденной с его помощью:

```
1.SELECT DISTINCT model FROM PC_ WHERE price =
2.(SELECT price FROM(
3.SELECT model,price FROM PC_ ORDER BY price
4.) X
5.WHERE ROWNUM = 1
6.);
```

Как говорил Соломон, от многой мудрости много скорби, и умножающий знание умножает печаль.

Используйте стандартные решения, сказал бы я. :-)

В заключение не могу не сказать о способе, использующем ранжирующие функции.

Идея решения состоит в ранжировании (функция **RANK**) строк по возрастанию цены и выборке (уникальных) строк, для которых ранг равен 1. Чтобы запрос работал под всеми СУБД, которые поддерживают оконные функции, этот алгоритм можно записать следующим образом:

```
1.SELECT DISTINCT model FROM (
2.SELECT model, Rank() OVER (ORDER BY price) rn FROM PC_
3.WHERE price IS NOT NULL
4.) X WHERE rn =1;
```

Тот факт, что при сортировке по возрастанию NULL-значения идут в начале (SQL Server) можно использовать в "полезных целях".

Пусть нам требуется вывести список рейсов, в котором рейсы из Ростова должны идти первыми, а затем остальные в алфавитном порядке города отправления.

Здесь весьма кстати пригодится функция **NULLIF**(town_from,'Rostov'), которая будет возвращать NULL, если городом отправления является 'Rostov'.

Задачу решает следующий запрос:

```
1.SELECT trip_no, town_from, town_to
2.FROM Trip
3.ORDER BY NULLIF(town_from,'Rostov'), trip_no;
```

Агрегатная функция от агрегатной функции

Давайте рассмотрим такую задачу:

Найти максимальное значение среди средних цен ПК, посчитанных для каждого производителя отдельно.

Посчитать средние значения стоимости по производителям труда не составляет:

```
1.SELECT AVG(price) avg_price
2.FROM Product P JOIN PC ON P.model = PC.model
3. GROUP BY maker;
```

Однако стандарт запрещает использовать подзапрос в качестве аргумента агрегатной функции, т.е. нельзя решить задачу следующим способом:

```
1. SELECT MAX(
2. SELECT AVG(price) avg_price
3. FROM Product P JOIN PC ON P.model = PC.model
4. GROUP BY maker
5. );
```

В подобных случаях используется подзапрос в предложении FROM:

```
1.SELECT MAX(avg_price)
2.FROM (SELECT AVG(price) avg_price
3.      FROM Product P JOIN PC ON P.model = PC.model
4.      GROUP BY maker
5.     ) X;
```

С помощью новых возможностей языка – оконных функций - эту задачу можно решить без подзапроса:

```
1.SELECT DISTINCT MAX(AVG(price)) OVER () max_avg_price
2.      FROM Product P JOIN PC ON P.model = PC.model
3.     GROUP BY maker;
```

Обратите внимание, что оконные функции допускают использование агрегатной функции в качестве аргумента. Ключевое слово **DISTINCT** необходимо здесь, поскольку максимальное значение, подсчитанное по всему набору средних значений, будет «приписано» каждому производителю.

Стандарт также запрещает использовать агрегатную функцию как аргумент другой агрегатной функции. Т.е. мы не можем решить нашу задачу следующим образом:

```
1. SELECT MAX(AVG(price)) max_avg_price
2.     FROM Product P JOIN PC ON P.model = PC.model
3.     GROUP BY maker;
```

Но не бывает правил без исключений. Как ни странно, но в **Oracle** подобные конструкции работают, и вышеприведенный запрос даст результат:

MAX_AVG_PRICE
850

Чтобы убедиться в этом, зайдите на страницу задач обучающего этапа на сайте sql-ex.ru, выберите Oracle в списке СУБД и выполните запрос с флажком "Без проверки".

Кстати говоря, решение с использованием оконной функции также будет работать в Oracle. Могу предположить, что решение без оконной функции фактически её и использует, неявно подразумевая предложение OVER().

Наверняка, вам встретятся решения подобных задач на основе сортировки с ограничением на число строк результирующего набора. Однако такие решения не являются легитимными с точки зрения стандарта языка и, как следствие, имеют различный синтаксис в разных реализациях. В качестве примера приведу решения нашей задачи в диалектах SQL Server и MySQL.

SQL Server

```
1.SELECT TOP 1 AVG(price) avg_price
2.FROM Product P JOIN PC ON P.model = PC.model
3.GROUP BY maker
4.ORDER BY avg_price DESC;
```

MySQL

```
1.SELECT AVG(price) avg_price
2.FROM Product P JOIN PC ON P.model = PC.model
3.GROUP BY maker
4.ORDER BY avg_price DESC
5.LIMIT 1;
```

Оба этих решения берут только первую строку из отсортированного по убыванию набора средних цен.

У начинающих изучать SQL зачастую вызывает проблему определение производителя, для которого достигается искомый максимум/минимум. Другими словами, требуется найти максимальную среднюю цену и производителя, средняя цена ПК которого совпадает с этой максимальной средней ценой.

Нестандартными средствами эта задача решается фактически рассмотренным выше запросом:

```
1.SELECT TOP 1 maker, AVG(price) avg_price
2.FROM Product P JOIN PC ON P.model = PC.model
3.GROUP BY maker
4.ORDER BY avg_price DESC;
```

Использование maker в списке столбцов предложения SELECT вполне допустимо, т.к. по этому столбцу выполняется группировка. Однако тут имеется одна «ловушка». Она связана с тем, что максимум может достигаться для нескольких производителей, и в данной постановке задачи их нужно выводить всех, в то время как мы ограничиваем выборку только одной

(первой) строкой. На этот случай диалект T-SQL имеет дополнительную конструкцию WITH TIES. Логически правильное решение будет иметь вид:

```
1.SELECT TOP 1 WITH TIES maker, AVG(price) avg_price
2.FROM Product P JOIN PC ON P.model = PC.model
3.GROUP BY maker
4.ORDER BY avg_price DESC;
```

Однако, если иметь в виду проблему переносимости кода, то следует предпочесть решение, использующее стандартные средства.

Примечание:

На сайте SQL-EX.RU проблема переносимости кода возникла в связи с нашим намерением реализовать упражнения для различных СУБД. Реализация потребовала бы минимальных средств, если бы тестовые решения, используемые для проверки, работали бы на всех предполагаемых СУБД без изменения своего кода. Поэтому следование стандарту может являться одним из требований тех. задания на проект.

Приведем ниже несколько стандартных решений рассматриваемой задачи.

1. Использование предиката ALL в предложении WHERE

```
1.SELECT maker, avg_price
2.FROM (SELECT maker, AVG(price) avg_price
3.      FROM Product P JOIN PC ON P.model=PC.model
4.      GROUP BY maker
5.     ) X
6.WHERE avg_price >= ALL(SELECT AVG(price) avg_price
7.      FROM Product P JOIN PC ON P.model=PC.model
8.      GROUP BY maker
9.       );
```

На естественном языке этот запрос звучит следующим образом: «Найти производителей, средняя цена на ПК у которых не меньше, чем средние цены у КАЖДОГО из производителей ПК».

2. Использование внутреннего соединения

```
1.SELECT maker, avg_price
2.FROM (SELECT maker, AVG(price) avg_price
3.      FROM Product P JOIN PC ON P.model=PC.model
4.      GROUP BY maker
5.      ) X JOIN
6.      (SELECT MAX(avg_price) max_price
7.       FROM (SELECT maker, AVG(price) avg_price
8.        FROM Product P JOIN PC ON P.model=PC.model
9.             GROUP BY maker
10.                  ) X
11.             ) Y ON avg_price = max_price;
```

Здесь мы соединяем подзапрос, определяющий производителей и средние цены на их ПК, с подзапросом, в котором определяется максимальная средняя цена. Соединение выполняется по условию равенства средней цены из первого подзапроса с максимальной ценой из второго.

3. Использование предиката ALL в предложении HAVING

```
1.SELECT maker, AVG(price) avg_price
2.FROM Product P JOIN PC ON P.model=PC.model
3.GROUP BY maker
4.HAVING AVG(price) >= ALL(SELECT AVG(price)
5.       FROM Product P JOIN PC ON P.model=PC.model
6.       GROUP BY maker
7.           );
```

Это решение отличается от первого варианта отсутствием «лишнего» запроса, который пришлось написать лишь затем, чтобы была возможность использовать алиас avg_price в предложении WHERE (смотри порядок обработки предложений оператора SELECT); с другой стороны,

использование в предложении WHERE агрегатной функции также запрещено правилами языка.

Все приведенные стандартные решения выглядят тяжеловесными, хотя и будут работать практически во всех СУБД. Эта громоздкость объясняется повторением в коде фактически одного и того же запроса. Однако общие табличные выражения – CTE, которые были введены в последних версиях стандарта, позволяют многократно ссылаться на один раз сформулированный запрос. Например, решения 1, 3 с помощью **CTE** можно записать в таком виде:

```
1.WITH cte(maker, avg_price)
2.AS (
3.SELECT maker, AVG(price) avg_price
4.FROM Product P JOIN PC ON P.model=PC.model
5.GROUP BY maker
6.   )
7.SELECT *
8.FROM cte
9.WHERE  avg_price>= ALL(SELECT avg_price
10.                          FROM cte
11.                          );
```

Замечу, что поддержка общих табличных выражений появилась в SQL Server 2005 и в PostgreSQL 8.4.

Произведение значений столбца

Почему среди агрегатных функций SQL нет произведения?

Такой вопрос часто задают в профессиональных социальных сетях. Речь идёт о произведении значений столбца таблицы при выполнении группировки. Функции типа **PRODUCT** нет в стандарте

языка, и я не знаю СУБД, которая её бы имела. Хорошей же новостью является то, что такую функцию просто выразить через три другие, которые есть в арсенале практически всех серверов БД. Итак.

Пусть требуется перемножить значения столбца value следующей таблицы:

```
1.SELECT value FROM (
2.VALUES(2),(3),(4),(5)
3.) X(value);
```

value
2
3
4
5

Воспользуемся следующим свойством логарифмов: логарифм произведения равен сумме логарифмов, для нашего примера это означает

```
1.ln(2*3*4*5) = ln(2) + ln(3) + ln(4) + ln(5)
```

Если теперь применить обратную к натуральному логарифму (**Ln**) функцию экспоненты (**exp**), то получим

```
1.exp(ln(2*3*4*5)) = 2*3*4*5 = exp(ln(2) + ln(3) + ln(4) +
  ln(5))
```

Итак, произведение чисел мы можем заменить выражением, стоящим в равенстве справа. Осталось записать эту формулу на языке SQL, учитывая, что сами числа находятся в столбце value.

```
1.SELECT exp(SUM(log(value))) product FROM (
```

```
2.VALUES(2),(3),(4),(5)
3.) X(value);
```

product
120

Правильность результата легко проверить устным счетом, или в Excel.

Рассмотренное решение не является универсальным. Поскольку логарифм не определен для чисел <=0, то если в столбце появятся такие значения, например,

```
1.SELECT exp(SUM(log(value))) product FROM (
2.VALUES(2),(-3),(4),(-5)
3.) X(value);
```

будет получено сообщение об ошибке:

An invalid floating point operation occurred.

(Попытка выполнить недопустимую операцию с плавающей запятой.)

Для учета "недопустимых" значений доработаем наше решение в соответствии со следующим алгоритмом:

1. Если среди значений есть нули, то результатом будет 0.
2. Если число отрицательных значений нечетное, то домножаем произведение абсолютных значений столбца на -1.
3. Если число отрицательных значений четное, то результатом будет произведение абсолютных значений столбца.

Вот решение с комментариями, реализующее этот алгоритм:

```
1.WITH T AS(SELECT * FROM (VALUES(-2),(-3),(4),(-5))
  X(value)),
2.P AS (
3.SELECT SUM(CASE WHEN value<0 THEN 1 ELSE 0 END) neg, --
  число отрицательных значений
4.SUM(CASE WHEN value>0 THEN 1 ELSE 0 END) pos, -- число
  положительных значений
```

```
5.COUNT(*) total -- общее число значений
6.FROM T)
7.SELECT CASE WHEN total <> pos+neg /* есть нули */ THEN 0
  ELSE
8.(CASE WHEN neg%2=1 THEN -1 ELSE +1 END)
  *exp(SUM(log(abs(value))))
9.END product
10.      FROM T,P
11.      WHERE value <> 0
12.      GROUP BY neg, pos, total;
```

product
-120

Обратите внимание на условие value <> 0 в последней строке запроса. Его присутствие связано с тем, что, хотя ветвь оператора CASE с вычислением выражения через логарифм не реализуется при наличии нулей среди значений столбца (возвращается 0), SQL Server всё равно вычисляет это выражение и возвращает ошибку.

Сообразительные уже спросили: "А как быть с NULL?"

Действительно, наше решение даёт в этом случае 0. Будем следовать общей логике поведения агрегатных функций - не учитывать NULL. Ниже приводится окончательное решение, которое имеет одно отличие по сравнению с предыдущим решением. Кто догадается какое?

```
1.WITH T AS(SELECT * FROM (VALUES(-2),(-3),(4),(-5),
  (NULL) ) X(value)),
2.P AS (
3.SELECT SUM(CASE WHEN value<0 THEN 1 ELSE 0 END) neg, --
  число отрицательных значений
4.SUM(CASE WHEN value>0 THEN 1 ELSE 0 END) pos, -- число
  положительных значений
5.COUNT(value) total -- общее число значений
6.FROM T)
7.SELECT CASE WHEN total <> pos+neg /* есть нули */ THEN 0
  ELSE
8.(CASE WHEN neg%2=1 THEN -1 ELSE +1 END)
  *exp(SUM(log(abs(value))))
9.END
```

```
10.     product  FROM T,P WHERE value <> 0 GROUP BY neg,
  pos, total;
```

Использование в запросе нескольких источников записей

Как видно из приведенной в конце предыдущего раздела синтаксической формы оператора **SELECT**, в предложении **FROM** допускается указание нескольких таблиц. Простое перечисление таблиц через запятую практически не используется, поскольку оно соответствует реляционной операции, которая называется декартовым произведением. То есть в результирующем наборе каждая строка из одной таблицы будет сочетаться с каждой строкой из другой. Например, для таблиц:

A

a	b
1	2
2	1

B

c	d
2	4
3	3

Результат запроса

```
1. SELECT *
2. FROM A, B;
```

будет выглядеть следующим образом:

a	b	c	d
1	2	2	4
1	2	3	3
2	1	2	4
2	1	3	3

Поэтому перечисление таблиц, как правило, используется совместно с условием соединения строк из разных таблиц, указываемым в предложении **WHERE**. Для приведенных выше таблиц таким условием может быть совпадение значений, скажем, в столбцах a и c:

```
1. SELECT *
2. FROM A, B
3. WHERE a = c;
```

Теперь результатом выполнения этого запроса будет следующая таблица:

a	b	c	d
2	1	2	4

то есть соединяются только те строки таблиц, у которых в указанных столбцах находятся равные значения (эквисоединение). Естественно, могут быть использованы любые условия, хотя эквисоединение используется чаще всего, поскольку эта операция воссоздает некую исходную сущность предметной области, декомпозированную на две других в результате процедуры нормализации в процессе построения логической модели.

Если разные таблицы имеют столбцы с одинаковыми именами, то для однозначности требуется использовать точечную нотацию, которая называется уточнением имени столбца:

<имя таблицы>.<имя столбца>

В тех случаях, когда это не вызывает неоднозначности, использование данной нотации не является обязательным.

Пример 5.6.1

Найти номер модели и производителя ПК, имеющих цену менее $600:

```
1.SELECT DISTINCT PC.model, maker
2.FROM PC, Product
3.WHERE PC.model = Product.model AND
4. price < 600;
```

В результате каждая модель одного и того же производителя выводится только один раз:

model	maker
1232	A
1260	E

Иногда в предложении **FROM** требуется указать одну и ту же таблицу несколько раз. В этом случае обязательным является переименование.

Пример 5.6.2

Выведи пары моделей, имеющих одинаковые цены:

```
1.SELECT DISTINCT A.model AS model_1, B.model AS model_2
2.FROM PC AS A, PC B
3.WHERE A.price = B.price AND
4.  A.model < B.model;
```

Здесь условие a.model < b.model используется для того, чтобы не выводились одинаковые пары, отличающиеся только перестановкой, например: {1232, 1233} и {1233, 1232}. **DISTINCT** применяется для того, чтобы исключить одинаковые строки, поскольку в таблице PC имеются модели с одинаковыми номерами по одной и той же цене. В результате получим следующую таблицу:

model_1	model_2
1232	1233
1232	1260

Переименование также является обязательным, если в предложении **FROM** используется подзапрос, так как, в противном случае, у нас нет возможности уточнения имени столбца из подзапроса. Так, первый пример можно переписать следующим образом:

```
1.SELECT DISTINCT PC.model, maker
2.FROM PC, (SELECT maker, model
3. FROM Product
4. ) AS Prod
5.WHERE PC.model = Prod.model AND
6. price < 600;
```

Обратите внимание, что в этом случае в других предложениях оператора **SELECT** уже нельзя использовать квалификатор Product, поскольку таблица Product уже не используется. Вместо него используется псевдоним Prod. Кроме того, ссылаться извне теперь можно только на те столбцы таблицы Product, которые перечислены в подзапросе.

За псевдонимом производного табличного выражения может в скобках стоять список имен столбцов, которые будут использоваться вместо имен табличного выражения. Порядок имен должен, естественно, соответствовать списку столбцов табличного выражения (в нашем случае - списку в предложении SELECT). Это способ позволяет избежать неоднозначности имен и, как следствие, необходимости их уточнения. Вот как может выглядеть предыдущий пример:

```
1.SELECT DISTINCT model, maker
2.FROM PC, (SELECT maker, model
3. FROM Product
4. ) AS Prod(maker, model_1)
5.WHERE model = model_1 AND
6. price < 600;
```

Явные операции соединения

В предложении **FROM** может быть указана **явная операция соединения** двух и более таблиц. Среди ряда операций соединения, описанных в стандарте языка SQL, многими серверами баз данных поддерживается только операция **соединения по предикату**. Синтаксис соединения по предикату имеет вид:

```
1.FROM <таблица 1>
```

```
2.  [INNER]
3.  {{LEFT | RIGHT | FULL }  [OUTER]} JOIN <таблица 2>
4.[ON <предикат>]
```

Соединение может быть либо внутренним (**INNER**), либо одним из внешних (**OUTER**). Служебные слова **INNER** и **OUTER** можно опускать, поскольку внешнее соединение однозначно определяется его типом — **LEFT** (левое), **RIGHT** (правое) или **FULL** (полное), а просто **JOIN** будет означать внутреннее соединение.

Предикат определяет условие соединения строк из разных таблиц. При этом **INNER JOIN** означает, что в результирующий набор попадут только те соединения строк двух таблиц, для которых значение предиката равно **TRUE**. Как правило, предикат определяет эквисоединение по внешнему и первичному ключам соединяемых таблиц, хотя это не обязательно.

Пример 5.6.3.

Найти производителя, номер модели и цену каждого компьютера, имеющегося в базе данных:

```
1.SELECT maker, Product.model AS model_1,
2.              PC.model AS model_2, price
3.FROM Product INNER JOIN
4. PC ON PC.model = Product.model
5. ORDER BY maker, model_2;
```

В данном примере в результирующем наборе будут соединяться только те строки из таблиц PC и Product, у которых совпадают номера моделей.

Для визуального контроля в результирующий набор включен как номер модели из таблицы PC, так и из таблицы Product:

Maker	model_1	model_2	price
A	1232	1232	600
A	1232	1232	400

A	1232	1232	350
A	1232	1232	350
A	1233	1233	600
A	1233	1233	950
A	1233	1233	980
A	1233	1233	970
B	1121	1121	850
B	1121	1121	850
B	1121	1121	850
E	1260	1260	350

Внешнее соединение **LEFT JOIN** означает, что помимо строк, для которых выполняется условие предиката, в результирующий набор попадут все остальные строки из первой таблицы (левой). При этом отсутствующие значения столбцов из правой таблицы будут заменены **NULL**-значениями.

Пример 5.6.4

Привести все модели ПК, их производителей и цену:

```
1. SELECT  maker,  Product.model  AS  model_1,  pc.model  AS
   model_2, price
2. FROM Product LEFT JOIN
3.  PC ON PC.model = Product.model
4. WHERE type = 'pc'
5. ORDER BY maker, PC.model;
```

Обратите внимание на то, что по сравнению с предыдущим примером пришлось использовать предложение **WHERE** для отбора только производителей ПК. В противном случае в результирующий набор попали бы также и модели портативных компьютеров, и принтеров. В рассмотренном ранее примере это условие было бы излишним, так как соединялись только те строки, у которых совпадали номера моделей, и одной из таблиц была таблица PC, содержащая только модели ПК. В результате выполнения запроса получим:

Maker	model_1	model_2	price
A	1232	1232	600
A	1232	1232	400
A	1232	1232	350
A	1232	1232	350
A	1233	1233	600
A	1233	1233	950
A	1233	1233	980
B	1121	1121	850
B	1121	1121	850
B	1121	1121	850
E	2111	NULL	NULL
E	2112	NULL	NULL
E	1260	1260	350

Поскольку моделей 2111 и 2112 из таблицы Product нет в таблице PC, в столбцах из таблицы PC содержится **NULL**.

Соединение **RIGHT JOIN** обратно соединению **LEFT JOIN**, то есть в результирующий набор попадут все строки из второй таблицы, которые будут соединяться только с теми строками из первой таблицы, для которых выполняется условие соединения. В нашем случае левое соединение

```
1.Product LEFT JOIN PC ON PC.model = Product.model
```

будет эквивалентно правому соединению

```
1.PC RIGHT JOIN Product ON PC.model = Product.model
```

Запрос же

```
1.SELECT maker, Product.model AS model_1, PC.model AS
  model_2, price
2.FROM Product RIGHT JOIN
3. PC ON PC.model = Product.model
4.ORDER BY maker, PC.model;
```

даст те же результаты, что и внутреннее соединение, поскольку в правой таблице (PC) нет таких моделей, которые отсутствовали бы в левой таблице (Product), что вполне естественно для типа связи «один ко многим», которая имеется между таблицами PC и Product.

Наконец, при полном соединении (**FULL JOIN**) в результирующую таблицу попадут не только те строки, которые имеют одинаковые значения в сопоставляемых столбцах, но и все остальные строки исходных таблиц, не имеющие соответствующих значений в другой таблице. В этих строках все столбцы той таблицы, в которой не было найдено соответствия, заполняются **NULL**-значениями. То есть полное соединение представляет собой комбинацию левого и правого внешних соединений. Так, запрос для таблиц A и B, приведенных в начале главы,

```
1.SELECT A.*, B.*
2.FROM A FULL JOIN
3.  B ON A.a = B.c;
```

даст следующий результат:

A	b	c	d
1	2	NULL	NULL
2	1	2	4
NULL	NULL	3	3

Заметим, что это соединение симметрично, то есть A **FULL JOIN** B эквивалентно B **FULL JOIN** A. Обратите также внимание на обозначение A.*, что означает вывести все столбцы таблицы A.

UNION JOIN

Этот тип соединения был введен в стандарте **SQL-92**, но в более поздних версиях стандарта отсутствует. В частности, его уже нет в стандарте SQL2003 (ANSI и ISO). Как и многие другие конструкции языка SQL, соединение **UNION JOIN** является избыточным, поскольку его можно выразить через разность полного и внутреннего соединений. Формально это можно записать следующим образом:

```
1.A UNION JOIN B :=
2.(A FULL JOIN B)
3.EXCEPT
4.(A INNER JOIN B)
```

Ну, а если полное соединение не поддерживается (MySQL), то его можно получить объединением левого и правого внешних соединений. Тогда наша формула примет вид

```
1.A UNION JOIN B :=
2.((A LEFT JOIN B)
3.UNION
4.(A RIGHT JOIN B))
5.EXCEPT
6.(A INNER JOIN B)
```

Давайте в качестве примера, где мог бы пригодиться этот тип соединения, рассмотрим следующую задачу.

Найти производителей, которые выпускают принтеры, но не ПК, или выпускают ПК, но не принтеры.

Будь у нас возможность использовать UNION JOIN, мы бы решили задачу так:

```
1. SELECT * FROM
2. (SELECT DISTINCT maker FROM Product WHERE type='pc') m_pc
3. UNION JOIN
4. (SELECT DISTINCT maker FROM Product WHERE type='printer')
   m_printer
5. ON m_pc.maker = m_printer.maker;
```

Воспользуемся формулой. Полное соединение производителей ПК и производителей принтеров даст нам как тех, кто производит что-то одно, так и тех, кто производит и то, и другое.

```
1. SELECT * FROM
2. (SELECT DISTINCT maker FROM Product WHERE type='pc') m_pc
3. FULL JOIN
4. (SELECT DISTINCT maker FROM Product WHERE type='printer')
   m_printer
5. ON m_pc.maker = m_printer.maker;
```

Теперь вычтем из результата тех, кто производит и то, и другое (внутренее соединение):

```
1. SELECT m_pc.maker m1, m_printer.maker m2  FROM
2. (SELECT maker FROM Product WHERE type='pc') m_pc
3. FULL JOIN
4. (SELECT maker FROM Product WHERE type='printer')
   m_printer
```

```
5. ON m_pc.maker = m_printer.maker
6. EXCEPT
7. SELECT * FROM
8. (SELECT maker FROM Product WHERE type='pc') m_pc
9. INNER JOIN
10.     (SELECT maker FROM Product WHERE type='printer')
   m_printer
11.     ON m_pc.maker = m_printer.maker;
```

Попутно я убрал из этого решения избыточные DISTINCT, поскольку EXCEPT выполнит исключение дубликатов. Это единственный полезный тут урок, т.к. операцию взятия разности (EXCEPT) можно заменить простым предикатом:

```
1. WHERE m_pc.maker IS NULL OR m_printer.maker IS NULL
```

или даже так

```
1. m_pc.maker + m_printer.maker IS NULL
```

ввиду того, что конкатенация с NULL-значением дает NULL.

```
1. SELECT * FROM
2. (SELECT DISTINCT maker FROM Product WHERE type='pc') m_pc
3. FULL JOIN
4. (SELECT DISTINCT maker FROM Product WHERE type='printer')
   m_printer
5. ON m_pc.maker = m_printer.maker
6. WHERE m_pc.maker IS NULL OR m_printer.maker IS NULL;
```

Наконец, чтобы представить результат в один столбец, воспользуемся функцией **COALESCE**:

```
1. SELECT COALESCE(m_pc.maker, m_printer.maker) FROM
```

```
2.(SELECT DISTINCT maker FROM Product WHERE type='pc') m_pc
3.FULL JOIN
4.(SELECT DISTINCT maker FROM Product WHERE type='printer')
  m_printer
5.ON m_pc.maker = m_printer.maker
6.WHERE m_pc.maker IS NULL OR m_printer.maker IS NULL;
```

Разумеется, это не единственный способ решения данной задачи. Он лишь демонстрирует замену репрессированного типа соединений.

Мне неизвестны СУБД, в которых было бы реализовано соединение UNION JOIN.

Ассоциативность и коммутативность соединений

Внутреннее и полное внешнее соединения являются как коммутативными, так и ассоциативными, т.е. для них справедливо следующее:

```
1.A [FULL | INNER] JOIN B = B [FULL | INNER] JOIN A
```

и

```
1.(A [FULL | INNER] JOIN B) [FULL | INNER] JOIN C =
2.A [FULL | INNER] JOIN (B [FULL | INNER] JOIN C)
```

Очевидно, что левое/правое соединения не коммутативны, т.к.

```
1.A LEFT JOIN B = B RIGHT JOIN A
```

но ассоциативны, например:

```
1.(A LEFT JOIN B) LEFT JOIN C = A LEFT JOIN (B LEFT JOIN C)
```

С практической точки зрения ассоциативность означает, что мы можем не расставлять скобки, определяющие прядок выполнения соединений.

Однако закон ассоциативности, справедливый для однотипных соединений, нарушается, если в одном запросе используются соединения разных типов. Покажем это на примере.

```
1.WITH a(a_id) AS
2.(SELECT * FROM (VALUES('1'),('2'),('3')) x(y)),
3.b(b_id) AS
4.(SELECT * FROM (VALUES('1'),('2'),('4')) x(y)),
5.c(c_id) AS
6.(SELECT * FROM (VALUES('5'),('2'),('3')) x(y))
7.SELECT a_id, b_id, c_id  FROM (a LEFT JOIN b ON a_id=b_id)
  INNER JOIN c ON b_id=c_id
8.UNION ALL
9.SELECT '','',''
10.     UNION ALL
11.     SELECT a_id, b_id, c_id  FROM  a LEFT JOIN (b INNER
  JOIN c ON b_id=c_id) ON a_id=b_id;
```

a_id	b_id	c_id
2	2	2
1	NULL	NULL
2	2	2
3	NULL	NULL

Результаты двух запросов отделены друг от друга пробельной строкой для удобства.

Заметим, что при отсутствии скобок мы получим результат, совпадающий с результатом первого запроса, поскольку соединения будут выполняться в том порядке, в каком они записаны.

Эквисоединения

Соединения, которые мы рассмотрели ранее и которые преобладают в примерах данного

учебника, называются соединениями по предикату.

Синтаксис этого вида соединения такой:

```
1. Таблица_1 <тип соединения> JOIN  Таблица_2 ON <предикат>
```

где

тип соединения := [INNER] | [OUTER]{LEFT | RIGHT | FULL}

Эти соединения являются наиболее общими, т.к. в качестве предиката может быть использовано любое логическое выражение. Именно по этой причине все диалекты поддерживают этот вид соединения.

Частным, но часто используемым соединением является эквисоединение - случай, когда предикат представляет собой равенство значений в столбцах соединяемых таблиц.

При этом соединяемые столбцы зачастую имеют одинаковые имена, поскольку в соединении участвуют таблицы, связанные внешним ключом. Впрочем, последнее не суть важно, т.к. мы можем переименовать столбцы, если это нам потребуется.

Так вот для этого частного случая соединения - эквисоединения по столбцам с одинаковыми именами - имеются отдельные синтаксические формы соединения: естественное соединение и соединение, использующее имена столбцов.

Естественное соединение

```
1. Таблица_1 NATURAL <тип соединения> JOIN  Таблица_2
```

Предикат здесь не нужен, т.к. он подразумевается, а именно попарное равенство всех столбцов с одинаковыми именами в обеих таблицах. Например, если у обеих соединяемых таблиц есть столбцы *a* и *b*, то естественное соединение

```
1. Таблица_1 NATURAL INNER JOIN  Таблица_2
```

будет эквивалентно такому соединению по предикату:

```
1. Таблица_1 INNER JOIN  Таблица_2 ON Таблица_1.a = Таблица_2.a AND Таблица_1.b = Таблица_2.b
```

Кроме того, при естественном соединении одноименные столбцы будут присутствовать в выборке в одном экземпляре. Сравните, например, результаты таких запросов (база данных Аэрофлот)

```
1.SELECT * FROM Pass_in_trip
2.       JOIN     Passenger    ON    Pass_in_trip.id_psg    =
  Passenger.id_psg
3.WHERE trip_no=1123;
```

trip_no	date	id_psg	place	id_psg	name
1123	2003-04-05 00:00:00	3	2a	3	Kevin Costner
1123	2003-04-08 00:00:00	1	4c	1	Bruce Willis
1123	2003-04-08 00:00:00	6	4b	6	Ray Liotta

и

```
1.SELECT * FROM Pass_in_trip
2.    NATURAL JOIN Passenger
3.WHERE trip_no=1123;
```

id_psg	trip_no	date	place	name
6	1123	2003-04-08 00:00:00	4b	Ray Liotta
3	1123	2003-04-05 00:00:00	2a	Kevin Costner
1	1123	2003-04-08 00:00:00	4c	Bruce Willis

Как видно из представленных результатов, столбец *id_psg*, по которому выполняется соединение, не повторяется для естественного соединения.

Из СУБД, доступных на сайте sql-ex.ru, только SQL Server не поддерживает естественное соединение. Если вы хотите поработать с естественным соединением практически, выберите в консоли PostgreSQL или MySQL.

Если требуется выполнить эквисоединение не по всем столбцам с совпадающими именами, а только по их части, тогда мы можем использовать соединение **USING**:

```
1. Таблица_1 <тип соединения> JOIN  Таблица_2 USING(<список
   столбцов>)
```

Список столбцов содержит те столбцы, по которым выполняется эквисоединение. Соответственно, в этом списке могут присутствовать только те из столбцов, имена которых совпадают в обеих соединяемых таблицах.

Сравните результаты следующих запросов (предикат в предложении WHERE использован лишь для сокращения размера выборки).

Соединение строк из таблиц Income и Outcome по равенству значений в столбце *date* (база данных Фирма вторсырья)

```
1. SELECT * FROM Income JOIN Outcome USING(date)
2. WHERE MONTH(date) >= 4;
```

date	code	point	inc	code	point	out
2001-04-13 00:00:00	6	1	5000.00	7	1	4490.00
2001-04-13 00:00:00	10	1	5000.00	7	1	4490.00
2001-05-11 00:00:00	7	1	4500.00	9	1	2530.00
2001-09-13 00:00:00	12	3	1350.00	16	3	1200.00
2001-09-13 00:00:00	13	3	1750.00	16	3	1200.00
2001-09-13 00:00:00	12	3	1350.00	17	3	1500.00
2001-09-13 00:00:00	13	3	1750.00	17	3	1500.00

Соединение строк из таблиц Income и Outcome по равенству значений в столбцах *date* и *point*

```
1. SELECT * FROM Income JOIN Outcome USING(date, point)
```

```
2. WHERE MONTH(date) >= 4;
```

point	date	code	inc	code	out
1	2001-04-13 00:00:00	6	5000.00	7	4490.00
1	2001-04-13 00:00:00	10	5000.00	7	4490.00
1	2001-05-11 00:00:00	7	4500.00	9	2530.00
3	2001-09-13 00:00:00	12	1350.00	16	1200.00
3	2001-09-13 00:00:00	13	1750.00	16	1200.00
3	2001-09-13 00:00:00	12	1350.00	17	1500.00
3	2001-09-13 00:00:00	13	1750.00	17	1500.00

Соединение строк из таблиц Income и Outcome по равенству значений в столбцах *date*, *point* и *code*

```
1. SELECT * FROM Income JOIN Outcome USING(date, point,
   code);
```

не возвращает строк.

Это соединение по всем столбцам с совпадающими именами эквивалентно естественному соединению

```
1. SELECT * FROM Income NATURAL JOIN Outcome;
```

Чтобы продемонстрировать вывод в последнем варианте, воспользуемся левым соединением

```
1. SELECT * FROM Income LEFT JOIN Outcome USING(date, point,
   code)
2. WHERE MONTH(date) >= 4;
```

code	point	date	inc	out
6	1	2001-04-13 00:00:00	5000.00	NULL
7	1	2001-05-11 00:00:00	4500.00	NULL

10	1	2001-04-13 00:00:00	5000.00	NULL
12	3	2001-09-13 00:00:00	1350.00	NULL
13	3	2001-09-13 00:00:00	1750.00	NULL

FULL JOIN и MySQL

Полное внешнее соединение (FULL JOIN) не поддерживается в **MySQL**. Можно считать, что это – «избыточная» операция, т.к. она представляется через объединение левого и правого внешних соединений. Например, запрос

```
1.--(1)--
2.SELECT * FROM Income_o I FULL JOIN Outcome_o O
3.     ON I.point = O.point AND I.date = O.date;
```

который на каждый рабочий день по каждому пункту выводит в одну строку приход и расход (схема «Вторсырье»), можно переписать в виде:

```
1.--(2)--
2.SELECT * FROM Income_o I LEFT JOIN Outcome_o O
3.      ON I.point = O.point AND I.date = O.date
4.UNION
5.SELECT * FROM Income_o I RIGHT JOIN Outcome_o O
6.      ON I.point = O.point AND I.date = O.date;
```

С логической точки зрения эти запросы эквивалентны; оба они выводят как дни, когда был и приход, и расход, так и дни, когда отсутствовала одна из операций (отсутствующие значения заменяются NULL). Однако с точки зрения производительности второй запрос проигрывает первому вдвое по оценке стоимости плана. Это связано с тем, что операция UNION приводит к

выполнению сортировки, которая отсутствует в плане первого запроса. Сортировка же необходима для процедуры исключения дубликатов, т.к. левое и правое соединения оба содержат строки, соответствующие внутреннему соединению, т.е. случаю, когда есть как приход, так и расход. Поэтому, если вместо **UNION** написать **UNION ALL**, то такие строки будут присутствовать в результирующем наборе в двух экземплярах.

Тем не менее, чтобы получить план, близкий по стоимости **FULL JOIN**, нужно избавиться от сортировки. Например, использовать UNION ALL, но в одном из объединяемых запросов исключить строки, соответствующие внутреннему соединению:

```
1. --(3)--
2. SELECT * FROM Income_o I LEFT JOIN Outcome_o O
3.       ON I.point = O.point AND I.date = O.date
4. UNION ALL
5. SELECT NULL, NULL, NULL,* FROM Outcome_o O
6. WHERE NOT EXISTS (SELECT 1 FROM Income_o I
7.                   WHERE I.point = O.point AND I.date =
   O.date);
```

Обратите внимание, что заведомо отсутствующие значения, которые появлялись в правом соединении решения (2), здесь формируются явным заданием NULL-значений. Если по каким-то причинам, явное задание NULL вместо соединения вам не подходит, можно оставить соединение, но это даст более дорогой план, хотя и он будет дешевле плана с сортировкой (2):

```
1. SELECT * FROM Income_o I LEFT JOIN Outcome_o O
2.       ON I.point = O.point AND I.date = O.date
3. UNION ALL
4. SELECT * FROM Income_o I RIGHT JOIN Outcome_o O
5.       ON I.point = O.point AND I.date = O.date
6. WHERE NOT EXISTS (SELECT 1 FROM Income_o I
7.                   WHERE I.point = O.point AND I.date =
   O.date);
```

Декартово произведение

Ранее мы уже рассмотрели реализацию декартова произведения (**пункт 5.6**), состоящую в перечислении через запятую табличных выражений в предложении **FROM** (таблицы, представления, подзапросы) при отсутствии предложения **WHERE**, связывающего столбцы из перечисленных источников строк. Кроме того, можно использовать еще и явную операцию соединения – **CROSS JOIN**, например:

```
1.SELECT Laptop.model, Product.model
2.FROM Laptop CROSS JOIN
3. Product;
```

Напомним, что при декартовом произведении каждая строка из первой таблицы соединяется с каждой строкой второй таблицы. В результате количество строк результирующего набора равно произведению количества строк операндов декартова произведения. В нашем примере таблица Laptop содержит 5 строк, а таблица Product — 16. В результате получается 5 * 16 = 80 строк. Поэтому мы не приводим здесь результат выполнения этого запроса. Вы можете сами проверить это утверждение, нажав ссылку "выполнить" или выполнив приведенный выше запрос с помощью консоли.

Примечание:

В чистом виде декартово произведение практически не используется, оно, как правило, является промежуточным результатом выполнения операции горизонтальной проекции (выборки) при наличии в операторе SELECT предложения WHERE.

Объединение

Для объединения запросов используется служебное слово **UNION**:

```
1.<запрос 1>
2.UNION [ALL]
```

```
3.<запрос 2>
```

Предложение **UNION** приводит к появлению в результирующем наборе всех строк каждого из запросов. При этом, если определен параметр **ALL**, то сохраняются все дубликаты выходных строк, в противном случае в результирующем наборе присутствуют только уникальные строки. Заметим, что можно связывать вместе любое число запросов. Кроме того, с помощью скобок можно задавать порядок объединения.

Операция объединения может быть выполнена только при выполнении следующих условий:

- количество выходных столбцов каждого из запросов должно быть одинаковым;
- выходные столбцы каждого из запросов должны быть совместимы между собой (в порядке их следования) по типам данных;
- в результирующем наборе используются имена столбцов, заданные в первом запросе;
- предложение **ORDER BY** применяется к результату соединения, поэтому оно может быть указано только в конце всего составного запроса.

Пример 5.7.1

Найти номера моделей и цены ПК и портативных компьютеров:

```
1.SELECT model, price
2.FROM PC
3.UNION
4.SELECT model, price
5.FROM Laptop
6.ORDER BY price DESC;
```

model	Price
1750	1200
1752	1150
1298	1050
1233	980

1321	970
1233	950
1121	850
1298	700
1232	600
1233	600
1232	400
1232	350
1260	350

Пример 5.7.2

Найти тип продукции, номер модели и цену ПК и портативных компьютеров:

```
1.SELECT Product.type, PC.model, price
2.FROM PC INNER JOIN
3.     Product ON PC.model = Product.model
4.UNION
5.SELECT Product.type, Laptop.model, price
6.FROM Laptop INNER JOIN
7.     Product ON Laptop.model = Product.model
8.ORDER BY price DESC;
```

Type	Model	price
Laptop	1750	1200
Laptop	1752	1150
Laptop	1298	1050

PC	1233	980
Laptop	1321	970
PC	1233	950
PC	1121	850
Laptop	1298	700
PC	1232	600
PC	1233	600
PC	1232	400
PC	1232	350
PC	1260	350

Рассмотрим следующую задачу.

Найти все имеющиеся единицы продукции производителя 'B'. Вывести номер модели и тип.

В базе имеется один ноутбук и три ПК от производителя B, при этом все три ПК - одной модели.

Если мы будем использовать объединение с помощью **UNION ALL**, то мы получим все эти изделия.

```
1. SELECT p.model, p.type FROM pc JOIN Product p ON PC.model=p.model WHERE
   maker='B'
2. UNION ALL
3. SELECT p.model, p.type FROM printer pr JOIN Product p ON pr.model=p.model
   WHERE maker='B'
4. UNION ALL
5. SELECT p.model, p.type FROM laptop lp JOIN Product p ON lp.model=p.model
   WHERE maker='B';
```

model	type
1121	PC
1121	PC
1121	PC
1750	Laptop

А если - UNION, то из результата будут исключены дубликаты строк:

```
1. SELECT p.model, p.type FROM pc JOIN Product p ON
   PC.model=p.model WHERE maker='B'
2. UNION
3. SELECT p.model, p.type FROM printer pr JOIN Product p ON
   pr.model=p.model WHERE maker='B'
4. UNION
5. SELECT p.model, p.type FROM laptop lp JOIN Product p ON
   lp.model=p.model WHERE maker='B';
```

model	type
1121	PC
1750	Laptop

Решение на основе UNION соответствует несколько иной задаче, которую можно было бы сформулировать следующим образом.

Выяснить, какие модели производителя 'B' имеются в наличии. Вывести номер модели и тип.

Пересечение и разность

В стандарте языка SQL имеются предложения оператора **SELECT** для выполнения операций пересечения и разности результатов запросов-операндов. Этими предложениями являются **INTERSECT [ALL]** (пересечение) и **EXCEPT [ALL]** (разность), которые работают аналогично предложению **UNION**. В результирующий набор попадают только те строки, которые присутствуют в обоих запросах (INTERSECT) или только те строки первого запроса, которые отсутствуют во втором (EXCEPT). При этом оба запроса, участвующих в операции, должны иметь одинаковое число столбцов, и соответствующие столбцы должны иметь одинаковые (или неявно приводимые) типы данных. Имена столбцов результирующего набора формируются из заголовков первого запроса.

Если не используется ключевое слово **ALL** (по умолчанию подразумевается DISTINCT), то при выполнении операции автоматически устраняются дубликаты строк. Если указано ALL, то количество дублированных строк подчиняется следующим правилам (n1 - число дубликатов строк первого запроса, n2 - число дубликатов строк второго запроса):

- **INTERSECT ALL**: min(n1, n2)
- **EXCEPT ALL**: n1 - n2, если n1>n2.

Пример 5.7.3.

Найти корабли, которые присутствуют как в таблице Ships, так и в таблице Outcomes.

```
1.SELECT name FROM Ships
2.INTERSECT
3.SELECT ship FROM Outcomes;
```

В реляционной алгебре операция пересечения является коммутативной, поскольку она применима к отношениям с одинаковыми заголовками. Мы и в SQL можем поменять запросы местами. Вышеприведенное решение даст тот же результат, что и следующее:

```
1.SELECT ship FROM Outcomes
2.INTERSECT
3.SELECT name FROM Ships;
```

за исключением заголовка. В первом случае единственный столбец будет иметь заголовок name, а во втором - ship. Поэтому запрос

```
1.SELECT name FROM (
2.SELECT ship FROM Outcomes
3.INTERSECT
4.SELECT name FROM Ships
5. ) x;
```

приведет к ошибке:

Invalid column name 'name'.

(неверное имя столбца 'name').

Пример 5.7.4

Найти корабли из таблицы Outcomes, которые отсутствуют в таблице Ships.

Задача легко решается при помощи оператора **EXCEPT**:

```
1.SELECT ship FROM Outcomes
2.EXCEPT
3.SELECT name FROM Ships;
```

Операция разности не является коммутативной, поэтому если переставить местами запросы, то мы получим решение совсем другой задачи:

"Найти корабли из таблицы Ships, которые отсутствуют в таблице Outcomes".

Эта задача на языке предметной области звучит так: "Найти корабли, которые не принимали участие в сражениях".

Заметим, что если какой-либо корабль принимал участие в сражениях несколько раз, то по причине исключения дубликатов он будет присутствовать только один раз в результирующем наборе. У нас есть такой корабль - California, но он присутствует также и в таблице Ships, а потому не выводится рассмотренным выше запросом. Поэтому, чтобы продемонстрировать сказанное, исключим его из результатов второго запроса в операции разности:

```
1.SELECT ship FROM Outcomes
2.EXCEPT
3.SELECT name FROM Ships WHERE name <> 'California';
```

ship
Bismarck
California
Duke of York
Fuso
Hood
King George V
Prince of Wales
Rodney
Schamhorst
West Virginia
Yamashiro

Для имеющегося набора данных тот же результат мы получим при выполнении следующего запроса:

```
1. SELECT ship FROM Outcomes
2. EXCEPT ALL
3. SELECT name FROM Ships;
```

(2 дубликата для 'California' в таблице Outcomes минус 1 - в Ships)

Соответственно, запрос

```
1. SELECT ship FROM Outcomes
2. EXCEPT ALL
3. SELECT name FROM Ships WHERE name<>'California';
```

даст нам два вхождения корабля 'California' в результирующем наборе (2 - 0 = 2):

ship
Bismarck
California
California
Duke of York
Fuso
Hood
King George V
Prince of Wales
Rodney
Schamhorst
West Virginia
Yamashiro

Следует сказать, что не все СУБД поддерживают эти предложения в операторе SELECT. Нет поддержки INTERSECT/EXCEPT, например, в **MySQL**, а в **MS SQL Server** она появилась, лишь начиная с версии 2005, и то без ключевого слова ALL. ALL с INTERSECT/EXCEPT также еще не реализована в **Oracle**. Следует отметить, что вместо стандартного EXCEPT в Oracle используется ключевое слово **MINUS**.

Поэтому для выполнения операций пересечения и разности могут быть использованы другие средства. Здесь уместно заметить, что один и тот же результат можно получить с помощью различных формулировок оператора SELECT. В случае пересечения и разности можно воспользоваться предикатом существования **EXISTS**.

В заключение рассмотрим пример на использование операции **INTERSECT ALL**.

Пример 5.7.5

Найти производителей, которые выпускают не менее двух моделей ПК и не менее двух моделей принтеров.

```
1.SELECT maker FROM (
2.SELECT maker FROM Product WHERE type='PC'
3.INTERSECT ALL
4.SELECT maker FROM Product WHERE type ='Printer'
5.) X GROUP BY maker HAVING COUNT(*)>1;
```

INTERSECT ALL в подзапросе этого решения оставит минимальное число дубликатов, т.е. если производитель выпускает 2 модели ПК и одну модель принтера (или наоборот), то он будет присутствовать в результирующем наборе один раз. Далее мы выполняем группировку по производителю, оставляя только тех из них, кто присутствует в результатах подзапроса более одного раза.

Конечно, мы можем решить эту задачу, не используя явно операцию пересечения. Например, одним подзапросом найдем производителей, которые выпускают не менее 2-х моделей ПК, другим - тех, кто выпускает не менее 2-х моделей принтеров. Решение задачи даст внутреннее соединение этих подзапросов. Ниже этот алгоритм реализован на основе еще одного стандартного типа соединений - **естественного соединения**:

```
1. SELECT PC.maker FROM (
2. SELECT maker FROM Product
3. WHERE type='PC' GROUP BY maker
4. HAVING COUNT(*)>1) PC
5. NATURAL JOIN
6. (
7. SELECT maker FROM Product
8. WHERE type='Printer' GROUP BY maker
9. HAVING COUNT(*)>1) Pr;
```

Естественное соединение (**NATURAL JOIN**) – это эквисоединение по столбцам с одинаковыми именами. SQL Server не поддерживает этот тип соединения, поэтому последний запрос можно выполнить, например, с помощью **PostgreSQL**.

Тот факт, что операция **EXCEPT** убирает из результата строки-дубликаты, даёт нам еще один способ исключения дубликатов. Продемонстрируем имеющиеся варианты на примере следующей задачи (используется база данных "Окраска").

Перечислить цвета имеющихся баллончиков с краской.

1. Использование DISTINCT

```
1. SELECT DISTINCT v_color FROM utV;
```

2. Использование GROUP BY

```
1. SELECT v_color FROM utV GROUP BY v_color;
```

3. Использование EXCEPT

Идея решения состоит в том, чтобы "вычесть" из имеющегося набора несуществующий цвет, например, 'Z':

```
1. SELECT v_color FROM utV
2. EXCEPT
3. SELECT 'Z';
```

Поскольку столбец v_color не допускает NULL-значений, последний запрос можно переписать в универсальной форме:

```
1. SELECT v_color FROM utV
```

```
2. EXCEPT
3. SELECT NULL;
```

SQL Server оценивает стоимость всех этих запросов равной. В этом нет ничего удивительного в силу того, что каждый из запросов выполняет чтение таблицы и сортировку, а это наиболее "тяжелые" операции плана запроса.

В связи с упражнением 6 (SELECT) рейтингового этапа возник вопрос относительно старшинства операций **UNION**, **EXCEPT** и **INTERSECT**. Логический порядок выполнения этих операций, который приводится в книге Мартина Грабера [4] "Справочное руководство по SQL", выглядит так:

- **UNION, EXCEPT**
- **INTERSECT**

В предположении, что логический порядок выполнения операций соответствует их старшинству, получается, что старшинство операций UNION и EXCEPT идентично и, следовательно, они должны выполняться в том порядке, в котором записаны, если этот порядок не изменяется скобками. При этом обе операции выполняются раньше, чем INTERSECT, т.е. они старше.

Рассмотрим три простых запроса, которые будем комбинировать различными способами, чтобы убедиться в этом:

```
1. --Модели и типы продукции производителя B
2. SELECT model, type FROM Product WHERE maker='B';
```

model	type
1121	PC
1750	Laptop

```
1. --Модели ноутбуков
2. SELECT model, type FROM Product WHERE type='Laptop';
```

model	type
1298	Laptop
1321	Laptop
1750	Laptop
1752	Laptop

```
1. --Модели ПК
2. SELECT model, type FROM Product WHERE type='PC';
```

model	type
1121	PC
1232	PC
1233	PC
1260	PC
2111	PC
2112	PC

Давайте сначала проверим первое утверждение. Если операция EXCEPT старше операции UNION, то запросы

```
1. SELECT model, type FROM Product WHERE maker='B'
2. UNION
3. SELECT model, type FROM Product WHERE type='Laptop'
4. EXCEPT
5. SELECT model, type FROM Product WHERE type='PC';
```

И

```
1. (SELECT model, type FROM Product WHERE maker='B'
```

```
2.UNION
3.SELECT model, type FROM Product WHERE type='Laptop')
4.EXCEPT
5.SELECT model, type FROM Product WHERE type='PC';
```

должны нам дать разные результаты. Однако это не так, и мы получаем один и тот же результирующий набор:

model	type
1298	Laptop
1321	Laptop
1750	Laptop
1752	Laptop

Аналогично, если операция UNION старше операции EXCEPT, то запросы

```
1.SELECT model, type FROM Product WHERE type='Laptop'
2.EXCEPT
3.SELECT model, type FROM Product WHERE type='PC'
4.UNION
5.SELECT model, type FROM Product WHERE maker='B';
```

И

```
1.(SELECT model, type FROM Product WHERE type='Laptop'
2.EXCEPT
3.SELECT model, type FROM Product WHERE type='PC')
4.UNION
5.SELECT model, type FROM Product WHERE maker='B';
```

должны нам дать разные результаты. И тут мы получаем одинаковый результат:

model	type
1121	PC

1298	Laptop
1321	Laptop
1750	Laptop
1752	Laptop

Итак, операции UNION и EXCEPT эквивалентны по старшинству.

Проверим теперь старшинство операции INTERSECT по отношению к другим операторам (в тестах можно взять любую из них, т.к. они имеют один и тот же порядок).

Если INTERSECT "младше" или эквивалентен UNION, то запросы

```
1.SELECT model, type FROM Product WHERE maker='B'
2.UNION
3.SELECT model, type FROM Product WHERE type='Laptop'
4.INTERSECT
5.SELECT model, type FROM Product WHERE type='PC';
```

и

```
1.(SELECT model, type FROM Product WHERE maker='B'
2.UNION
3.SELECT model, type FROM Product WHERE type='Laptop')
4.INTERSECT
5.SELECT model, type FROM Product WHERE type='PC';
```

должны дать одинаковые результаты. Однако мы получаем разные результирующие наборы. Первый запрос дает

model	type
1121	PC
1750	Laptop

в то время как второй

model	type
1121	PC

Вывод. Логический порядок, приведенный в начале статьи не соответствует старшинству операций, и, на мой взгляд, его следует поменять на обратный:

- **INTERSECT**
- **UNION, EXCEPT**

Предикат EXISTS

Синтаксис:

```
1.EXISTS::=
2. [NOT] EXISTS (<табличный подзапрос>)
```

Предикат **EXISTS** принимает значение **TRUE**, если подзапрос содержит любое количество строк, иначе его значение равно **FALSE**. Для **NOT EXISTS** все наоборот. Этот предикат никогда не принимает значение **UNKNOWN**.

Обычно предикат **EXISTS** используется в зависимых (коррелирующих) подзапросах. Этот вид подзапроса имеет внешнюю ссылку, связанную со значением в основном запросе. Результат подзапроса может зависеть от этого значения и должен оцениваться отдельно для каждой строки запроса, в котором содержится данный подзапрос. Поэтому предикат **EXISTS** может иметь разные значения для разных строк основного запроса.

Пример на пересечение.

Найти тех производителей портативных компьютеров, которые также производят принтеры:

```
1.SELECT DISTINCT maker
2.FROM Product AS lap_product
3.WHERE type = 'laptop' AND
4. EXISTS (SELECT maker
5. FROM Product
6. WHERE type = 'printer' AND
7. maker = lap_product.maker
8. );
```

В подзапросе выбираются производители принтеров и сравниваются с производителем, значение которого передается из основного запроса. В основном же запросе отбираются производители портативных компьютеров. Таким образом, для каждого производителя портативных компьютеров проверяется, возвращает ли подзапрос строки (которые говорят о том, что этот производитель также выпускает принтеры). Поскольку два условия в предложении **WHERE** должны выполняться одновременно (**AND**), то в результирующий набор попадут нужные нам строки. **DISTINCT** используется для того, чтобы каждый производитель присутствовал в выходных данных только один раз. В результате получим

maker
A

Пример на разность.

Найти производителей портативных компьютеров, которые не производят принтеров:

```
1.SELECT DISTINCT maker
2.FROM Product AS lap_product
```

```
3.WHERE type = 'laptop' AND
4. NOT EXISTS (SELECT maker
5. FROM Product
6. WHERE type = 'printer' AND
7. maker = lap_product.maker
8. );
```

В этом случае достаточно заменить в предыдущем примере **EXISTS** на **NOT EXISTS**. То есть выходные данные составят только те уникальные строки основного запроса, для которых подзапрос не возвращает ни одной строки. В итоге получим:

maker
B
C

Реляционное деление

Рассмотрим следующую задачу.

Определить производителей, которые выпускают модели всех типов (схема "Компьютерная фирма").

Ключевым словом здесь является "всех", т.е. производитель в таблице Product должен иметь модели каждого типа, т.е. и PC, и Laptop, и Printer.

Как раз для решения подобных задач в реляционную алгебру Коддом была введена специальная операция реляционного деления (DIVIDE BY).

С помощью этой операции наша задача решается очень просто:

```
1.Product[maker, type] DIVIDE BY Product[type]
```

Здесь квадратными скобками обозначается операция взятия проекции на соответствующие атрибуты.

Операция реляционного деления избыточна, т.е. она может быть выражена через другие операции реляционной алгебры. Возможно, поэтому ее нет в языке SQL.

На примере решения сформулированной задачи я хочу показать несколько приемов реализации операции реляционного деления на языке SQL.

Группировка

Если использовать тот факт, что, согласно описанию предметной области, типов продукции всего три, то мы можем выполнить группировку по производителю и подсчитать количество уникальных типов. Затем мы отберем только тех производителей, у которых это число равно трем.

Итак,

```
1.SELECT maker
2.FROM Product
3.GROUP BY maker
4.HAVING COUNT(DISTINCT type) = 3;
```

Однако если число типов продукции произвольно, то такое решение будет правильным только при нынешнем состоянии базы данных, а не при любом возможном. А это значит, что мы должны константу заменить "переменной", значением которой будет текущее число типов, т.е. подзапросом:

```
1.SELECT maker
2.FROM Product
3.GROUP BY maker
4.HAVING COUNT(DISTINCT type) =
5.             (SELECT COUNT(DISTINCT type) FROM Product);
```

Разность

Если взять операцию разности ВСЕХ имеющихся типов моделей и типов у конкретного производителя, то результирующая выборка не должна содержать строк.

```
1.SELECT DISTINCT maker
2.FROM Product Pr1
```

```
3.WHERE 0 = (SELECT COUNT(*) FROM
4.(SELECT type FROM Product
5. EXCEPT
6. SELECT type FROM Product Pr2
7. WHERE Pr2.maker = Pr1.maker
8. ) X );
```

Этот запрос можно написать короче, если воспользоваться тем **свойством**, что истинностное значение предиката **ALL** есть TRUE, если подзапрос не возвращает строк:

```
1.SELECT DISTINCT maker
2.FROM Product Pr1
3.WHERE type = ALL
4.(SELECT type FROM Product
5. EXCEPT
6. SELECT type FROM Product Pr2
7. WHERE Pr2.maker = Pr1.maker
8. );
```

Для искомых производителей список типов в предикате ALL будет пуст (предикат равен TRUE). В остальных случаях он будет содержать типы моделей, отсутствующие у производителя из внешнего запроса, поэтому операция сравнения (равенство "=") для всех его моделей даст FALSE.

Существование

Не должно существовать такого типа продукции, которого бы не было у искомого производителя.

```
1.SELECT DISTINCT maker
2.FROM Product Pr1
3.WHERE NOT EXISTS
4.  (SELECT type
5.    FROM Product
6.    WHERE type NOT IN
7.                (SELECT type
8.                  FROM Product Pr2
9.                  WHERE Pr1.maker = Pr2.maker
```

```
10.                   )
11.        );
```

Кроме первого варианта с группировкой все остальные решения используют коррелирующие подзапросы для определения множества типов моделей производителя из основного запроса.

Следует также отметить, что решение с группировкой не подойдет для случая, когда требуется выполнить деление не на все множество имеющихся типов, а на некоторое их подмножество. Например, если требуется найти производителей, у которых множество типов включает в себя (или совпадает) множество типов, определяемое некоторыми критериями. Другие же приемы можно адаптировать для решения подобной задачи.

Использование ключевых слов SOME (ANY) и ALL с предикатами сравнения

```
1.<выражение> <оператор сравнения> SOME | ANY (<подзапрос>)
```

SOME и **ANY** являются синонимами, то есть может использоваться любое из них. Результатом подзапроса является один столбец величин. Если хотя бы для одного значения V, получаемого из подзапроса, результат операции "*<значение выражения> <оператор сравнения> V*" равняется **TRUE**, то предикат **ANY** также равняется **TRUE**.

```
1.<выражение> <оператор сравнения> ALL (<подзапрос>)
```

Исполняется так же, как и **ANY**, однако значение предиката **ALL** будет истинным, если для всех значений V, получаемых из подзапроса, предикат "*<значение выражения> <оператор сравнения> V*" дает **TRUE**.

Пример 5.8.1.

Найти поставщиков компьютеров, моделей которых нет в продаже (то есть модели этих поставщиков отсутствуют в таблице PC)

```
1.SELECT DISTINCT maker
2.FROM Product
3.WHERE type = 'pc' AND
```

```
4.  NOT model = ANY (SELECT model
5.  FROM PC
6.  );
```

Оказалось, что только у поставщика E есть модели, отсутствующие в продаже:

Maker
E

Рассмотрим подробно этот пример. Предикат

```
1.model = ANY (SELECT model
2.  FROM PC
3.  );
```

вернет значение **TRUE**, если модель, определяемая полем model основного запроса, найдется в списке моделей таблицы PC (возвращаемом подзапросом). Поскольку предикат используется в запросе с отрицанием **NOT**, то значение **TRUE** будет получено, если модели не окажется в списке. Этот предикат проверяется для каждой записи основного запроса, которыми являются все модели ПК (предикат type = 'pc') из таблицы Product. Результирующий набор состоит из одного столбца — имени производителя. Чтобы один производитель не выводился несколько раз (что может случиться, если он производит несколько моделей, отсутствующих в таблице PC), используется служебное слово **DISTINCT**, исключающее дубликаты.

Пример 5.8.2.

Найти модели и цены портативных компьютеров, стоимость которых превышает стоимость любого ПК

```
1.SELECT DISTINCT model, price
2.FROM Laptop
3.WHERE price > ALL (SELECT price
4.  FROM PC
5.  );
```

model	Price
1298	1050
1750	1200
1752	1150

Приведем формальные правила оценки истинности предикатов, использующих параметры ANY|SOME и ALL.

- Если определен параметр ALL или SOME и все результаты сравнения значения выражения и каждого значения, полученного из подзапроса, являются TRUE, истинностное значение равно TRUE.
- Если результат выполнения подзапроса не содержит строк и определен параметр ALL, результат равен TRUE. Если же определен параметр SOME, результат равен FALSE.
- Если определен параметр ALL и результат сравнения значения выражения хотя бы с одним значением, полученным из подзапроса, является FALSE, истинностное значение равно FALSE.
- Если определен параметр SOME и хотя бы один результат сравнения значения выражения и значения, полученного из подзапроса, является TRUE, истинностное значение равно TRUE.
- Если определен параметр SOME и каждое сравнение значения выражения и значений, полученных из подзапроса, равно FALSE, истинностное значение тоже равно FALSE.
- В любом другом случае результат будет равен **UNKNOWN**.

Еще раз о подзапросах

Заметим, что в общем случае запрос возвращает множество значений. Поэтому использование подзапроса в предложении **WHERE** без предикатов **EXISTS**, **IN**, **ALL** и **ANY**, которые дают булево значение, может привести к ошибке времени выполнения запроса.

Пример 5.8.3

Найти модели и цены ПК, стоимость которых превышает минимальную стоимость портативных компьютеров:

```
1.SELECT DISTINCT model, price
2.FROM PC
3.WHERE price > (SELECT MIN(price)
4. FROM Laptop
5. );
```

Этот запрос вполне корректен, так как скалярное значение price сравнивается с подзапросом, который возвращает единственное значение. В результате получим четыре модели ПК:

model	price
1121	850
1233	950
1233	970
1233	980

Однако, если в ответ на вопрос «найти модели и цены ПК, стоимость которых совпадает со стоимостью портативных компьютеров» написать следующий запрос

```
1.SELECT DISTINCT model, price
2.FROM PC
3.WHERE price = (SELECT price
4. FROM Laptop
5. );
```

то при выполнении последнего мы можем получить такое сообщение об ошибке:

Subquery returned more than 1 value. This is not permitted when the subquery follows =, !=, <, <= , >, >= or when the subquery is used as an expression.

(«Подзапрос вернул более одного значения. Это не допускается в тех случаях, когда подзапрос следует после =, !=, <, <=, >, >= или когда подзапрос используется в качестве выражения».)

Эта ошибка будет возникать при сравнении скалярного значения с подзапросом, который возвращает более одного значения.

Подзапросы, в свою очередь, также могут содержать вложенные запросы.

С другой стороны, подзапрос, возвращающий множество строк и содержащий несколько столбцов, вполне естественно может использоваться в предложении **FROM**. Это, например, позволяет ограничить набор столбцов и/или строк при выполнении операции соединения таблиц.

Пример 5.8.4

Вывести производителя, тип, модель и частоту процессора для Портативных компьютеров, частота процессора которых превышает 600 МГц.

Этот запрос может быть сформулирован, например, следующим образом:

```
1. SELECT prod.maker, lap.*
2. FROM (SELECT 'laptop' AS type, model, speed
3.  FROM laptop
4.  WHERE speed > 600
5.  ) AS lap INNER JOIN
6.  (SELECT maker, model
7.  FROM product
8.  ) AS prod ON lap.model = prod.model;
```

В результате получим:

maker	type	model	speed
B	laptop	1750	750
A	laptop	1752	750

Наконец, подзапросы могут присутствовать в предложении **SELECT**. Это иногда позволяет весьма компактно сформулировать запрос.

Пример 5.8.5

Найти разницу между средними значениями цены портативных компьютеров и ПК, то есть насколько в среднем портативный компьютер стоит дороже, чем ПК.

Здесь вообще можно обойтись одним предложением **SELECT** в основном запросе:

```
1. SELECT (SELECT AVG(price)
2.  FROM Laptop
3.  ) -
4.  (SELECT AVG(price)
5.  FROM PC
6.  ) AS dif_price;
```

В результате получим

dif_price
328.3333

Преобразование типов и оператор CAST

В реализациях языка **SQL** может быть выполнено неявное преобразование типов. Так, например, в SQL Server и Sybase ASE.Transact-SQL при сравнении или комбинировании значений типов smallint и int, данные типа smallint неявно преобразуются к типу int. Подробно о явном и неявном преобразовании типов в SQL Server можно прочитать в BOL.

Пример 5.9.1.

Вывести среднюю цену портативных компьютеров с предваряющим текстом «средняя цена = ».

Попытка выполнить запрос

```
1. SELECT 'Средняя цена = '+AVG(price)
2. FROM Laptop;
```

приведет к сообщению об ошибке:

Implicit conversion from data type varchar to money is not allowed. Use the CONVERT function to run this query.

(«Не допускается неявное преобразование типа varchar к типу money. Используйте для выполнения этого запроса функцию **CONVERT**».)

Это сообщение означает, что система не может выполнить неявное преобразование типа varchar к типу money. В подобных ситуациях может помочь явное преобразование типов. При этом, как указано в сообщении об ошибке, можно воспользоваться функцией **CONVERT**. Однако эта функция не стандартизована, поэтому в целях переносимости рекомендуется использовать стандартное выражение **CAST**. С него и начнем.

Итак, если переписать наш запрос в виде

```
1. SELECT 'Средняя цена = '+ CAST(AVG(price) AS CHAR(15))
2. FROM Laptop;
```

в результате получим то, что требовалось:

```
1. Средняя цена = 1003.33
```

Мы использовали выражение явного преобразования типов CAST для приведения среднего значения цены к строковому представлению.

Синтаксис выражения **CAST** очень простой

```
1. CAST(<выражение> AS <тип данных>)
```

***Внимание*:**

Следует иметь в виду, во-первых, что не любые преобразования типов возможны (стандарт содержит таблицу допустимых преобразований типов данных). Во-вторых, результат функции CAST для значения выражения, равного NULL, тоже будет NULL.

Рассмотрим еще один пример.

Пример 5.9.2

Определить средний год спуска на воду кораблей из таблицы Ships.

Запрос:

```
1.SELECT AVG(launched)
2.FROM Ships;
```

даст результат 1926. В принципе все правильно, так как мы получили в результате то, что просили — год. Однако среднее арифметическое будет составлять примерно 1926,9091. Тут следует напомнить, что агрегатные функции (за исключением функции **COUNT**, которая всегда возвращает целое число) наследуют тип данных обрабатываемых значений. Поскольку поле launched — целочисленное, мы и получили среднее значение с отброшенной дробной частью (заметьте — не округленное).

А если нас интересует результат с заданной точностью, скажем, до двух десятичных знаков? Применение выражения **CAST** к среднему значению ничего не даст по указанной выше причине. Действительно,

```
1.SELECT CAST(AVG(launched) AS NUMERIC(6,2))
2.FROM Ships;
```

вернет значение 1926.00. Следовательно, **CAST** нужно применить к аргументу агрегатной функции:

```
1.SELECT AVG(CAST(launched AS NUMERIC(6,2)))
2.FROM Ships;
```

Результат — 1926.90909. Опять не то. Причина состоит в том, что при вычислении среднего значения было выполнено неявное преобразование типа. Сделаем еще один шаг:

```
1.SELECT CAST(AVG(CAST(launched AS NUMERIC(6,2))) AS
  NUMERIC(6,2))
2.FROM Ships;
```

В результате получим то, что нужно — 1926.91. Однако это решение выглядит очень громоздко. Заставим неявное преобразование типа поработать на нас:

```
1.SELECT CAST(AVG(launched*1.0) AS NUMERIC(6,2))
```

```
2.FROM Ships;
```

Теперь мы использовали неявное преобразование целочисленного аргумента к точному числовому типу (EXACT NUMERIC), умножив его на вещественную единицу, после чего применили явное приведения типа результата агрегатной функции.

Аналогичные преобразования типа можно выполнить с помощью функции SQL Server **CONVERT**:

```
1.SELECT CONVERT(NUMERIC(6,2),AVG(launched*1.0))
2.FROM Ships;
```

Функция **CONVERT** имеет следующий синтаксис:

```
1.CONVERT    (<тип_данных[(<длина>)]>,    <выражение>    [,
  <стиль>])
```

Основное отличие функции **CONVERT** от функции **CAST** состоит в том, что первая позволяет форматировать данные (например, темпоральные данные типа datetime) при преобразовании их к символьному типу и указывать формат при обратном преобразовании. Разные целочисленные значения необязательного аргумента стиль соответствуют различным типам форматов. Рассмотрим следующий пример

```
1.SELECT CONVERT(char(25), CONVERT(datetime,'20030722'));
```

Здесь мы преобразуем строковое представление даты к типу datetime, после чего выполняем обратное преобразование, чтобы продемонстрировать результат форматирования. Поскольку значение аргумента стиль не задано используется значение по умолчанию (0 или 100). В результате получим:

Jul 22 2003 12:00AM

Ниже приведены некоторые другие значения аргумента стиль и результат, полученный на приведенном выше примере. Заметим, что увеличение значения стиль на 100 приводит к четырехзначному отображению года.

1	07/22/03
11	03/07/22
3	22/07/03
121	2003-07-22 00:00:00.000

Перечень всех возможных значений аргумента стиль можно посмотреть в BOL.

Есть одна особенность использования оператора **CAST** в SQL Server, связанная с преобразованием числа к его строковому представлению. Что произойдет, если число символов в числе превышает размер строки? Например,

```
1.SELECT CAST(1234.6 AS VARCHAR(5));
```

Следует ожидать, что мы получим сообщение об ошибке. Правильно, вот это сообщение:

Arithmetic overflow error converting numeric to data type varchar.

(«Ошибка арифметического переполнения при преобразовании числа к типу данных VARCHAR».)

Естественно, что мы будем ожидать того же сообщения и при выполнении следующего оператора:

```
1.SELECT CAST(123456 AS VARCHAR(5));
```

Но нет. В результате мы получим символ «*» вместо сообщения об ошибке. Мы не беремся судить, с чем это связано, однако, однажды мы столкнулись с проблемой диагностики ошибки в коде, в котором впоследствии выполнялось обратное преобразование к числовому типу.

В нашем простейшем примере это будет выглядеть так:

```
1. SELECT CAST(CAST(123456 AS VARCHAR(5)) AS INT);
```

Вот тут-то мы и получаем ошибку:

Syntax error converting the varchar value '' to a column of data type int.*

(«Ошибка синтаксиса при преобразовании значения «*» к типу данных INT».)

Примечание:

Функция SQL Server и Sybase ASE.Transact-SQL CONVERT ведет себя аналогичным образом.

Преобразование типа money

Денежный тип данных не является стандартным. В SQL Server имеется два денежных типа:

money: диапазон значений от –922,337,203,685,477.5808 до 922,337,203,685,477.5807

smallmoney: диапазон значений от -214 748,3648 до 214 748,3647

Точность обоих типов одна десятитысячная.

Константу типа money можно задать с помощью префикса $, или же использовать преобразование типов, например:

```
1. SELECT 1.2 num, $1.2 mn1, CAST(1.2 AS MONEY) mn2;
```

num	mn1	mn2
1.2	1,20	1,20

Обратите внимание на запятую в качестве разделителя "рублей" и "копеек" - не точка!

Преобразование к целому типу для чисел и денег выполняется по разному: в первом случае дробная часть отбрасывается, во втором происходит округление.

```
1. SELECT CAST(1.75 AS INT) int_num, CAST($1.75 AS INT)
   int_mon;
```

int_num	int_mon
1	2

Деньги таки, их просто так терять нельзя!

Оператор CASE

Пусть требуется вывести список всех моделей ПК с указанием их цены. При этом если модель отсутствует в продаже (ее нет в таблице PC), то вместо цены вывести текст «Нет в наличии».

Список всех моделей ПК с ценами можно получить с помощью запроса:

```
1.SELECT DISTINCT Product.model, price
2.FROM Product LEFT JOIN
3. PC ON Product.model = PC.model
4.WHERE product.type = 'pc';
```

В результирующем наборе отсутствующая цена будет заменена **NULL**-значением:

Model	price
1121	850
1232	350
1232	400
1232	600
1233	600
1233	950

1233	980
1260	350
2111	NULL
2112	NULL

Чтобы заменить **NULL**-значения нужным текстом, можно воспользоваться оператором **CASE**:

```
1.SELECT DISTINCT product.model,
2. CASE
3. WHEN price IS NULL
4. THEN 'Нет в наличии'
5. ELSE CAST(price AS CHAR(20))
6. END price
7.FROM Product LEFT JOIN
8. PC ON Product.model = PC.model
9.WHERE product.type = 'pc';
```

Оператор **CASE** в зависимости от указанных условий возвращает одно из множества возможных значений. В нашем примере условием является проверка на **NULL**. Если это условие выполняется, то возвращается текст «Нет в наличии», в противном случае (ELSE) возвращается значение цены.

Здесь есть один принципиальный момент. Поскольку результатом оператора **SELECT** всегда является таблица, то все значения любого столбца должны иметь один и тот же тип данных (с учетом неявного приведения типов). Поэтому мы не можем наряду с ценой (числовой тип) выводить символьную константу. Вот почему к полю price применяется преобразование типов, чтобы привести его значения к символьному представлению. В результате получим:

model	price
1121	850
1232	350
1232	400

1232	600
1233	600
1233	950
1233	980
1260	350
2111	Нет в наличии
2112	Нет в наличии

Оператор **CASE** может быть использован в одной из двух синтаксических форм записи:

1-я форма:

```
1.CASE <проверяемое выражение>
2. WHEN <сравниваемое выражение 1>
3. THEN <возвращаемое значение 1>
4. …
5. WHEN <сравниваемое выражение N>
6. THEN <возвращаемое значение N>
7. [ELSE <возвращаемое значение>]
8.END
```

2-я форма:

```
1.CASE
2. WHEN <предикат 1>
3. THEN <возвращаемое значение 1>
4. …
5. WHEN <предикат N>
6. THEN <возвращаемое значение N>
7. [ELSE <возвращаемое значение>]
8.END
```

Все предложения **WHEN** должны иметь одинаковую синтаксическую форму, то есть нельзя смешивать первую и вторую формы. При использовании первой синтаксической формы условие **WHEN** удовлетворяется, как только значение проверяемого выражения станет равным значению выражения,

указанного в предложении **WHEN**. При использовании второй синтаксической формы условие **WHEN** удовлетворяется, как только предикат принимает значение **TRUE**. При удовлетворении условия оператор **CASE** возвращает значение, указанное в соответствующем предложении **THEN**. Если ни одно из условий **WHEN** не выполнилось, то будет использовано значение, указанное в предложении **ELSE**. При отсутствии **ELSE**, будет возвращено **NULL**-значение. Если удовлетворены несколько условий, то будет возвращено значение предложения **THEN** первого из них, так как остальные просто не будут проверяться.

В приведенном выше примере была применена вторая форма оператора **CASE**.

Заметим, что для проверки на **NULL** стандарт предлагает более короткую форму — оператор **COALESCE**. Он имеет произвольное число параметров и возвращает значение первого из них, отличного от **NULL**. Для двух параметров оператор *COALESCE(A, B)* эквивалентен следующему оператору **CASE**:

```
1.CASE
2. WHEN A IS NOT NULL
3. THEN A
4. ELSE B
5.END
```

Решение рассмотренного выше примера при использовании оператора **COALESCE** можно переписать следующим образом:

```
1.SELECT DISTINCT Product.model,
2. COALESCE(CAST(price AS CHAR(20)),'Нет в наличии') price
3.FROM Product LEFT JOIN
4. PC ON Product.model = PC.model
5.WHERE Product.type = 'pc';
```

Применение первой синтаксической формы оператора **CASE** можно продемонстрировать на следующем примере.

Пример 5.10.1

Вывести все имеющиеся модели ПК с указанием цены. Отметить самые дорогие и самые дешевые модели.

```
1.SELECT DISTINCT model, price,
```

```
2. CASE price
3. WHEN (SELECT MAX(price)
4. FROM PC
5. )
6. THEN 'Самый дорогой'
7. WHEN (SELECT MIN(price)
8. FROM PC
9. )
10.       THEN 'Самый дешевый'
11.       ELSE 'Средняя цена'
12.       END comment
13.      FROM PC
14.      WHERE price IS NOT NULL
15.      ORDER BY price;
```

В результате выполнения запроса получим:

model	price	comment
1232	350.0	Самый дешевый
1260	350.0	Самый дешевый
1232	400.0	Средняя цена
1232	600.0	Средняя цена
1233	600.0	Средняя цена
1121	850.0	Средняя цена
1233	950.0	Средняя цена
1233	980.0	Самый дорогой

Оператор CASE может быть использован не только в предложении SELECT. Здесь вы можете найти другие примеры его использования.

Рассмотрим еще несколько примеров.

Посчитать количество рейсов из Ростова в Москву, и количество рейсов, выполняемых в остальные города.

Здесь мы можем воспользоваться вычисляемым столбцом, по значениям которого будем выполнять группировку:

```
1.SELECT flag, COUNT(*) qty FROM
2.(SELECT CASE WHEN town_to ='Moscow' THEN 'Moscow' ELSE
  'Other' END flag
3.FROM Trip
4.WHERE town_from='Rostov'
5.) X
6.GROUP BY flag;
```

flag	qty
Moscow	4
Other	2

Посчитать общее количество рейсов из Ростова и количество рейсов, пунктом назначения которых не является Москва.

В этой задаче тоже требуется выполнить агрегацию по двум выборкам, при этом одна из выборок является подмножеством второй. Поэтому здесь напрямую не подойдёт вычисляемый столбец, по которому можно выполнить группировку. Это годилось для решения предыдущей задачи, когда множество делилось на собственные непересекающиеся подмножества, по каждому из которых требовалось выполнить агрегацию.

Для решения данной задачи мы можем посчитать количество по всему множеству и использовать подзапрос для подсчета значений в подмножестве (второе обращение к таблице) или использовать CASE в сочетании с агрегатной функцией, чтобы избежать повторного чтения таблицы. Давайте посмотрим, как оценит оптимизатор эти варианты.

Использование подзапроса

```
1.SELECT COUNT(*) total,
2.(SELECT COUNT(*) FROM Trip
```

```
3. WHERE  town_from='Rostov'  AND  town_to  <>  'Moscow')
   non_moscow
4. FROM Trip
5. WHERE town_from='Rostov';
```

Использование CASE с агрегатной функцией

```
1. SELECT COUNT(*) total_qty,
2. SUM(CASE  WHEN  town_to  <>'Moscow'  THEN  1  ELSE  0  END)
   non_moscow
3. FROM Trip
4. WHERE town_from='Rostov';
```

Результат, естественно, будет одинаков:

total	non_moscow
6	2

а вот стоимость второго запроса, как и ожидалось, оказалась вдвое ниже.

Вы можете сравнить реальное время выполнения, если сгенерируете достаточный объём данных.

Второй вариант можно записать более компактно, если использовать функцию **NULLIF** - сокращенный вариант частного случая использования CASE:

```
1. SELECT COUNT(*) total_qty,
2. COUNT(NULLIF(town_to, 'Moscow')) non_moscow
3. FROM Trip
4. WHERE town_from='Rostov';
```

Функция NULLIF возвращает NULL, если её аргументы равны, или первый аргумент в противном случае.

В решении используется тот факт, что агрегатные функции не учитывают NULL-значения, которые появляются в аргументе функции COUNT тогда, когда город прибытия равен 'Moscow'.

Начиная с версии 2012, в SQL Server появилась функция **IIF**, хорошо известная тем, кто использует VBA. Эта функция является альтернативой выражению CASE в MS Access и имеет следующий синтаксис:

```
1. IIF(<условие>, <выражение, если условие истинно>,
   <выражение, если условие не истинно>)
```

Функция возвращает результат вычисления выражения из второго аргумента, если условие есть TRUE; в противном случае возвращается результат вычисления выражения из третьего аргумента. Таким образом, функция

```
1. IIF(condition, expression_1, expression_2)
```

эквивалентна следующему выражению CASE:

```
1.   CASE WHEN condition THEN expression_1 ELSE expression_2
   END
```

С помощью функции IIF мы можем переписать решение первой задачи следующим образом:

```
1. SELECT DISTINCT product.model,
2.      IIF(price IS NULL, N'Нет в наличии',CAST(price AS
   CHAR(20))) price
3.     FROM Product LEFT JOIN
4.     PC ON Product.model = PC.model
5.     WHERE product.type = 'PC';
```

В том случае, если вариантов ветвления больше двух, можно использовать вложенные функции IIF. Например, для решения задачи 5.10.1 можно использовать такой запрос:

```
1. SELECT DISTINCT model, price,
2.       IIF(price=(SELECT MAX(price) FROM PC),N'Самый
   дорогой',
3.         IIF(price=(SELECT MIN(price) FROM PC), N'Самый
   дешевый',N'Средняя цена')) comment
4.     FROM PC
5.     ORDER BY price;
```

Если так и дальше пойдет, то скоро в T-SQL появится оператор SWITCH.

Операторы модификации данных

Язык манипуляции данными (DML — Data Manipulation Language) помимо

оператора **SELECT**, осуществляющего извлечение информации из базы данных, включает операторы, изменяющие состояние данных. Этими операторами являются:

оператор	функция
INSERT	Добавление записей (строк) в таблицу БД
UPDATE	Обновление данных в столбце таблицы БД
DELETE	Удаление записей из таблицы БД

Вы можете попрактиковаться в написании этих операторов на странице с **упражнениями по DML** на сайте SQL-EX.RU

Оператор INSERT

Оператор **INSERT** вставляет новые записи в таблицу. При этом значения столбцов могут представлять собой литеральные константы, либо являться результатом выполнения подзапроса. В первом случае для вставки каждой строки используется отдельный оператор **INSERT**; во втором случае будет вставлено столько строк, сколько возвращается подзапросом.

Синтаксис оператора следующий:

```
1.INSERT INTO <имя таблицы>[(<имя столбца>,...)]
2.{VALUES (<значение столбца>,…)}
3.| <выражение запроса>
4.| {DEFAULT VALUES}
```

Как видно из представленного синтаксиса, список столбцов не является обязательным (об этом говорят квадратные скобки в описании синтаксиса). В том случае, если он отсутствует, список вставляемых значений должен быть полный, то есть обеспечивать значения для всех столбцов таблицы. При этом порядок значений должен соответствовать порядку, заданному оператором **CREATE TABLE** для таблицы, в которую вставляются строки. Кроме того, эти значения должны относиться к тому же типу данных, что и столбцы, в которые они вносятся. В качестве примера рассмотрим вставку строки в таблицу Product, созданную следующим оператором **CREATE TABLE**:

```
1.CREATE TABLE product
2.(
3. maker char (1) NOT NULL,
4. model varchar (4) NOT NULL,
5. type varchar (7) NOT NULL
6.);
```

Пусть требуется добавить в эту таблицу модель ПК 1157 производителя B. Это можно сделать следующим оператором:

```
1.INSERT INTO Product
2.VALUES ('B', 1157, 'PC');
```

Если задать список столбцов, то можно изменить «естественный» порядок их следования:

```
1.INSERT INTO Product (type, model, maker)
2.VALUES ('PC', 1157, 'B');
```

Казалось бы, это совершенно излишняя возможность, которая делает конструкцию только более громоздкой. Однако она становится выигрышной, если столбцы имеют значения по умолчанию. Рассмотрим следующую структуру таблицы:

```
1.CREATE TABLE product_D
2.(
3. maker char (1) NULL,
4. model varchar (4) NULL,
5. type varchar (7) NOT NULL DEFAULT 'PC'
6. );
```

Отметим, что здесь значения всех столбцов имеют значения по умолчанию (первые два — NULL, а последний столбец — type — PC). Теперь мы могли бы написать:

```
1.INSERT INTO Product_D (model, maker)
2.VALUES (1157, 'B');
```

В этом случае отсутствующее значение при вставке строки будет заменено значением по умолчанию — PC. Заметим, что если для столбца в операторе **CREATE TABLE** не указано значение по умолчанию и не указано ограничение **NOT NULL**, запрещающее использование **NULL** в данном столбце таблицы, то подразумевается значение по умолчанию **NULL**.

Возникает вопрос: а можно ли не указывать список столбцов и, тем не менее, воспользоваться значениями по умолчанию? Ответ положительный. Для этого нужно вместо явного указания значения использовать зарезервированное слово **DEFAULT**:

```
1.INSERT INTO Product_D
2.VALUES ('B', 1158, DEFAULT);
```

Поскольку все столбцы имеют значения по умолчанию, для вставки строки со значениями по умолчанию можно было бы написать:

```
1.INSERT INTO Product_D
2.VALUES (DEFAULT, DEFAULT, DEFAULT);
```

Однако для этого случая предназначена специальная конструкция DEFAULT VALUES (см. синтаксис оператора), с помощью которой вышеприведенный оператор можно переписать в виде

```
1.INSERT INTO Product_D DEFAULT VALUES;
```

Заметим, что при вставке строки в таблицу проверяются все ограничения, наложенные на данную таблицу. Это могут быть ограничения первичного ключа или уникального индекса, проверочные ограничения типа **CHECK**, ограничения ссылочной целостности. В случае нарушения какого-либо ограничения вставка строки будет отклонена. Рассмотрим теперь случай использования подзапроса. Пусть нам требуется вставить в таблицу Product_D все строки из таблицы Product, относящиеся к моделям персональных компьютеров (type = ‘PC’). Поскольку необходимые нам значения уже имеются в некоторой таблице, то формирование вставляемых строк вручную, во-первых, является неэффективным, а, во-вторых, может допускать ошибки ввода. Использование подзапроса решает эти проблемы:

```
1.INSERT INTO Product_D
2.SELECT *
3.FROM Product
4.WHERE type = 'PC';
```

Использование в подзапросе символа «*» является в данном случае оправданным, так как порядок следования столбцов является одинаковым для обеих таблиц. Если бы это было не так, следовало бы применить список столбцов либо в операторе **INSERT**, либо в подзапросе, либо в обоих местах, который приводил бы в соответствие порядок следования столбцов:

```
1.INSERT INTO Product_D(maker, model, type)
2.SELECT *
3.FROM Product
4.WHERE type = 'PC';
```

или

```
1.INSERT INTO Product_D
2.SELECT maker, model, type
3.FROM Product
4.WHERE type = 'PC';
```

или

```
1.INSERT INTO Product_D(maker, model, type)
2.SELECT maker, model, type
3.FROM Product
4.WHERE type = 'PC';
```

Здесь, также как и ранее, можно указывать не все столбцы, если требуется использовать имеющиеся значения по умолчанию, например:

```
1.INSERT INTO Product_D (maker, model)
2.SELECT maker, model
3.FROM Product
4.WHERE type = 'PC';
```

В данном случае в столбец type таблицы Product_D будет подставлено значение по умолчанию PC для всех вставляемых строк.

Отметим, что при использовании подзапроса, содержащего предикат, будут вставлены только те строки, для которых значение предиката равно **TRUE** (не **UNKNOWN**!). Другими словами, если бы столбец type в таблице Product допускал бы **NULL**-значение, и это значение присутствовало бы в ряде строк, то эти строки не были бы вставлены в таблицу Product_D.

Преодолеть ограничение на вставку одной строки в операторе **INSERT** при использовании конструктора строки в предложении **VALUES** позволяет искусственный прием использования подзапроса, формирующего строку с предложением **UNION ALL**. Так если нам требуется вставить несколько строк при помощи одного оператора **INSERT**, можно написать:

```
1.INSERT INTO Product_D
2.SELECT 'B' AS maker, 1158 AS model, 'PC' AS type
3.UNION ALL
4.SELECT 'C', 2190, 'Laptop'
5.UNION ALL
6.SELECT 'D', 3219, 'Printer';
```

Использование **UNION ALL** предпочтительней **UNION** даже, если гарантировано отсутствие строк-дубликатов, так как в этом случае не будет выполняться проверка для исключения дубликатов.

Следует отметить, что вставка нескольких кортежей с помощью конструктора строк уже реализована в SQL Server 2008. С учетом этой возможности, последний запрос можно переписать в виде:

```
1.INSERT INTO Product_D VALUES
2.('B', 1158, 'PC'),
3.('C', 2190, 'Laptop'),
4.('D', 3219, 'Printer');
```

Заметим, что **MySQL** допускает еще одну нестандартную синтаксическую конструкцию, выполняющую вставку строки в таблицу в стиле оператора UPDATE:

```
1. INSERT [INTO] <имя таблицы>
2.   SET {<имя столбца>={<выражение> | DEFAULT}}, ...
```

Рассмотренный в начале параграфа пример с помощью этого оператора можно переписать так:

```
1. INSERT INTO Product
2. SET maker = 'B',
3.        model = 1157,
4.        type = 'PC';
```

Вставка строк в таблицу, содержащую автоинкрементируемое поле

Многие коммерческие продукты допускают использование автоинкрементируемых столбцов в таблицах, то есть столбцов, значение которых формируется автоматически при добавлении новых записей. Такие столбцы широко используются в качестве первичных ключей таблицы, так как они автоматически обеспечивают уникальность за счет того, что генерируемые значения не повторяются. Типичным примером столбца такого типа является последовательный счетчик, который при вставке строки генерирует значение на единицу большее предыдущего значения (значения, полученного при вставке предыдущей строки).

Ниже приводится пример создания таблицы Printer_Inc с автоинкрементируемым столбцом (code) в MS SQL Server.

```
1.CREATE TABLE Printer_Inc
2.(
3. code int IDENTITY(1,1) PRIMARY KEY ,
4. model varchar (4) NOT NULL ,
5. color char (1) NOT NULL ,
6. type varchar (6) NOT NULL ,
7. price float NOT NULL
8.);
```

Автоинкрементируемое поле определяется посредством конструкции IDENTITY (1, 1). При этом первый параметр свойства IDENTITY (1) определяет, с какого значения начнется отсчет, а второй, — какой шаг будет использоваться для приращения значения. Таким образом, в нашем примере первая вставленная запись будет иметь в столбце code значение 1, вторая — 2 и т. д.

Поскольку в поле code значение формируется автоматически, оператор

```
1.INSERT INTO Printer_Inc
2.VALUES (15, 3111, 'y', 'laser', 599);
```

приведет к ошибке, даже если в таблице нет строки со значением в поле code, равным 15. Поэтому для вставки строки в таблицу просто не будем указывать это поле точно так же, как и в случае использования значения по умолчанию, то есть

```
1.INSERT INTO Printer_Inc (model, color, type, price)
2. VALUES (3111, 'y', 'laser', 599);
```

В результате выполнения этого оператора в таблицу Printer_Inc будет вставлена информация о модели 3111 цветного лазерного принтера, стоимость

которого равна $599. В поле code окажется значение, которое только случайно может оказаться равным 15. В большинстве случаев этого оказывается достаточно, так как значение автоинкрементируемого поля, как правило, не несет никакой информации; главное, чтобы оно было уникальным.

Однако бывают случаи, когда требуется подставить вполне конкретное значение в автоинкрементируемое поле. Например, нужно перенести уже имеющиеся данные во вновь создаваемую структуру; при этом эти данные участвуют в связи «один-ко-многим» со стороны «один». Таким образом, мы не можем допустить тут произвола. С другой стороны, не хочется отказываться от автоинкрементируемого поля, так как оно упростит обработку данных при последующей эксплуатации базы данных.

Поскольку стандарт языка SQL не предполагает наличия автоинкрементируемых полей, то не существует и единого подхода. Здесь мы покажем, как это реализуется в MS SQL Server. Оператор

```
1.SET IDENTITY_INSERT < имя таблицы > { ON | OFF };
```

отключает (значение ON) или включает (OFF) использование автоинкремента. Поэтому чтобы вставить строку со значением 15 в поле code, нужно написать:

```
1.SET IDENTITY_INSERT Printer_Inc ON;
2.INSERT INTO Printer_Inc(code, model, color, type, price)
3.VALUES (15, 3111, 'y', 'laser', 599);
```

Обратите внимание, что список столбцов в этом случае является обязательным, то есть мы не можем написать так:

```
1.SET IDENTITY_INSERT Printer_Inc ON;
2.INSERT INTO Printer_Inc
3.VALUES (15, 3111, 'y', 'laser', 599);
```

ни, тем более, так:

```
1.SET IDENTITY_INSERT Printer_Inc ON;
2.INSERT INTO Printer_Inc(model, color, type, price)
3.VALUES (3111, 'y', 'laser', 599);
```

В последнем случае в пропущенный столбец code значение не может быть подставлено автоматически, так как автоинкрементирование отключено.

Важно отметить, что если значение 15 окажется максимальным в столбце code, то далее нумерация продолжится со значения 16. Естественно, если включить автоинкрементирование:

```
1.SET IDENTITY_INSERT Printer_Inc OFF;
```

Наконец, рассмотрим пример вставки данных из таблицы Product в таблицу Product_Inc, сохранив значения в поле code:

```
1.SET IDENTITY_INSERT Printer_Inc ON;
2.INSERT INTO Printer_Inc(code, model,color,type,price)
3. SELECT * FROM Printer;
```

По поводу автоинкрементируемых столбцов следует добавить следующее. Пусть последнее значение в поле code было равно 16, после чего строка с этим значением была удалена. Какое значение будет в этом столбце после вставки новой строки? Правильно, 17, так как последнее значение счетчика сохраняется, несмотря на удаление строки, его содержащей. Поэтому нумерация значений в результате удаления и добавления строк не будет последовательной. Это является еще одной причиной для вставки строки с заданным (пропущенным) значением в автоинкрементируемом столбце.

Рассмотрим теперь таблицу с единственным автоинкрементируемым столбцом (SQL Server):

```
1.CREATE TABLE test_Identity(
2.    id int IDENTITY(1,1) PRIMARY KEY
3.);
```

Как вставить в нее строки? Попытка не указывать значение

```
1. INSERT INTO test_Identity VALUES();
```

или использовать значение по умолчанию

```
1. INSERT INTO test_Identity VALUES(DEFAULT);
```

к успеху не приводит - ошибка.

Понятно, что мы можем вставить конкретное значение, если отключим счетчик:

```
1. SET IDENTITY_INSERT test_Identity ON;
2. INSERT INTO test_Identity(id) VALUES(5);
3. SELECT * FROM test_Identity;
```

но тогда в нем нет для нас никакого смысла.

Уточним вопрос: как вставить в таблицу именно последовательные значения счетчика? Оказывается, что ответ лежит на поверхности, а именно, в стандартном синтаксисе:

```
1. SET IDENTITY_INSERT test_Identity OFF;
2. INSERT INTO test_Identity DEFAULT VALUES;
```

Вряд ли вы будете использовать DEFAULT VALUES в других случаях, т.к. при наличии первичного ключа воспользоваться значениями по умолчанию для всех столбцов таблицы можно будет только один раз. Тут же мы можем повторить этот оператор столько раз, сколько последовательных значений счетчика нам потребуется добавить в таблицу.

Обратимся теперь к другим СУБД, которые имеют в своем арсенале автоинкрементируемые столбцы.

MySQL

MySQL не поддерживает предложения DEFAULT VALUES. Вставить строку со значениями по умолчанию в таблицу можно другим стандартным способом, используя ключевое слово **DEFAULT** для каждого столбца таблицы - VALUES(DEFAULT, DEFAULT, ...).

А как здесь вставить очередное значение счетчика в единственный автоинкрементируемый столбец?

```
1.CREATE TABLE test_Identity (
2. id int(11) NOT NULL AUTO_INCREMENT,
3.  PRIMARY KEY  (id)
4.);
```

Очень просто. Оказывается будут работать те интуитивные приемы, которые мы безуспешно пытались применить в случае SQL Server, а именно, так

```
1.INSERT INTO test_Identity VALUES();
```

или так

```
1.INSERT INTO test_Identity VALUES(DEFAULT);
```

После выполнения обоих этих операторов получим:

id
1
2

Заметим, что после вставки конкретного значения в автоинкрементируемый столбец (в MySQL это можно сделать обычным оператором вставки),

которое будет превышать максимальное имеющееся, приращение счетчика продолжится уже с него. Например:

```
1.INSERT INTO test_Identity VALUES(8);
2.INSERT INTO test_Identity VALUES(DEFAULT);
3.SELECT * FROM test_Identity;
```

id
1
2
8
9

PostgreSQL

```
1. CREATE TABLE identity_table(id serial PRIMARY KEY);
```

Для вставки очередных значений счетчика мы можем воспользоваться следующими рассмотренными выше приемами:

```
1.INSERT INTO identity_table DEFAULT VALUES;
2.INSERT INTO identity_table VALUES(DEFAULT);
3.INSERT INTO identity_table(id) VALUES(DEFAULT);
4. SELECT * FROM identity_table;
```

id
1
2
3

Однако, если вставить конкретное значение (превышающее максимальное значение, достигнутое счетчиком)

```
1.INSERT INTO identity_table(id) VALUES(5);
```

и продолжить заполнять значения счетчика,

```
1.INSERT INTO identity_table VALUES(DEFAULT);
```

то нумерация продолжается не с максимального значения, а с последнего достигнутого при генерации:

id
1
2
3
5
4

При этом, когда счетчик достигает 5 при генерации очередного значения, получаем ошибку, связанную с нарушением ограничения первичного ключа:

ERROR: duplicate key value violates unique constraint "identity_table_pkey"

DETAIL: Key (id)=(5) already exists.

Если же таблица не имеет ключа на автоинкрементируемом столбце, то мы получим дубликаты, после чего нумерация продолжится дальше. Вот скрипт, который поясняет сказанное:

```
1.CREATE TABLE identity_table_wo(id serial);
2.INSERT INTO identity_table_wo(id) VALUES(DEFAULT);
3.INSERT INTO identity_table_wo(id) VALUES(2);
4.INSERT INTO identity_table_wo(id)
  VALUES(DEFAULT),(DEFAULT);
```

```
5.SELECT * FROM identity_table_wo;
```

id
1
2
2
3

Как сбросить значение счетчика в заданное значение (MySQL)?

Воспользуемся таблицей, рассмотренной на предыдущей странице, и вставим в неё 3 строки.

```
1.CREATE TABLE test_Identity (
2.      id int(11) NOT NULL AUTO_INCREMENT,
3.       PRIMARY KEY   (id)
4.     );
5.     INSERT INTO test_Identity VALUES(),(),();
6.     SELECT * FROM test_Identity;
```

id
1
2
3

Если мы удалим последнюю строку, нумерация продолжится не с 3, а с 4. Т.е. последнее значение счётчика сохраняется и используется при последующем добавлении строк:

```
1.DELETE FROM test_Identity WHERE id=3;
2.INSERT INTO test_Identity VALUES();
3.    SELECT * FROM test_Identity;
```

id
1
2
4

Возникает вопрос: "А можно ли сделать так, чтобы нумерация продолжилась с последнего имеющегося значения?" Оставляя в стороне вопрос о том, зачем это нужно, ответим - можно. Но устанавливать это значение счётчика нужно вручную. Итак,

```
1.DELETE FROM test_Identity WHERE id=4;
2.ALTER TABLE test_Identity AUTO_INCREMENT = 3;
3.INSERT INTO test_Identity VALUES(),(),();
4.    SELECT * FROM test_Identity;
```

id
1
2
3
4
5

Конструктор значений таблицы

Синтаксис конструктора значений таблицы:

```
1.VALUES
2.(<элемент конструктора>, <элемент конструктора>, ...),
3.(<элемент конструктора>, <элемент конструктора>, ...),
4....
5.(<элемент конструктора>, <элемент конструктора>, ...)
```

При этом элементом конструктора может быть:

- выражение, вычисляющее значение, совместимое с типом данных соответствующего столбца таблицы;

- DEFAULT - для подстановки значения по умолчанию для соответствующего столбца таблицы;

- NULL;

- подзапрос, возвращающий одно значение, совместимое с типом данных соответствующего столбца таблицы.

Конструктор значений таблицы может использоваться для вставки набора строк в существующую таблицу с помощью одного оператора INSERT.

Создадим следующую таблицу для выполнения примеров:

```
1.CREATE TABLE Items (
2.item_no int PRIMARY KEY,
3.maker char(10),
4.type char(10) DEFAULT 'PC',
5.value int
6. );
```

Вставим в таблицу 4 строки, используя конструктор.

```
1.INSERT INTO Items VALUES
2.(1, 'A', 'Laptop', 12),
```

```
3.(2, 'B', DEFAULT, NULL),
4.(3, 'C', 'Printer', (SELECT CAST(model AS int) FROM
  Printer WHERE code=1)),
5.(4, 'C', 'Printer', (SELECT CAST(model AS int) FROM
  Printer WHERE code=77));

1.SELECT * FROM Items;
```

item_no	maker	type	value
1	A	Laptop	12
2	B	PC	NULL
3	C	Printer	3001
4	C	Printer	NULL

Последнее значение в двух последних строках было получено с помощью подзапроса, который возвращает либо одно значение (поскольку выполняется отбор по ключу) с номером модели из таблицы Printer, либо ни одного. Последнее имеет место для четвертой строки, поскольку коду 77 не отвечает никакая строка таблицы Printer. В этом случае будет записано NULL-значение.

Конструктор значений таблицы может использоваться также в предложении FROM. В параграфе, посвященном генерации числовой последовательности, последний пример, который находит 100 последовательных незанятых номеров моделей, с учетом этой возможности можно переписать более компактно:

```
1.SELECT (SELECT MAX(model)
2.    FROM Product
3.    ) + 5*5*(a-1) + 5*(b-1) + c AS num
4.    FROM
5.    (VALUES(1),(2),(3),(4),(5)) x(a) CROSS JOIN
6.    (VALUES(1),(2),(3),(4),(5)) y(b)  CROSS JOIN
7.    (VALUES(1),(2),(3),(4),(5)) z(c)
8.    WHERE 5*5*(a-1) + 5*(b-1) + c <= 100
9.    ORDER BY 1;
```

Еще один пример использования конструктора значений таблицы для трансформации строки в столбец можно увидеть в главе, посвященной оператору CROSS APPLY.

Оператор UPDATE

Оператор **UPDATE** изменяет имеющиеся данные в таблице. Команда имеет следующий синтаксис:

```
1. UPDATE <имя таблицы>
2. SET {<имя столбца> = {<выражение для вычисления значения
   столбца>
3. | NULL
4. | DEFAULT},...}
5. [ {WHERE <предикат>}]
```

С помощью одного оператора могут быть заданы значения для любого количества столбцов. Однако в одном и том же операторе **UPDATE** можно вносить изменения в каждый столбец указанной таблицы только один раз. При отсутствии предложения **WHERE** будут обновлены все строки таблицы.

Если столбец допускает **NULL**-значение, то его можно указать в явном виде. Кроме того, можно заменить имеющееся значение на значение по умолчанию (DEFAULT) для данного столбца.

Ссылка на «выражение для вычисления значения столбца» может относиться к текущим значениям в изменяемой таблице. Например, мы можем уменьшить все цены портативных компьютеров на 10 процентов с помощью следующего оператора:

```
1. UPDATE Laptop
2. SET price = price*0.9;
```

Разрешается также значения одних столбцов присваивать другим столбцам. Пусть, например, требуется заменить жесткие диски менее 10 Гбайт в

портативных компьютерах. При этом емкость новых дисков должна составлять половину объема RAM, имеющейся в данных устройствах. Эту задачу можно решить следующим образом:

```
1.UPDATE Laptop
2.SET hd = ram/2 WHERE hd < 10;
```

Естественно, типы данных столбцов hd и ram должны быть совместимы. Для приведения типов может использоваться выражение **CAST** (пункт 5.9).

Если требуется изменять данные в зависимости от содержимого некоторого столбца, можно воспользоваться выражением **CASE** (пункт 5.9) Если, скажем, нужно поставить жесткие диски объемом 20 Гбайт на портативные компьютеры с памятью менее 128 Мбайт и 40 гигабайтные — на остальные портативные компьютеры, то можно написать такой запрос:

```
1.UPDATE Laptop
2.SET hd = CASE
3.WHEN ram < 128
4.THEN 20
5.ELSE 40
6. END;
```

Для вычисления значений столбцов допускается также использование подзапросов. Например, требуется укомплектовать все портативные компьютеры самыми быстрыми процессорами из имеющихся в наличии. Тогда можно написать:

```
1.UPDATE Laptop
2.SET speed = (SELECT MAX(speed)
3.FROM Laptop
4.);
```

Необходимо сказать несколько слов об автоинкрементируемых столбцах. Если столбец code в таблице Laptop определен как IDENTITY(1,1), то следующий оператор

```
1.UPDATE Laptop
2.SET code = 5
3.WHERE code = 4;
```

не будет выполнен, так как автоикрементируемое поле не допускает обновления, и мы получим соответствующее сообщение об ошибке. Чтобы выполнить все же эту задачу, можно поступить следующим образом. Сначала вставить нужную строку, используя **SET IDENTITY_INSERT**, после чего удалить старую строку:

```
1.SET IDENTITY_INSERT Laptop_ID ON;
2.INSERT INTO Laptop_ID(code, model, speed, ram, hd, price,
  screen)
3.SELECT 5, model, speed, ram, hd, price, screen
4.FROM Laptop_ID WHERE code = 4;
5.DELETE FROM Laptop_ID
6.WHERE code = 4;
```

Разумеется, другой строки со значением code = 5 в таблице быть не должно.

В Transact-SQL оператор **UPDATE** расширяет стандарт за счет применения необязательного предложения **FROM**. В этом предложении специфицируется таблица, обеспечивающая критерий для операции обновления. Дополнительную гибкость здесь дают операции соединения таблиц.

Пример 6.2.1.

Пусть требуется указать «No PC» (нет ПК) в столбце type для тех моделей ПК из таблицы Product, для которых нет соответствующих строк в таблице PC. Решение посредством соединения таблиц можно записать так

```
1.UPDATE Product
2.SET type = 'No PC'
3.FROM Product pr LEFT JOIN
4. PC ON pr.model=PC.model
5.WHERE type = 'pc' AND
6. PC.model IS NULL;
```

Здесь применяется внешнее соединение, в результате чего столбец PC.model для моделей ПК, отсутствующих в таблице PC, будет содержать **NULL**-значение, что и используется для идентификации подлежащих обновлению строк. Естественно, эта задача имеет решение и в «стандартном» исполнении:

```
1.UPDATE Product
2.SET type = 'No PC'
3.WHERE type = 'pc' AND
4. model NOT IN (SELECT model
5.  FROM PC
6.  );
```

Оператор DELETE

Оператор **DELETE** удаляет строки из временных или постоянных базовых таблиц, представлений или курсоров, причем в двух последних случаях действие оператора распространяется на те базовые таблицы, из которых извлекались данные в эти представления или курсоры. Оператор удаления имеет простой синтаксис:

```
1.DELETE FROM <имя таблицы >
2.[WHERE <предикат>];
```

Если предложение **WHERE** отсутствует, удаляются все строки из таблицы или представления (представление должно быть обновляемым). Более быстро эту операцию (удаление всех строк из таблицы) можно в Transact-SQL также выполнить с помощью команды

```
1.TRUNCATE TABLE <имя таблицы>
```

Однако есть ряд особенностей в реализации команды **TRUNCATE TABLE**, которые следует иметь в виду:

- не журнализируется удаление отдельных строк таблицы; в журнал записывается только освобождение страниц, которые были заняты данными таблицы;
- не отрабатывают триггеры, в частности, триггер на удаление;
- команда неприменима, если на данную таблицу имеется ссылка по внешнему ключу, и даже если внешний ключ имеет опцию каскадного удаления.
- значение счетчика (**IDENTITY**) сбрасывается в начальное значение.

Пример 6.3.1

Требуется удалить из таблицы Laptop все портативные компьютеры с размером экрана менее 12 дюймов.

```
1.DELETE FROM Laptop
2.WHERE screen < 12;
```

Все блокноты можно удалить с помощью оператора

```
1.DELETE FROM Laptop;
```

или

```
1.TRUNCATE TABLE Laptop;
```

Transact-SQL расширяет синтаксис оператора **DELETE**, вводя дополнительное предложение **FROM**:

```
1.FROM <источник табличного типа>
```

При помощи источника табличного типа можно конкретизировать данные, удаляемые из таблицы в первом предложении **FROM**.

При помощи этого предложения можно выполнять соединения таблиц, что логически заменяет использование подзапросов в предложении **WHERE** для идентификации удаляемых строк. Поясним сказанное на примере.

Пример 6.3.2

Пусть требуется удалить те модели ПК из таблицы Product, для которых нет соответствующих строк в таблице PC.

Используя стандартный синтаксис, эту задачу можно решить следующим запросом:

```
1. DELETE FROM Product
2. WHERE type = 'pc' AND
3.  model NOT IN (SELECT model
4.  FROM PC
5.  );
```

Заметим, что предикат type = 'pc' необходим здесь, чтобы не были удалены также модели принтеров и портативных компьютеров.

Эту же задачу можно решить с помощью дополнительного предложения **FROM** следующим образом:

```
1. DELETE FROM Product
2. FROM Product pr LEFT JOIN
3.  PC ON pr.model = PC.model
4. WHERE type = 'pc' AND
5.  PC.model IS NULL;
```

Здесь применяется внешнее соединение, в результате чего столбец PC.model для моделей ПК, отсутствующих в таблице PC, будет содержать **NULL**-значение, что и используется для идентификации подлежащих удалению строк.

Оператор TRUNCATE TABLE

Как отмечалось выше, при выполнении этой команды значение счетчика (IDENTITY) сбрасывается в начальное значение. Давайте проверим это утверждение в MS SQL Server. Для начала создадим таблицу с автоинкрементируемым столбцом, и добавим в нее три строки.

```
1. CREATE TABLE Truncate_test (id INT IDENTITY(5,5) PRIMARY
   KEY, val INT);
2. GO
3. INSERT INTO Truncate_test(val)
4. VALUES (1),(2),(3);
5. SELECT * FROM Truncate_test;
6. GO
```

Начальным значением счетчика является 5, приращение счетчика выполняется также с шагом 5.
В результате получим:

id	val
5	1
10	2
15	3

Теперь удалим строки с помощью оператора **DELETE**, после чего снова вставим те же строки в таблицу.

```
1.DELETE FROM Truncate_test;
2.GO
3.INSERT INTO Truncate_test(val)
4.VALUES (1), (2), (3);
5.SELECT * FROM Truncate_test;
6.GO
```

id	val
20	1
25	2
30	3

Как видно из результата, состояние счетчика не было сброшено, и приращение продолжилось с последнего значения (15), в отличие от использования оператора TRUNCATE TABLE:

```
1.TRUNCATE TABLE Truncate_test;
2.GO
3.INSERT INTO Truncate_test(val)
4.VALUES (1),(2),(3);
5.SELECT * FROM Truncate_test;
6.GO
```

id	val
5	1
10	2
15	3

В то же время Стандарт предполагает несколько иное поведение. Стандартный синтаксис имеет вид

```
1. TRUNCATE TABLE < имя таблицы > [{CONTINUE IDENTITY} |
   {RESTART IDENTITY}]
```

т.е. значение счетчика может быть сброшено (опция **RESTART IDENTITY**) или продолжено (опция **CONTINUE IDENTITY**). И, кстати, значением по умолчанию является как раз CONTINUE IDENTITY, что эквивалентно поведению при использованию оператора DELETE (без предложения WHERE).

Оператор TRUNCATE TABLE неприменим, если на таблицу имеется ссылка по внешнему ключу. Это стандартное поведение имеет место в SQL Server. Если создать, например, такую ссылающуюся таблицу, которая даже не будет содержать данных

```
1. CREATE TABLE Trun_Ref(id INT REFERENCES Truncate_test);
```

оператор TRUNCATE TABLE приведет к следующей ошибке:

Cannot truncate table 'Truncate_test' because it is being referenced by a FOREIGN KEY constraint.

(Невозможно усечь таблицу 'Truncate_test', поскольку на нее ссылается ограничение FOREIGN KEY) .

Теперь проверим, насколько близки к стандарту другие реализации.

PostgreSQL

1. Поддерживаются опции CONTINUE IDENTITY и RESTART IDENTITY, при этом опция CONTINUE IDENTITY принимается по умолчанию.
2. Можно удалить одним оператором строки из нескольких таблиц, перечислив их через запятую.
3. Допускаются каскадные операции, т.е. усечение связанных таблиц:

```
1.TRUNCATE TABLE Truncate_test RESTART IDENTITY CASCADE;
```

Причем происходит именно усечение, а не удаление связанных строк. Т.е. если вставить в ссылающуюся таблицу среди прочих строку с NULL-значением во внешнем ключе

```
1.INSERT INTO Trun_Ref VALUES(1), (2), (NULL);
```

то она также будет удалена наряду с остальными.

Oracle

1. В Oracle нет функции для автоинкремента, которую можно указать в определении столбца. Однако поведение автоинкремента можно сымитировать с помощью последовательности (**SEQUENCE**). Например, подобную ранее рассмотренной таблицу Truncate_test в Oracle можно создать следующим образом:

```
1.CREATE SEQUENCE u_seq
2. START WITH     5
3. INCREMENT BY   5;
4./
5.CREATE TABLE Truncate_test (id INT PRIMARY KEY, val
  int);
6./
7.INSERT INTO Truncate_test(id, val)
```

```
8.VALUES (u_seq.NEXTVAL,1);
9.INSERT INTO Truncate_test(id, val)
10. VALUES (u_seq.NEXTVAL,2);
11. INSERT INTO Truncate_test(id, val)
12. VALUES (u_seq.NEXTVAL,3);
```

2. При выполнении оператора TRUNCATE TABLE Truncate_test состояние счетчика (последовательности) не сбрасывается, и нумерация будет продолжена.
3. Каскадные операции не допускаются, т.е. оператор неприменим, если на таблицу есть ссылка по внешнему ключу.

MySQL

1. Не поддерживаются опции CONTINUE IDENTITY и RESTART IDENTITY, при этом состояние счетчика (**AUTO_INCREMENT**) сбрасывается.
2. Допускаются каскадные операции по аналогии с оператором DELETE, т.е. если внешний ключ имеет опцию **ON DELETE CASCADE**, то удаляются только связанные записи. Это значит, что строки с NULL-значением внешнего ключа остаются после выполнения оператора

```
1.TRUNCATE TABLE Truncate_test;
```

Готовимся ко второму этапу тестирования

Задачи, рассмотренные нами ранее в главах 3–4, составляют первый, или обучающий, этап тестирования на сайте. Эти относительно простые задачи, для решения которых, как правило, не требуется знания особенностей реализации. Это означает, что решение практически любой из них

можно получить, используя конструкции языка SQL, оформленные стандартом **SQL-92**.

На втором этапе предлагаются более сложные задачи, для решения которых уже не обойтись без знаний конкретной реализации, например, функций работы со строками и значениями типа даты/времени.

Задачи этого этапа не рассматриваются в книге, так как по результатам их решения на сайте можно получить сертификат. Тем не менее, мы считаем необходимым познакомить читателя со специфическими особенностями **SQL Server**, а также некоторыми алгоритмическими приемами, которые могут применяться для решения как задач на сайте, так и задач, часто возникающих на практике.

Функции Transact-SQL для работы со строками и данными типа даты/времени

Стандарт SQL-92 специфицирует незначительное число функций для работы со строковыми значениями и значениями даты и времени. Что касается последних, то они ограничиваются лишь функциями, возвращающими системную дату/время. Например,
функция CURRENT_TIMESTAMP возвращает сразу и дату, и время. Плюс имеются функции возвращающие что-либо одно.

Естественно, в силу такой ограниченности, реализации языка расширяют стандарт за счет добавления функций, облегчающий работу пользователей с данными этого типа. Это обусловлено еще и тем, что на нижнем уровне соответствия стандарту (Entry Level) не требуется поддержка стандартизованных функций этих типов.

Функция DATEADD

Функция **DATEADD** (datepart, number, date) возвращает значение типа datetime, которое получается добавлением к дате date количества интервалов типа datepart, равного number (целое число).

Например, мы можем к заданной дате добавить любое число лет, дней, часов, минут и т. д.

Допустимые значения аргумента datepart приведены ниже в таблице и взяты из электронной документации к SQL Server — Books On Line (BOL).

Datepart	Допустимые сокращения
Year — год	yy, yyyy
Quarter — квартал	qq, q
Month — месяц	mm, m
Dayofyear — день года	dy, y
Day — день	dd, d
Week — неделя	wk, ww
Hour — час	hh
Minute — минута	mi, n
Second — секунда	ss, s
Millisecond - миллисекунда	ms

Пусть сегодня 28.10.2005, и мы хотим узнать, какой день будет через неделю. Мы можем написать:

```
1. SELECT DATEADD(day, 7, current_timestamp);
```

а можем и так:

```
1. SELECT DATEADD(ww, 1, current_timestamp);
```

В результате получим одно и то же значение; что-то типа 2005-11-04 00:11:28.683.

Однако мы не можем в этом случае написать:

```
1. SELECT DATEADD(mm, 1/4, current_timestamp);
```

и не потому, что четверть месяца не равна в точности неделе, а потому, что дробная часть значения аргумента datepart отбрасывается, и мы получим 0 вместо одной четвертой и, как следствие, текущий день.

Кроме того, мы можем использовать вместо **CURRENT_TIMESTAMP** функцию T-SQL **GETDATE**() с тем же самым эффектом. Наличие двух идентичных функций поддерживается, видимо, в ожидании последующего развития стандарта.

Пример 7.1.1

Определить, какой будет день через неделю после последнего полета.

Примечание:

В примерах данной главы используется база данных «Аэрофлот». Описание этой схемы и всех остальных схем, используемых в настоящее время на сайте для решения задач, можно найти в Примечании 1.

```
1. SELECT DATEADD(day, 7, (SELECT MAX(date) max_date
2.  FROM pass_in_trip
```

```
3.  )
4.  );
```

Применение подзапроса в качестве аргумента допустимо, так как этот подзапрос возвращает единственное значение типа datetime.

На примере задачи 7.1.1 рассмотрим добавление интервала к дате для других СУБД.

MySQL

MySQL имеет похожую функцию с непохожими аргументами. Вот синтаксис этой функции:

```
1.DATE_ADD(date, INTERVAL value addunit)
```

Здесь

date - дата, к которой прибавляется интервал;

value - величина интервала;

addunit - тип интервала.

Допустимы следующие типы интервалов, имена которых говорят сами за себя:

MICROSECOND
SECOND
MINUTE
HOUR
DAY
WEEK
MONTH
QUARTER

YEAR
SECOND_MICROSECOND
MINUTE_MICROSECOND
MINUTE_SECOND
HOUR_MICROSECOND
HOUR_SECOND
HOUR_MINUTE
DAY_MICROSECOND
DAY_SECOND
DAY_MINUTE
DAY_HOUR
YEAR_MONTH

Решение нашей задачи для MySQL примет вид:

```
1. SELECT DATE_ADD((SELECT MAX(date) FROM pass_in_trip),
2.                 interval 7 day) next_wd;
```

next_wd
2005-12-06 00:00:00

Чтобы добавить интервал, представляющий собой несколько компонентов времени, используется подстрока из стандартного представления

даты/времени. Так, например, чтобы добавить к '2018-01-27T13:00:00' один день и 3 часа, можно написать:

```
1. SELECT    DATE_ADD('2018-01-27T13:00:00'    ,    interval
   '1T3'  DAY_HOUR);
```

2018-01-28 16:00:00

Добавление 1 дня и 15 секунд будет выглядеть так:

```
1. SELECT     DATE_ADD('2018-01-27T13:00:00',     interval
   '01T00:00:15'  DAY_SECOND);
```

2018-01-28 13:00:15

PostgreSQL и Oracle

Эти СУБД не используют функцию. Для добавления интервала применяется обычный оператор сложения "+":

```
1. SELECT MAX("date") + interval '7' day next_wd
2. FROM pass_in_trip;
```

Обратите внимание, что величина интервала должна иметь символьный тип данных.

Добавить 1 день и 3 часа

PostgreSQL

У PostgreSQL нет составных интервалов, поэтому можно либо выразить величину интервала в терминах меньшего интервала

```
1. SELECT    timestamp'2018-01-27T13:00:00'    +    interval
   '27'  hour;
```

либо добавить два интервала

```
1. SELECT timestamp'2018-01-27T13:00:00' + interval '3' hour
   + interval '1' day;
```

Аналогично можно поступить для добавления одного дня и 15 секунд, например:

```
1. SELECT  timestamp'2018-01-27T13:00:00'  +  interval  '15'
   second + interval '1' day;
```

Oracle

Oracle позволяет использовать составные интервалы, например, 1 день и 3 часа:

```
1. SELECT  timestamp'2018-01-27  13:00:00'  +  interval  '01
   03'  DAY TO HOUR FROM dual;
```

и 1 день 15 секунд

```
1. SELECT  timestamp'2018-01-27  13:00:00'  +  interval  '01
   00:00:15'  DAY TO SECOND FROM dual;
```

Разумеется, можно также прибавить два простых интервала, как и в случае PostgreSQL.

Функция DATEDIFF

Синтаксис:

```
1.DATEDIFF(datepart, startdate, enddate)
```

Функция возвращает интервал времени, прошедшего между двумя временными отметками — startdate (начальная отметка) и enddate (конечная отметка). Этот интервал может быть измерен в разных единицах. Возможные варианты определяются аргументом datepart и перечислены выше применительно к функции DATEADD.

Пример 7.1.2

Определить количество дней, прошедших между первым и последним совершенными рейсами.

```
1.SELECT DATEDIFF(dd, (SELECT MIN(date)
2. FROM pass_in_trip
3. ),
4. (SELECT MAX(date)
5. FROM pass_in_trip
6. )
7. );
```

Пример 7.1.3

Определить продолжительность рейса 1123 в минутах.

Здесь следует принять во внимание, что время вылета (time_out) и время прилета (time_in) хранится в полях типа datetime таблицы Trip. Как уже отмечалось, SQL Server вплоть до версии 2005 не имел отдельных темпоральных типов данных для даты и времени. Поэтому при вставке в поле datetime только времени (например, UPDATE trip SET time_out = ’17:24:00’ WHERE trip_no = 1123), время будет дополнено значением даты по умолчанию (1900-01-01), являющейся начальной точкой отсчета времени.

Напрашивающееся решение

```
1.SELECT DATEDIFF(mi, time_out, time_in) dur
2.FROM trip
```

```
3.WHERE trip_no = 1123;
```

(которое дает 760 минут) будет неверным по двум причинам.

Во-первых, для рейсов, которые вылетают в один день, а прилетают на следующий, вычисленное таким способом значение будет неправильным;

Во-вторых, ненадежно делать какие либо предположения относительно дня, который присутствует только в силу необходимости соответствовать типу datetime.

Но как определить, что самолет приземлился на следующий день? Тут помогает описание предметной области, где говорится, что полет не может продолжаться более суток. Итак, если время прилета не больше, чем время вылета, то этот факт имеет место.

Теперь второй вопрос: как посчитать только время, с каким бы днем оно ни стояло?

Здесь может помочь функция T-SQL **DATEPART**, речь о которой пойдет в **следующем пункте**.

Следует иметь в виду одну особенность использования функции **DATEDIFF**. Начнем с примеров. Сначала посчитаем число недель с воскресенья 23-10-2005 до субботы 29-10-2005. Итак,

```
1.SELECT DATEDIFF(wk, '20051023', '20051029');
```

Здравый смысл подсказывает, что это полная неделя, однако, вышеприведенный запрос дает 0. Теперь возьмем интервал с субботы 29-10-2005 до воскресенья 30-10-2005:

```
1.SELECT DATEDIFF(wk, '20051029', '20051030');
```

В результате получим 1, то есть одну неделю. Пора дать объяснения. Дело в том, что функция DATEDIFF фактически считает неделей не число дней, а число переходов с субботы на воскресенье. Если это иметь в виду, то тогда становится понятным еще более удивительный пример:

```
1. SELECT DATEDIFF(wk, '20051029 23:59:00', '20051030 00:01:00');
```

который тоже дает единицу!

Странно? Возможно, но, как говорится, кто предупрежден, тот вооружен. То же самое имеет место и для других интервалов. Например, количество дней дает нам не число часов, деленное нацело на 24 (количество часов в сутках), а число переходов через полночь. Хотите подтверждения? Пожалуйста

```
1. SELECT DATEDIFF(dd, '20051029 23:59:00', '20051030 00:01:00');
```

В результате получаем один день, в то же время

```
1. SELECT DATEDIFF(dd, '20051029 00:00:00', '20051029 23:59:59');
```

дает нам 0.

Если хотите, можете продолжить эксперименты с другими временными интервалами. Вы можете воспользоваться для этого предоставляемой Консолью.

Задача. Посчитать число минут в интервале между двумя датами – '2011-10-07 23:43:00' и '2011-10-08 01:23:00'

SQL Server

Встроенная функция **DATEDIFF** решает проблему:

```
1. SELECT DATEDIFF (minute, '2011-10-07T23:43:00', '2011-10-08T01:23:00');
```

Результат – 100 минут.

Примечание:

В запросе используется стандартное представление даты (ISO) в виде текстовой строки 'yyyy-mm-ddThh:mm:ss', которое не зависит ни от локальных настроек сервера, ни и от самого сервера.

MySQL

Функция DATEDIFF есть и в MySQL, однако она имеет совсем другой смысл. DATEDIFF вычисляет число дней между двумя датами, являющихся аргументами этой функции. При этом если дата представлена в формате дата-время, используется только составляющая даты. Поэтому все три нижепредставленных запроса дадут один и тот же результат -1. Положительный результат будет получен, если первый аргумент больше второго.

```
1. SELECT DATEDIFF('2011-10-07T23:43:00', '2011-10-08T01:23:00');
2. SELECT DATEDIFF('2011-10-07', '2011-10-08');
3. SELECT DATEDIFF('2011-10-07T23:43:00', '2011-10-08');
```

Один день мы получим даже в случае, если интервал между датами составляет все одну секунду:

```
1. SELECT DATEDIFF('2011-10-07T23:59:59', '2011-10-08T00:00:00');
```

Решение же нашей задачи можно получить при помощи другой встроенной функции – TIMESTAMPDIFF, которая аналогичная функции DATEDIFF в SQL Server:

```
1. SELECT TIMESTAMPDIFF(minute, '2011-10-07T23:43:00', '2011-10-08T01:23:00');
```

PostgreSQL

В PostgreSQL нет функции, подобной DATEDIFF (SQL Server) или TIMESTAMPDIFF (MySQL). Поэтому для решения задачи можно применить следующую последовательность действий:

1. представить разность между двумя датами интервалом;
2. посчитать число секунд в интервале;
3. поделить полученное на шаге 2 значение на 60, чтобы выразить результат в минутах.

Для получения интервала можно взять разность двух значений темпорального типа, При этом требуется явное преобразование типа:

```
1. SELECT timestamp '2011-10-08T01:23:00' - timestamp '2011-10-07T23:43:00';
```

или в стандартном исполнении

```
1. SELECT CAST('2011-10-08T01:23:00' AS timestamp) - CAST('2011-10-07T23:43:00' AS timestamp);
```

Результат - "01:40:00", который есть не что иное, как один час и сорок минут.

Можно также воспользоваться встроенной функцией **AGE**, которая выполняет заодно неявное преобразование типа своих аргументов:

```
1. SELECT AGE ('2011-10-08T01:23:00', '2011-10-07T23:43:00');
```

Для получения числа секунд в интервале воспользуемся функцией

```
1. EXTRACT(EPOCH FROM < interval >)
```

Значение интервала представим последним способом как наиболее коротким. Для получения окончательного решения задачи поделим полученный результат на 60:

```
1. SELECT  EXTRACT(EPOCH  FROM  AGE('2011-10-08T01:23:00',
   '2011-10-07T23:43:00'))/60;
```

В данном случае мы имеем результат, выраженный целым числом, поскольку оба значения времени в задаче не содержали секунд. В противном случае, будет получено десятичное число, например: 99.75:

```
1. SELECT  EXTRACT(EPOCH  FROM  AGE  ('2011-10-08T01:23:00',
   '2011-10-07T23:43:15'))/60;
```

Oracle

Oracle тоже не имеет в своем арсенале функции типа DATEDIFF. Кроме того, Oracle не поддерживает стандартное текстовое представление даты/времени.

Мы можем посчитать интервал в минутах, приняв во внимание, что разность дат (значений типа **date**) в Oracle дает в результате число дней (суток). Тогда для вычисления интервала в минутах нужно просто разность дат умножить на 24 (число часов в сутках), а затем на 60 (число минут в часе):

```
1. SELECT (CAST('2011-10-08 01:23:00' AS date) - CAST('2011-
   10-07 23:43:00' AS date))*24*60
2. FROM dual;
```

Подобным образом можно получить и другие временные интервалы.

Функция DATEPART

Синтаксис:

```
1.DATEPART(datepart , date)
```

Эта функция возвращает целое число, представляющее собой указанную аргументом datepart часть заданной вторым аргументом даты (date).

Список допустимых значений аргумента datepart, описанный выше в данном разделе, дополняется еще одним значением

Datepart	Допустимые сокращения
Weekday — день недели	dw

Заметим, что возвращаемое функцией **DATEPART** значение в этом случае (номер дня недели) зависит от настроек, которые можно изменить с помощью оператора **SET DATEFIRST**, устанавливающего первый день недели. Для кого-то понедельник — день тяжелый, а для кого-то — воскресенье. Как и многие региональные настройки, по умолчанию это значение в SQL Server зависит от выбранного языка.

Однако вернемся к **примеру 7.1.3**. В предположении, что время вылета/прилета является кратным минуте, мы можем его определить как сумму часов и минут. Поскольку функции даты/времени работают с целочисленными значениями, приведем результат к наименьшему интервалу — минутам. Итак, время вылета рейса 1123 в минутах:

```
1.SELECT DATEPART(hh, time_out)*60 + DATEPART(mi, time_out)
2.FROM Trip
3.WHERE trip_no = 1123;
```

и время прилета

```
1.
  SELECT DATEPART(hh, time_in)*60 + DATEPART(mi, time_in)
2.FROM Trip
3.WHERE trip_no = 1123;
```

Теперь мы должны сравнить, превышает ли время прилета время вылета. Если это так, следует вычесть из первого второе, чтобы получить продолжительность рейса. В противном случае к разности нужно добавить одни сутки (24*60 = 1440 минут).

```
1. SELECT CASE
2.  WHEN time_dep >= time_arr
3.  THEN time_arr - time_dep + 1440
4.  ELSE time_arr - time_dep
5.  END dur
6. FROM  (SELECT  DATEPART(hh,  time_out)*60  +  DATEPART(mi,
   time_out) time_dep,
7.  DATEPART(hh,   time_in)*60   +   DATEPART(mi,   time_in)
   time_arr
8.  FROM Trip
9.  WHERE trip_no = 1123
10.        ) tm;
```

Здесь, чтобы не повторять длинные конструкции в операторе **CASE**, использован подзапрос. Конечно, результат получился достаточно громоздким, зато абсолютно корректным в свете сделанных к этой задаче замечаний.

Пример 7.1.4

Определить дату и время вылета рейса 1123.

В таблице совершенных рейсов Pass_in_trip содержится только дата рейса, но не время, так как в соответствии с предметной областью каждый рейс может выполняться только один раз в день. Для решения этой задачи нужно к дате, хранящейся в таблице Pass_in_trip, добавить время из таблицы Trip

```
1. SELECT      DISTINCT      pt.trip_no,       DATEADD(mi,
   DATEPART(hh,time_out)*60 +
2.  DATEPART(mi,time_out), date)  [time]
3. FROM Pass_in_trip pt JOIN
4.  Trip t ON pt.trip_no = t.trip_no
5. WHERE t.trip_no = 1123;
```

Выполнив запрос, получим следующий результат

Trip_no	Time
1123	2003-04-05 16:20:00.000
1123	2003-04-08 16:20:00.000

DISTINCT необходим здесь, чтобы исключить возможные дубликаты, поскольку номер и дата рейса дублируются в этой таблице для каждого пассажира данного рейса.

В ряде случаев функцию **DATEPART** можно заменить более простыми функциями. Вот они:

DAY(date) — целочисленное представление дня указанной даты. Эта функция эквивалентна функции DATEPART(dd, date).

MONTH(date) — целочисленное представление месяца указанной даты. Эта функция эквивалентна функции DATEPART(mm, date).

YEAR(date) — целочисленное представление года указанной даты. Эта функция эквивалентна функции DATEPART(yy, date).

MySQL

В MySQL для извлечения каждой составляющей даты имеется соответствующая функция. Так, например, можно получить минуты времени отправления рейсов, которые вылетают в 1-ом часу дня (база данных "Аэропорт"):

Решение 1

```
1.SELECT time_out, MINUTE(time_out)
2.FROM trip
3.WHERE HOUR(time_out) = 12;
```

time_out	min
1900-01-01 12:35:00	35
1900-01-01 12:00:00	0

Компоненту времени можно также получить при помощи функции **EXTRACT**. С помощи этой функции решение задачи можно записать в виде:

Решение 2

```
1. SELECT time_out, EXTRACT(MINUTE FROM time_out) AS MIN
2. FROM trip
3. WHERE EXTRACT(HOUR FROM time_out) = 12;
```

Дополнительно с помощью функции EXTRACT можно получить составные компоненты даты/времени, например, год и месяц. Решим такую задачу.

Посчитать количество окрасок по месяцам с учетом года (база данных "Окраска").

Решение 3

```
1. SELECT YEAR(b_datetime) AS y, MONTH(b_datetime) AS m,
2. EXTRACT(YEAR_MONTH FROM b_datetime) AS ym, COUNT(*) AS
   qty
3. FROM utb
4. GROUP BY EXTRACT(YEAR_MONTH FROM b_datetime);
```

y	m	ym	qty
2000	1	200001	1
2001	1	200101	1
2002	1	200201	1

2002	6	200206	1
2003	1	200301	69
2003	2	200302	7
2003	3	200303	2
2003	4	200304	2
2003	5	200305	2
2003	6	200306	2

Полный список допустимых компонент можно найти, например, здесь.

Примечание:

Здесь уместно будет отметить довольно вольную трактовку группировки в MySQL, при которой в списке столбцов предложения SELECT могут присутствовать столбцы с детализированными данными, отсутствующими в предложении GROUP BY. Очевидно, что агрегаты здесь подразумеваются, т.к., в противном случае, отсутствует однозначная интерпретация операции. Предполагаю, что здесь неявно используются функция MIN или MAX, но я даже не буду это специально выяснять, т.к. предпочитаю в подобных случаях следовать стандарту.

PostgreSQL

Функций типа Year, Month и т.д. в PostgreSQL, насколько мне известно, нет. Однако есть функция EXTRACT, и второе решение первой задачи, которое мы написали для MySQL, будет работать и под этой СУБД:

```
1.SELECT time_out, EXTRACT(MINUTE FROM time_out) AS MIN
2.FROM trip
3.WHERE EXTRACT(HOUR FROM time_out) = 12;
```

Имеется также функция, аналогичная **DATEPART** в MSSQL. Различия в синтаксисе, надеюсь, станут очевидны из примера решения той же задачи с использованием этой функции:

```
1.SELECT time_out, DATE_PART('MINUTE', time_out) AS MIN
2.FROM trip
3.WHERE DATE_PART('HOUR', time_out) = 12;
```

Перейдем к решению второй задачи. Составные компоненты не поддерживаются в функции EXTRACT для PostgreSQL. Поэтому можно использовать "классическую" групировку по двум компонентам - году и месяцу:

Решение 4

```
1.SELECT EXTRACT(YEAR FROM b_datetime) AS y,
2.EXTRACT(MONTH FROM b_datetime) AS m, COUNT(*) AS qty
3.FROM utb
4.GROUP BY EXTRACT(YEAR FROM b_datetime), EXTRACT(MONTH
  FROM b_datetime);
```

Тем не менее, мы можем реализовать идею составной компоненты при помощи функции форматирования даты **TO_CHAR**:

```
1.SELECT TO_CHAR(b_datetime, 'YYYYMM') AS ym, COUNT(*) AS
  qty
2.FROM utb
3.GROUP BY TO_CHAR(b_datetime, 'YYYYMM');
```

Результат будет аналогичен результату решения 3 за отсутствием первых двух столбцов, которые, разумеется, можно добавить в вывод, включив их одновременно в предложение GROUP BY, как это сделано в решении 4.

Функция DATENAME

Функция **DATENAME**(datepart, date) возвращает символьное представление составляющей (datepart) указанной даты (date). Аргумент, определяющий составляющую даты, может принимать одно из значений, перечисленных в таблице в **начале этой главы**.

Это дает нам простую возможность конкатенировать компоненты даты, получая любой нужный формат представления. Например, конструкция

```
1. SELECT DATENAME(weekday, '20031231')+', ' + DATENAME(day, '20031231') +
2. ' ' + DATENAME(month, '20031231') + ' ' + DATENAME(year,'20031231');
```

даст нам следующий результат

Wednesday, 31 December 2003

Следует отметить, что данная функция выявляет отличие значений day и dayofyear аргумента datepart. Первый дает символьное представление дня указанной даты, в то время как второй дает символьное представление этого дня от начала года. То есть

```
1. SELECT DATENAME(day, '20031231');
```

даст нам 31, в то время как для

```
1. SELECT DATENAME(dayofyear, '20031231');
```

результатом будет 365.

Первый день недели

Задача. Определить дату, на которую выпал первый понедельник января 2013 года.

При некоторых предположениях решить эту задачу можно следующим образом:

```
1.WITH num(n) AS(/* с помощью рекурсивного CTE создаем таблицу со столбцом n
2.                  и значениями от 0 до 6 */
3.SELECT 0
4.UNION ALL
5.SELECT n+1 FROM num
6.WHERE n < 6),
7.dat AS (/* создаем таблицу с датами от 1 до 7 января 2013 года */
8.SELECT DATEADD(dd, n, CAST('2013-01-01' AS DATETIME)) AS day FROM num
9.)
10.     SELECT day FROM dat WHERE DATEPART(dw, day) = 1; /* выбираем день, соответствующий
11.                               первому дню недели */
```

Предположение, о котором говорилось выше, состоит в том, что первым днем недели считается понедельник. Однако если вы выполните этот запрос на сайте sql-ex.ru, то получите

day
2013-01-06

А это - воскресенье. Причина в том, что настройки на сайте полагают первым днем недели воскресенье. А можно ли написать решение, которое не зависело бы от настроек сервера? Попробуем сделать так: будем не номер дня использовать при фильтрации, а его название (поменяется лишь последняя строка, но я повторяю весь запрос с тем, чтобы его можно было выполнить без редактирования):

```
1.WITH num(n) AS (SELECT 0
2.UNION ALL
3.SELECT n+1 FROM num
4.WHERE n < 6),
5.dat AS (
6.SELECT DATEADD(dd, n, CAST('2013-01-01' AS DATETIME))
  AS day FROM num
7.)
8.SELECT day FROM dat WHERE DATENAME(dw, day) = 'monday';
```

day
2013-01-07

Теперь правильно, но будет ли этот запрос всегда верен, если "зашить" его в код приложения? Ответ - нет. Если я поменяю языковые настройки:

```
1.SET LANGUAGE russian;
```

то получу пустой набор строк, поскольку в этом случае последнюю строку запроса следовало бы написать так

```
1.SELECT day FROM dat WHERE DATENAME(dw, day) =
  N'понедельник';
```

Функция @@DATEFIRST

@@DATEFIRST возвращает число, которое определяет первый день недели, установленный для текущей сессии. При этом 1 соответствует понедельнику, а 7, соответственно, воскресенью. Т.е. если

```
1.SELECT @@DATEFIRST;
```

возвращает 7, то первым днем недели считается воскресенье (соответствует текущим настройкам на сайте).

Для того, чтобы решение нашей задачи не зависело от значения, установленного для первого дня недели, воспользуемся функцией @@DATEFIRST. Сделать это можно, например, так

```
1.WITH num(n) AS(
2.SELECT 0
3.UNION ALL
4.SELECT n+1 FROM num
5.WHERE n < 6),
6.dat AS (
7.SELECT DATEADD(dd, n, CAST('2013-01-01' AS DATETIME))
  AS day FROM num
8.)
9.SELECT day, DATENAME(dw, day) week_day FROM dat WHERE
  DATEPART(dw, day) =
10.     1+(8-@@DATEFIRST) % 7;
```

day	week_day
2013-01-07	Monday

Чтобы изменить значение первого дня недели (на время текущей сессию), можно использовать оператор SET DATEFIRST. Так, если выполнить следующий код

```
1.SET DATEFIRST 1;
2.SELECT @@DATEFIRST;
```

то мы получим значение 1, т.е. неделя теперь начинается с понедельника.

Настройку первого дня недели может также поменять изменение языка сессии. Например, если мы выполним

```
1.SET LANGUAGE us_english;
2.SELECT @@DATEFIRST;
```

то опять получим 7 (воскресенье), поскольку это значение DATEFIRST принято по умолчанию для английского (американского) языка. В то время как выбор русского языка (SET LANGUAGE russian;) сделает первым днем недели понедельник.

Однако вы можете поменять параметры языка и первого дня недели одновременно, чтобы, скажем, получить английский язык, и при этом педеля будет начинаться с понедельника:

```
1. SET DATEFIRST 1;
2. SET LANGUAGE us_english;
3. SELECT @@DATEFIRST;
```

Языковые настройки влияют, в частности, на символьное представление компонентов даты/времени. Сравните, например, результаты:

```
1. DECLARE @dt DATE = '2012-12-17'; -- 17 декабря 2012 года
2. SET LANGUAGE us_english;
3. SELECT DATENAME(DW, @dt) AS day_of_week, DATENAME(MONTH,@dt) AS month;
```

day_of_week	month
Monday	December

```
1. SET LANGUAGE russian;
2. SELECT DATENAME(DW, @dt) AS day_of_week, DATENAME(MONTH, @dt) AS month;
```

day_of_week	month
понедельник	Декабрь

```
1. SET LANGUAGE german;
2. SELECT DATENAME(DW, @dt) AS day_of_week, DATENAME(MONTH, @dt) AS month;
```

day_of_week	month
Montag	Dezember

Вывод. Если вы хотите, чтобы зависящие от настроек сервера/базы запросы всегда давали верный результат, можно задавать необходимые настроечные параметры для сессии, в которой эти запросы выполняются, или же проверять в каждом подобном запросе значения соответствующих параметров, в частности, с помощью функции @@DATEFIRST.

Особенности: MSSQL

Решить эту задачу (Определить дату, на которую выпал первый понедельник января 2013 года.) можно и без применения функции @@DATEFIRST, воспользовавшись методом, предложенным Ициком Бен-Ганом [8]. Идея решения заключается в том, что:

1. Первое января 1900г. было понедельником.

2. Количество дней между двумя одинаковыми днями недели всегда кратно семи.

```
1.declare  @anchor_date datetime
2.declare  @reference_date datetime
3.SELECT                         @anchor_date='19000101',
  @reference_date='20130505'
4.SELECT DATEADD(day, DATEDIFF(day, @anchor_date,
5.DATEADD(year,         DATEDIFF(year,          '19000101',
  @reference_date), '19000101') - 1) /7*7 + 7,
6.@anchor_date);
```

Особенности: MSSQL

Можно упомянуть еще один способ решения этой задачи без @@DATEFIRST, который использует сравнение любой из функций DATEPART/DATENAME с такой же функцией от даты, которая ЗАВЕДОМО является понедельником. Например, от уже упомянутого 1-го января 1900г.

Тогда первоначальный запрос из этой главы выглядел бы так:

```
1.SELECT  day  FROM  dat  WHERE DATEPART(dw,  day)  =
  DATEPART(dw, '19000101');
```

или так

```
1. SELECT day FROM dat WHERE DATENAME(dw, day) = DATENAME(dw, '19000101');
```

Функция DATEFROMPARTS

Функция **DATEFROMPARTS** появилась в SQL Server версии 2012.

У функции DATEFROMPARTS 3 целочисленных аргумента, представляющих собой год, месяц и день, а возвращаемое значение есть соответствующая дата типа DATE. Она даёт удобный способ формирования даты, когда её компоненты хранятся отдельно или передаются с клиента.

Например,

```
1. SELECT DATEFROMPARTS(2017, 5, 25);
```

вернёт

2017-05-25

Получить дату 25 числа текущего месяца можно так

```
1. SELECT DATEFROMPARTS(YEAR(CURRENT_TIMESTAMP), MONTH(CURRENT_TIMESTAMP), 25);
```

Помимо даты можно "собирать" значения типа времени и временной метки (datetime), используя компоненты времени - часы, минуты, секунды и доли секунды. Например, следующий запрос

```
1. SELECT TIMEFROMPARTS(9, 38, 59, 998, 7);
```

даст значение типа TIME:

09:38:59.0000998

Обратите внимание на последний параметр, который указывает на число знаков в представлении долей секунды. Этот целочисленный параметр может принимать значение от 0 до 7, но не может быть меньше числа цифр в представлении долей секунды.

Например,

```
1.SELECT TIMEFROMPARTS(9, 38, 59, 998, 3);
```

вернет

09:38:59.998

в то время как

```
1.SELECT TIMEFROMPARTS(9, 38, 59, 998, 2);
```

вернет ошибку:

Cannot construct data type time, some of the arguments have values which are not valid.

(Не удалось сконструировать тип данных time. Некоторые аргументы имеют недопустимые значения.)

Сравните с

```
1.SELECT TIMEFROMPARTS(9, 38, 59, 098, 2), TIMEFROMPARTS(9,
  38, 59, 098, 3);
```

Результат

09:38:59.98	09:38:59.098

Функция DATETIMEFROMPARTS формирует значение типа DATETIME. Она имет семь целочисленных параметров: год, месяц, день, часы, минуты, секунды, миллисекунды. При этом миллисекунды округляются с точностью до одного из значений: .000, .003, .007.

```
1.SELECT DATETIMEFROMPARTS(2017, 5, 13, 9, 38, 59, 998),
2. DATETIMEFROMPARTS(2017, 5, 13, 9, 38, 59, 999),
3.DATETIMEFROMPARTS(2017, 5, 13, 9, 38, 59, 993);
```

Результат

2017-05-13 09:38:59.997	2017-05-13 09:39:00.000	2017-05-13 09:38:59.993

Функции работы со строками в MS SQL SERVER

Вот полный перечень функций работы со строками, взятый из BOL:

ASCII	NCHAR	SOUNDEX
CHAR	PATINDEX	SPACE
CHARINDEX	REPLACE	STR
DIFFERENCE	QUOTENAME	STUFF
LEFT	REPLICATE	SUBSTRING
LEN	REVERSE	UNICODE
LOWER	RIGHT	UPPER
LTRIM	RTRIM	

Функции ASCII и CHAR

Начнем с двух взаимно-обратных функций — **ASCII** и **CHAR**:

Функция **ASCII** возвращает ASCII-код крайнего левого символа строкового выражения, являющегося аргументом функции.

Вот, например, как можно определить, сколько имеется разных букв, с которых начинаются названия кораблей в таблице Ships:

```
1.SELECT COUNT(DISTINCT ASCII(name))
2.FROM Ships;
```

Результат — 11. Чтобы выяснить, какие это буквы, мы можем применить функцию **CHAR**, которая возвращает символ по известному ASCII-коду (от 0 до 255):

```
1.SELECT DISTINCT CHAR(ASCII(name))
2.FROM Ships
3.ORDER BY 1;
```

Следует отметить, что аналогичный результат можно получить проще с помощью еще одной функции — **LEFT**.

Функция LEFT

Функция имеет следующий синтаксис:

```
1.LEFT(строковое выражение, целочисленное выражение)
```

и вырезает заданное вторым аргументом число символов слева из строки, являющейся первым аргументом. Итак,

```
1.SELECT DISTINCT LEFT(name, 1)
2.FROM Ships
3.ORDER BY 1;
```

А вот как, например, можно получить таблицу кодов всех алфавитных символов:

```
1.SELECT CHAR(ASCII('a')+ num-1) letter, ASCII('a')+ num -
  1 [code]
2.FROM (SELECT 5*5*(a-1)+5*(b-1) + c AS num
3. FROM (SELECT 1 a UNION ALL SELECT 2 UNION ALL SELECT 3
4. UNION ALL SELECT 4 UNION ALL SELECT 5
5. ) x CROSS JOIN
6. (SELECT 1 b UNION ALL SELECT 2 UNION ALL SELECT 3
7. UNION ALL SELECT 4 UNION ALL SELECT 5
8. ) y CROSS JOIN
9. (SELECT 1 c UNION ALL SELECT 2 UNION ALL SELECT 3
10.       UNION ALL SELECT 4 UNION ALL SELECT 5
11.       ) z
12.       ) x
13.      WHERE ASCII('a')+ num -1 BETWEEN ASCII('a') AND
  ASCII('z');
```

Здесь используется алгоритм генерации числовой последовательности, изложенный в главе 8.

Как известно, коды строчных и прописных букв отличаются. Поэтому, чтобы получить полный набор без переписывания запроса, достаточно просто дописать к вышеприведенному коду аналогичный:

```
1.UNION
2.SELECT CHAR(ASCII('A')+ num-1) letter, ASCII('A')+ num -
  1 [code]
3.FROM (SELECT 5*5*(a-1)+5*(b-1) + c AS num
4. FROM (SELECT 1 a UNION ALL SELECT 2 UNION ALL SELECT 3
5. UNION ALL SELECT 4 UNION ALL SELECT 5
6. ) x CROSS JOIN
7. (SELECT 1 b UNION ALL SELECT 2 UNION ALL SELECT 3
8. UNION ALL SELECT 4 UNION ALL SELECT 5
9. ) y CROSS JOIN
10.       (SELECT 1 c UNION ALL SELECT 2 UNION ALL SELECT 3
11.       UNION ALL SELECT 4 UNION ALL SELECT 5
12.       ) z
13.       ) x
14.      WHERE ASCII('A')+ num -1 BETWEEN ASCII('A') AND
  ASCII('Z')
```

Чтобы таблица выглядела более патриотично, достаточно заменить латинские буквы a и A на неотличимые на вид русские — а и А, а z и Z на я и Я. Вот только буквы «ё» вы не увидите в этой таблице, так как в кодовой таблице **ASCII** эти символы лежат отдельно, что легко проверить:

```
1. SELECT ASCII('ё')
2. UNION ALL
3. SELECT ASCII('Ё');
```

Полагаем, что для вас не составит сложности добавить эту букву в таблицу при необходимости.

Рассмотрим теперь задачу определения нахождения искомой подстроки в строковом выражении. Для этого могут использоваться две функции — **CHARINDEX** и **PATINDEX**. Обе они возвращают начальную позицию (позицию первого символа подстроки) подстроки в строке.

Функции CHARINDEX и PATINDEX

Функция CHARINDEX имеет синтаксис:

```
1. CHARINDEX( искомое_выражение, строковое_выражение [,
   стартовая_позиция ])
```

Здесь необязательный целочисленный параметр стартовая_позиция определяет позицию в строковом выражении, начиная с которой выполняется поиск. Если этот параметр опущен, поиск выполняется от начала строкового_выражения. Например, запрос

```
1. SELECT name
2. FROM Ships
3. WHERE CHARINDEX('sh', name) > 0;
```

будет выводить те корабли, в которых имеется сочетание символов «sh». Здесь используется тот факт, что если искомая строка не будет обнаружена, то функция CHARINDEX возвращает 0. Результат выполнения запроса будет содержать следующие корабли:

name
Kirishima
Musashi
Washington

Следует отметить, что если искомая подстрока либо строковое выражение есть **NULL**, то результатом функции тоже будет **NULL**.

Следующий пример определяет позиции первого и второго вхождения символа a в имени корабля California:

```
1.SELECT CHARINDEX('a',name) first_a,
2. CHARINDEX('a', name, CHARINDEX('a', name)+1) second_a
3.FROM Ships
4.WHERE name='California';
```

Обратите внимание, что при определении второго символа в функции **CHARINDEX** используется стартовая позиция, которой является позиция следующего за первой буквой а символа — *CHARINDEX('a', name)+1*. Правильность результата — 2 и 10 — легко проверить.

Функция **PATINDEX** имеет следующий синтаксис:

```
1.PATINDEX( '%образец%' , строковое_выражение )
```

Главное отличие этой функции от **CHARINDEX** заключается в том, что поисковая строка может содержать подстановочные знаки — % и _. При этом концевые знаки % являются обязательными. Например, использование этой функции в первом примере будет иметь вид

```
1. SELECT name
2. FROM Ships
3. WHERE PATINDEX('%sh%', name) > 0;
```

А вот, например, как можно найти имена кораблей, которые содержат последовательность из трех символов, первый и последний из которых есть *e*:

```
1. SELECT name
2. FROM Ships
3. WHERE PATINDEX('%e_e%', name) > 0;
```

Результат выполнения этого запроса выглядит следующим образом:

Name
Revenge
Royal Sovereign

Функция RIGHT

Парная к **LEFT** функция **RIGHT** возвращает заданное число символов справа из строкового выражения:

```
1. RIGHT(строковое выражения, число символов)
```

Вот, например, как можно определить имена кораблей, которые начинаются и заканчиваются на одну и ту же букву:

```
1. SELECT name
```

```
2.FROM Ships
3.WHERE LEFT(name, 1) = RIGHT(name, 1);
```

То, что в результате мы получим пустой результирующий набор, означает, что таких кораблей в базе данных нет. Давайте возьмем комбинацию значений — класс и имя корабля.

Соединение двух строковых значений в одно называется конкатенацией, и в SQL Server для этой операции используется знак «+» (в стандарте «||»). Итак,

```
1.SELECT *
2.FROM (SELECT class +' '+ name AS cn
3. FROM Ships
4. ) x
5.WHERE LEFT(cn, 1) = RIGHT(cn, 1);
```

Здесь мы разделяем пробелом имя класса и имя корабля. Кроме того, чтобы не повторять всю конструкцию в качестве аргумента функции, используем подзапрос. Результат будет иметь вид:

Cn
Iowa Missouri
North Carolina Washington

А если строковое выражение будет содержать лишь одну букву? Запрос выведет ее. В этом легко убедиться, написав:

```
1.SELECT *
2.FROM (SELECT class +' '+ name AS cn
3. FROM Ships
4. UNION ALL
5. SELECT 'a' AS cn
6. ) x
7.WHERE LEFT(cn, 1) = RIGHT(cn, 1);
```

Чтобы исключить этот случай, можно воспользоваться еще одной полезной функцией **LEN**.

Функция LEN

Функция **LEN**(строковое выражение) возвращает число символов в строке, задаваемой строковым выражением. Ограничимся случаем, когда число символов больше единицы:

```
1.SELECT *
2.FROM (SELECT class +' '+ name AS cn
3. FROM Ships
4. UNION ALL
5. SELECT 'a' AS nc
6. ) x
7.WHERE LEFT(cn, 1) = RIGHT(cn, 1) AND
8. LEN(cn) > 1;
```

Реализация функции LEN() в MS SQL Server имеет одну особенность, а именно, она не учитывает концевые пробелы.

Давайте выполним следующий код:

```
1.DECLARE @chr AS CHAR(12), @vchr AS VARCHAR(12);
2.SELECT @chr = 'abcde' + REPLICATE(' ', 5), @vchr =
  'abcde'+REPLICATE(' ', 5);
3.SELECT LEN(@chr), LEN(@vchr);
4.SELECT DATALENGTH(@chr), DATALENGTH(@vchr);
```

5	5
12	10

Функция **REPLICATE** добавляет справа к константе 'abcde' пять пробелов, которые не учитываются функцией LEN, - в обоих случаях мы получаем 5.

Функция **DATALENGTH** возвращает число байтов в строковом представлении и демонстрирует различие в типах данных CHAR и VARCHAR. DATALENGTH возвращает 12 для типа CHAR(12), и 10 - для VARCHAR(12).

Как и следовало ожидать, DATALENGTH для переменной типа VARCHAR вернуло фактическую длину строковой переменной. Но почему для переменной типа CHAR результат оказался равным 12? Дело в том, что CHAR является типом данных фиксированной длины. Если значение переменной меньше объявленной длины, а мы использовали CHAR(12), значение переменной будет дополнено концевыми пробелами, чтобы "выровнять" длину до 12 символов.

На сайте имеются задачи, в которых требуется упорядочить (найти максимум и т. д.) в числовом порядке значения, представленные в текстовом формате. Например, номер места в самолете (2d) или скорость привода CD-ROM (24x). Проблема заключается в том, что текст сортируется так (по возрастанию):

11a
1a
2a

Действительно,

```
1.SELECT '1a' AS place
2.UNION ALL
3.SELECT '2a'
4.UNION ALL
5.SELECT '11a'
6.ORDER BY 1;
```

Если же требуется упорядочить места в самолете в порядке следования рядов, то порядок должен быть таким:

1a

2a
11a

Чтобы добиться такого порядка, нужно выполнить сортировку по числовым значениям, присутствующим в тексте. Можно использовать такой алгоритм:

1. Извлечь число из строки.
2. Привести его к числовому формату.
3. Выполнить сортировку по приведенному значению.

Так как нам известно, что буква только одна, то для извлечения числа из строки можно воспользоваться следующей конструкцией, которая не зависит от числа цифр в номере места:

```
1.LEFT(place, LEN(place) - 1)
```

Если только этим и ограничиться, то получим

place
1a
11a
2a

Приведение к числовому формату может быть следующим:

```
1.CAST (LEFT(place, LEN(place) - 1) AS INT)
```

Осталось выполнить сортировку:

```
1.SELECT *
2.FROM (SELECT '1a' AS place
3. UNION ALL
4. SELECT '2a'
5. UNION ALL
6. SELECT '11a'
```

```
7. ) x
8.ORDER BY CAST(LEFT(place, LEN(place) - 1) AS INT);
```

Что и требовалось доказать.

Ранее мы для извлечения числа из текстовой строки пользовались функцией **LEFT**, так как нам было известно априори, какое число символов нужно убрать справа (один). А если же нужно извлечь строку из подстроки не по известной позиции символа, а по самому символу? Например: извлечь все символы до первой буквы x (значение скорости привода CD-ROM).

В этом случае мы можем использовать также уже рассмотренную ранее функцию **CHARINDEX**, которая позволит определить неизвестную позицию символа:

```
1.SELECT model, LEFT(cd, CHARINDEX('x', cd) -1)
2.FROM PC;
```

Функция LEN(), концевые пробелы и уникальность

Недавно я столкнулся с тем, что не смог добавить два значения типа VARCHAR, отличающиеся только концевым пробелом в столбец составного первичного ключа (SQL Server 2008). Возможно, этот факт для кого-то является очевидным, но мне показалось странным, что в принципе разные значения считаются дубликатами. Со значениями типа CHAR(n), который имеет фиксированную длину, все понятно, т.к. короткие строки дополняются пробелами до длины n. Поэтому вводимые значения, которые отличаются лишь концевыми пробелами, оказываются неразличимыми. Но концевой пробел в значении типа VARCHAR является как бы обычным символом.

Вот простой эксперимент.

```
1.CREATE TABLE Test_Trailing_Space2
2. ( num int NOT NULL, name VARCHAR(10) NOT NULL,
3. PRIMARY KEY(num, name) );
4.GO
5.INSERT INTO Test_Trailing_Space2 VALUES(1, 'John');
6.INSERT INTO Test_Trailing_Space2 VALUES(1, 'John ');
7.GO
```

Вторая строка не будет вставлена в таблицу, при этом будет получено сообщение о нарушении ограничения первичного ключа. Т.е. вторая строка считается дубликатом первой. Может быть дело в том, что при вставке концевой пробел был отсечен. Но нет, вставим другую уникальную строку с концевым пробелом и проверим наличие в ней концевого пробела:

```
1.INSERT INTO Test_Trailing_Space2 VALUES(2, 'John ');
2.GO
3.SELECT *, LEN(name) len1,DATALENGTH(name) len2
4.FROM Test_Trailing_Space2;
```

Результат:

num	name	len1	len2
1	John	4	4
2	John	4	5

Значение в столбце len2 показывает, что пробел присутствует в данных, но, тем не менее, значения 'John' и 'John ' считаются дубликатами:

```
1.SELECT DISTINCT name
2.FROM Test_Trailing_Space2;
```

name
John

Очевидно, что все дело в функции **LEN()**, которая, как известно, не учитывает концевые пробелы. Я не нашел этой информации в BOL, но, видимо, именно эта функция используется при сравнении значений любых строковых типов. Мне стало интересно, как ведут себя другие СУБД в этом случае, и я повторил эксперимент для MySQL и PostgreSQL. Были получены следующие результаты.

MySQL (версия 5.0)

```
1. SELECT *, OCTET_LENGTH(name) AS len1, LENGTH(name) AS len2
2. FROM Test_Trailing_Space2;
```

1	John	4	4
2	John	5	5

```
1. SELECT DISTINCT name FROM Test_Trailing_Space2;
```

John

PostgreSQL (версия 8.3.6)

```
1. SELECT *, OCTET_LENGTH(name) AS len1, LENGTH(name) AS len2
2. FROM Test_Trailing_Space2;
```

1	"John"	4	4
2	"John "	5	5

```
1. SELECT DISTINCT name
2. FROM Test_Trailing_Space2;
```

"John"
"John "

Как видно, и MySQL, и PostgreSQL учитывают пробел как в числе символов, так и в числе байтов, используемых для хранения строкового значения. При этом MySQL и SQL Server, в отличие от PostgreSQL, считают строки, различающиеся лишь концевыми пробелами, дубликатами. Естественно, PostgreSQL позволяет вставить и такую строку в рассматриваемую таблицу:

```
1. INSERT INTO Test_Trailing_Space2 VALUES(1, 'John ');
```

Вместо выводов

Я далек от того, чтобы высказываться относительно правильности той или иной реализации и, тем более, спорить о том, какая СУБД лучше. Я считаю, что нужно знать досконально ту СУБД, которую вы используете в своей профессиональной деятельности. Изучайте документацию и все подвергайте проверке, не полагаясь на интуицию и «здравый» смысл.

Функция SUBSTRING

Функция **SUBSTRING**(выражение, начальная позиция, длина) позволяет извлечь из выражения его часть заданной длины, начиная от заданной начальной позиции. Выражение может быть символьной или бинарной строкой, а также иметь тип text или image. Например, если нам потребуется получить три символа в названии корабля, начиная со второго символа, то сделать это без помощи функции **SUBSTRING** будет не просто. А так мы напишем:

```
1.SELECT name, SUBSTRING(name, 2, 3)
2.FROM Ships;
```

В случае, когда нужно извлечь все символы, начиная с заданного, мы также можем применить эту функцию. Например,

```
1. SELECT name, SUBSTRING(name, 2, LEN(name))
2. FROM Ships;
```

даст нам все символы в названиях кораблей от второй буквы в имени. Обратите внимание на то, что для указания числа извлекаемых символов мы использовали функцию *LEN(name)*, которая возвращает число символов в имени. Понятно, что поскольку нам нужны символы, начиная со второго, то их число будет меньше общего количества символов в имени. Однако это не вызывает ошибки, поскольку если указанное число символов превышает возможное число, то будут извлечены все символы до конца строки. Поэтому мы берем их с запасом, не утруждая себя вычислениями.

Функция REVERSE

Эта функция переворачивает строку, как бы читая ее справа налево. То есть результатом запроса

```
1. SELECT REVERSE('abcdef');
```

будет *fedcba*. Если бы отсутствовала функция **RIGHT**, то запрос

```
1. SELECT RIGHT('abcdef', 3);
```

можно было бы равносильно заменить запросом

```
1. SELECT REVERSE(LEFT(REVERSE('abcdef'), 3));
```

Польза этой функции в следующем. Пусть нам требуется определить позицию не первого, а последнего вхождения некоторого символа (или последовательности символов) в строке. Вспомним пример, в котором мы определяли позицию первого символа а в названии корабля California:

```
1. SELECT CHARINDEX('a', name) first_a
2. FROM Ships
3. WHERE name = 'California';
```

Определим теперь позицию последнего вхождения в это название символа a. Функция **CHARINDEX**(*'a',* **REVERSE**(*name))* позволит найти эту позицию, но справа. Для получения позиции этого же символа слева достаточно написать

```
1. SELECT LEN(name) + 1 - CHARINDEX('a', REVERSE(name))
   first_a
2. FROM Ships
3. WHERE name = 'California';
```

Функция REPLACE

Функция

```
1. REPLACE(строка1, строка2, строка3)
```

заменяет в *строке1* все вхождения *строки2* на *строку3*. Эта функция, безусловно, полезна в операторах обновления (UPDATE), если нужно изменить (исправить) содержимое столбца. Пусть, например, нужно заменить все пробелы дефисом в названиях кораблей. Тогда можно написать:

```
1. UPDATE Ships
2. SET name = REPLACE(name, ' ', '-');
```

Примечание:

Этот пример можно выполнить на странице с упражнениями DML, где разрешаются запросы на изменение данных.

Однако эту функцию можно применять и в более нетривиальных случаях. Давайте определим, сколько раз в названии корабля используется буква «a».

Идея проста: заменим каждую искомую букву двумя любыми символами, после чего посчитаем разность длин полученной и искомой строки. Итак,

```
1. SELECT name, LEN(REPLACE(name, 'a', 'aa')) - LEN(name)
2. FROM Ships;
```

А если нам нужно определить число вхождений произвольной последовательности символов, скажем, передаваемой в качестве параметра в хранимую процедуру? Примененный выше алгоритм в этом случае следует дополнить делением на число символов в искомой последовательности:

```
1. DECLARE @str AS VARCHAR(100);
2. SET @str='ma';
3. SELECT name, (LEN(REPLACE(name, @str, @str + @str)) -
   LEN(name))/LEN(@str) FROM Ships;
```

Следует быть весьма осторожным с применением данного метода к нахождению числа пробелов. Помимо того, что функция LEN не учитывает концевые пробелы, результат будет зависеть от типа данных.

В строке `World Wide Web` 14 символов. Заменим теперь каждый пробел двухсимвольной строкой, и подсчитаем длину до замены и после:

```
1. declare @w char(50) ='World Wide Web';
2. SELECT len(REPLACE(@w,' ','xx')) after, len(@w) before;
```

after	before
88	14

Мы хотели добавить 2 символа, а получили 88 вместо 16. Дело в том, что тип CHAR является точным типом, а это означает, что длина любой строки этого типа будет иметь указанный размер. Мы задали 50 символов, следовательно,

строковое значение будет дополняться концевыми пробелами для выравнивания его до 50 символов, Итого, дополнительно получим (50-14)x2 = 72, и всего 72 + 16 = 88.

Вот если бы мы описали переменную типом VARCHAR(50), то получили желаемый результат:

after	before
16	14

Функции REPLICATE и STUFF

Для удвоения числа искомых символов здесь применялась конкатенация — *@str+@str*. Однако для этой цели можно использовать еще одну функцию — **REPLICATE**, которая повторяет первый аргумент такое число раз, которое задается вторым аргументом.

```
1. SELECT name, (LEN(REPLACE(name, @str, REPLICATE(@str,
   2))) —
2.  LEN(name))/LEN(@str)
3. FROM Ships;
```

То есть мы повторяем дважды подстроку, хранящуюся в переменной @str.

Если же нужно заменить в строке не определенную последовательность символов, а заданное число символов, начиная с некоторой позиции, то проще выбрать функцию **STUFF**:

```
1. STUFF(строка1, стартовая позиция, L , строка2)
```

Эта функция заменяет подстроку длиной L, которая начинается со стартовой позиции в строке1 на строку2.

Пример 7.2.1

Изменить имя корабля: оставив в его имени 5 первых символов, дописать «_» (нижнее подчеркивание) и год спуска на воду. Если в имени менее 5 символов, дополнить его пробелами.

Можно решать эту задачу с помощью разных функций. Мы же попытаемся это сделать с помощью функции **STUFF**.

В первом приближении напишем (ограничимся запросом на выборку):

```
1. SELECT name, STUFF(name, 6, LEN(name), '_'+launched)
2. FROM Ships;
```

В качестве третьего аргумента (количества символов для замены) мы используем **LEN**(*name*), так как нам нужно заменить все символы до конца строки, поэтому мы берем с запасом — исходное число символов в имени. И все же этот запрос вернет ошибку. Причем дело не в третьем аргументе, а в четвертом, где выполняется конкатенация строковой константы и числового столбца. Ошибка приведения типа. Для преобразования числа к его строковому представлению можно воспользоваться еще одной встроенной функцией — **STR**.

Параметр *L* функции STUFF целочисленный; это значит, что он может принимать отрицательные значения и 0. Для отрицательных значений функция STUFF вернет NULL, как и для случая, когда второй параметр превышает длину *строки1*. Нуль же означает вставку *строки2* в *строку1*, начиная с позиции, заданной вторым параметром.

Пример 7.2.2

Добавить разделитель "-" в строковое представление даты в формате YYYYMMDD

```
1. SELECT STUFF(STUFF('20121119',5,0,'-'),8,0,'-');
```

Функции STR, SPACE, LTRIM и RTRIM

Функция **STR**() преобразует число к его символьному представлению:

```
1. STR(число с плавающей точкой [, длина [, число десятичных
   знаков ] ] )
```

При этом преобразовании выполняется округление, а длина задает длину результирующей строки. Например,

STR(3.3456, 5, 1)	3.3
STR(3.3456, 5, 2)	3.35
STR(3.3456, 5, 3)	3.346
STR(3.3456, 5, 4)	3.346

Обратите внимание, что если полученное строковое представление числа меньше заданной длины, то добавляются лидирующие пробелы. Если же результат больше заданной длины, то усекается дробная часть (с округлением); в случае же целого числа получаем соответствующее число звездочек «*»:

```
1. STR(12345,4,0)  ****
```

Кстати, по умолчанию используется длина в 10 символов. Имея в виду, что год представлен четырьмя цифрами, напишем:

```
1. SELECT name, STUFF(name, 6, LEN(name), '_'+STR(launched,
   4))
2. FROM Ships;
```

Уже почти все правильно. Осталось учесть случай, когда число символов в имени менее 6, так как в этом случае функция **STUFF** дает **NULL**. Ну что ж, вытерпим до конца мучения, связанные с использованием этой функции в данном примере, попутно применив еще одну строковую функцию.

Добавим конечные пробелы, чтобы длина имени была заведомо больше 6. Для этого имеется специальная функция SPACE(число пробелов):

```
1. SELECT name, STUFF(name + SPACE(6), 6, LEN(name), '_'+STR(launched,4))
2. FROM Ships;
```

Функции **LTRIM**(строковое выражение) и **RTRIM**(строковое выражение) Отсекают, соответственно, лидирующие и конечные пробелы строкового выражения, которое неявно приводится к типу varchar.

Пусть требуется построить такую строку:

<имя пассажира>_<идентификатор пассажира>

на базе таблицы Passenger. Если мы напишем:

```
1. SELECT name + '_' + CAST(id_psg AS VARCHAR)
2. FROM Passenger;
```

то в результате получим что-то типа:

A _1

Это связано с тем, что столбец name имеет тип *CHAR(30)*. Для этого типа короткая строка дополняется пробелами до заданного размера (у нас 30 символов). Здесь нам как раз и поможет функция **RTRIM**:

```
1. SELECT RTRIM(name) + '_' + CAST(id_psg AS VARCHAR)
2. FROM Passenger;
```

Для усечения концевых пробелов в SQL Server изначально имелось две функции - LTRIM и RTRIM - для усечения пробелов слева и справа соответственно. Чтобы удалить пробелы с обеих сторон строки, последовательно применялись обе функции (в примере для наглядности используется функция DATALENGTH , возвращающая число символов с учетом пробелов):

```
1. declare @s varchar(10) ='  x ' -- слева 2 пробела, справа - 1
2. SELECT datalength(@s) s, datalength(ltrim(@s)) ls,
3.                  datalength(rtrim(@s))       rs,
   datalength(ltrim(rtrim(@s))) bs;
```

s	ls	rs	bs
4	2	3	1

Начиная с версии SQL Server 2017, к ним добавилась новая функция - **TRIM**, которая выполняет усечение с обеих сторон строки-аргумента:

```
1. SELECT datalength(@s) s, datalength(trim(@s)) ts;
```

s	ts
4	1

Кроме того, функция TRIM приобрела дополнительный функционал - возможность усекать произвольные концевые символы.

При этом усекаемые символы можно задавать списком, перечисляя их в произвольном порядке. Усекаться (с обеих сторон) будут все входящие в список символы, пока не появится "посторонний", т.е. не входящий в список. Лучше всего продемонстрировать сказанное на примере.

```
1. declare @s1 varchar(10)='xxaxbxy'
2. SELECT trim('yx' FROM @s1) ts1,trim('x' FROM @s1) ts2;
```

ts1	ts2
axb	axbxy

Конечно, это еще не стандартное поведение, но уже близко. А вот что говорит стандарт относительно функции TRIM:

< trim function> ::= TRIM < left paren> < trim operands> < right paren>

< trim operands> ::= [[< trim specification>] [< trim character>] FROM] < trim source>

< trim source> ::= < character value expression>

< trim specification> ::= LEADING | TRAILING | BOTH

< trim character> ::= < character value expression>

MySQL

В MySQL используется стандартный синтаксис функции TRIM. При этом, в отличие от SQL Server, удаляется указанная подстрока, а не все символы из списка:

```
1.SELECT TRIM(LEADING 'xy' FROM 'xyxybarxyx') ls,
2.TRIM(TRAILING 'xy' FROM 'xyxybarxyx') rs,
3.TRIM(BOTH 'yx' FROM 'xyxybarxyx') bs;
```

ls	rs	bs
barxyx	xyxybarxyx	xyxybarx

PostgreSQL

PostgreSQL сочетает поведение MySQL и SQL Server, т.е. удаляются все символы из списка:

```
1.SELECT TRIM(LEADING 'xy' FROM 'xyxybarxyx') ls,
2.TRIM(TRAILING 'xy' FROM 'xyxybarxyx') rs,
3.TRIM(BOTH 'yx' FROM 'xyxybarxyx') bs;
```

ls	rs	bs
barxyx	xyxybar	bar

Oracle

Oracle допускает усечение только одного символа, а не подстроки или символов из списка:

```
1.SELECT TRIM(LEADING 'x' FROM 'xxybarxyx') ls,
2.TRIM(TRAILING 'x' FROM 'xxybarxyx') rs,
3.TRIM(BOTH 'x' FROM 'xxybarxyx') bs
4.FROM dual;
```

ls	rs	bs
ybarxyx	xxybarxy	ybarxy

Функции LOWER, UPPER, SOUNDEX и DIFFERENCE

Функции **LOWER**(*строковое выражение*) и **UPPER**(*строковое выражение*) преобразуют все символы аргумента, соответственно, к нижнему и верхнему регистру. Эти функции оказываются полезными при сравнении регистрозависимых строк.

Пара интересных функций **SOUNDEX**(*строковое выражение*) и **DIFFERENCE**(*строковое выражение_1*, *строковое выражение_2*) позволяют определить близость звучания слов. При этом **SOUNDEX**() возвращает четырехсимвольный код, используемый для сравнения, а **DIFFERENCE**() собственно и оценивает близость звучания двух сравниваемых строковых выражений. Поскольку эти функции не поддерживают кириллицы, отсылаем

интересующихся к BOL за примерами их использования.

В заключение приведем функции и несколько примеров применения **UNICODE**.

Функции UNICODE и NCHAR

Функция **UNICODE**(*строковое выражение*) возвращает номер в кодировке Unicode первого символа строкового выражения. Функция **NCHAR**(*целое*) возвращает символ по его номеру в кодировке Unicode. Приведем несколько примеров.

```
1.SELECT ASCII('а'), UNICODE('а');
```

Возвращает код **ASCII** и номер русской буквы «а» в кодировке Unicode: 224 и 1072.

```
1.SELECT CHAR(ASCII('а')), CHAR(UNICODE('а'));
```

Здесь мы пытаемся восстановить символ по его коду. Получаем а и **NULL**. **NULL**-значение возвращается потому, что кода 1072 нет в обычной кодовой таблице.

```
1.SELECT CHAR(ASCII('а')), NCHAR(UNICODE('а'));
```

Теперь все нормально, в обоих случаях мы получаем а.

Наконец,

```
1. SELECT NCHAR(ASCII('a'));
```

даст à, так как номер 224 в кодировке Unicode соответствует именно этой букве.

Функция ROUND

Рассмотрим такую задачу.

Посчитать средний размер жесткого диска ПК. Результат представить с двумя знаками после десятичной точки.

Выполнив запрос

```
1. SELECT AVG(hd) AS avg_hd FROM pc;
```

мы получим такой результат:

avg_hd
13.6666666666667

Чтобы выполнить требуемое округление можно воспользоваться функцией **ROUND**:

```
1. SELECT round(AVG(hd),2) AS avg_hd FROM pc;
```

avg_hd
13.67

Второй аргумент этой функции как раз указывает число десятичных знаков результата.

Как видно, результат был округлен по арифметическим правилам. Однако у функции ROUND есть третий необязательный аргумент, который говорит о том, округлять ли результат (значение 0 - принимается по умолчанию) или отбрасывать цифры сверх удерживаемых (ненулевое значение).

Т.е. если мы перепишем наш запрос так:

```
1. SELECT round(AVG(hd),2,1) AS avg_hd FROM pc;
```

то получим другой результат:

avg_hd
13.66

Кстати, округлять можно до любого знака, не только десятичного. Например, чтобы округлять до десятков, сотен и т.д., используются отрицательные значения второго аргумента. Следующий запрос округляет результат до десятков.

```
1. SELECT round(AVG(hd),-1) AS avg_hd FROM pc;
```

avg_hd
10

Следует отметить, что функция ROUND выполняет округление, но не меняет тип результата. Т.е. если аргумент будет иметь тип dec(12,6), то и результат округления будет того же типа, а именно,

avg_hd
13.670000

В этом легко убедиться, выполнив запрос

```
1. SELECT round(CAST(AVG(hd) AS dec(12,6)),2) AS avg_hd FROM
   pc;
```

Поэтому, если вы хотите избавиться от хвостовых нулей, используйте преобразование к нужному вам типу, например, dec(12,2). Тогда нам и функция ROUND не потребуется.

```
1. SELECT CAST(AVG(hd) AS DEC(12,2)) AS avg_hd FROM pc;
```

Функции CEILING и FLOOR

Функция CEILING

Функция CEILING возвращает наименьшее целое число, которое больше или равно числовому выражению, являющемуся аргументом функции.

Следующий запрос

```
1. SELECT 6.28 val, CEILING(6.28) pos_val, CEILING(-6.28)
   neg_val;
```

даст такие результаты:

val	pos_val	neg_val
6.28	7	-6

Возвращаемый функцией результат имеет тот же тип, что и аргумент функции.

Однако рассмотренный пример вроде бы говорит об обратном. Более того, даже выполнив явное преобразование типа для чисел с фиксированной и плавающей точкой, мы получим в результате целое число за исключением константы типа MONEY (в Management Studio):

```
1. SELECT CEILING(CAST(6.28 AS DEC(6,2))) ex_num,
```

```
2.CEILING(CAST(6.28 AS FLOAT)) apr_num, CEILING($6.28)
  money_num;
```

ex_num	apr_num	money_num
7	7	7,00

При использовании других клиентских программ/драйверов вы вполне можете получить другое визуальное представление данных. Выполните, например, последний запрос непосредственно в учебнике. Что у вас получилось?

Можно сказать, что формат отображения является лишь "косвенной уликой" относительно вердикта о типе данных результата. Более надежным критерием является объём, который значение занимает в памяти.

```
1.SELECT DATALENGTH(6.28) val, DATALENGTH(CEILING(6.28))
  num_val,
2.DATALENGTH(CAST(CEILING(6.28) AS INT)) int_val;
```

val	num_val	int_val
5	5	4

Т.е. результат применения функции CEILING занимает в памяти столько же места, что и исходное значение, в то время как результат, явно преобразованный к целому типу, - 4 байта.

Функция FLOOR

Функция FLOOR, напротив, возвращает наибольшее целое число, которое меньше или равно числовому выражению, являющемуся аргументом функции.

```
1.SELECT 6.28 val, FLOOR(6.28) pos_val, FLOOR(-6.28)
  neg_val;
```

Функции LOG и EXP

Функция **LOG**(x) возвращает натуральный логарифм выражения *x*. Результат имеет тип float.

Начиная с версии SQL Server 2012, эта функция приобрела необязательный аргумент, который задает основание логарифма.

val	pos_val	neg_val
6.28	6	-7

```
1.SELECT LOG(2) ln, LOG(2, 10) log10;
```

ln	log10
0.693147180559945	0.301029995663981

Этот запрос возвращает натуральный и десятичный логарифмы числа 2.

Имеется также унаследованная функция **LOG10**(x), возвращающая десятичный логарифм выражения *x*. Она изначально являлась избыточной, ввиду известной формулы преобразования логарифмов:

$$LOG_b a = LOG_c a / LOG_c b$$

Соответственно три следующих выражения дадут один и тот же результат (0.301029995663981):

```
1.SELECT LOG(2,10) log_1, LOG10(2) log_2, LOG(2)/LOG(10)
  log_3;
```

Функция **EXP**(x) - экспонента - возвращает число *e* в степени *x*. Тип возвращаемого значения FLOAT.

Эта функция является обратной к функции LOG:

```
1.SELECT EXP(LOG(2)) a, LOG(EXP(2)) b;
```

a	b
2	2

Еще одно полезное свойство преобразования логарифмов - а именно, логарифм произведения равен сумме логарифмов сомножителей - позволит нам вычислить произведение значений в столбце таблицы, т.е.

```
1.log(a*b*c) = log(a) + log(b) + log(c).
```

В справедливости данного свойства мы можем легко убедиться на примере:

```
1.SELECT LOG(2*5*7) log_prod,  LOG(2) + LOG(5) + LOG(7)
  sum_log;
```

log_prod	sum_log
4,24849524204936	4,24849524204936

Среди агрегатных функций SQL нет функции произведения значений. Тем не менее, используя вышеупомянутое свойство логарифмов и элементарные преобразования, мы можем свести эту задачу к вычислению суммы. Действительно,

```
1.a*b*c = exp(log(a*b*c)) = exp(log(a) + log(b) + log(c)).
```

Вычислить факториал числа, равного количеству строк в таблице Laptop.

Решение

```
1.SELECT EXP(SUM(LOG(rn))) FROM(
2.SELECT ROW_NUMBER() OVER(ORDER BY code) rn FROM laptop
3.) X;
```

Здесь во внутреннем запросе мы пронумеровали записи таблицы Laptop с помощью ранжирующей функции ROW_NUMBER. (Заметим, что порядок, в котором нумеруются строки, в данном случае значения не имеет). Затем просто перемножили эти номера.

Функции POWER и SQRT

Функция SQL Server **POWER** (x, y) возводит x в степень y.

x является выражением типа **FLOAT**, или типа, неявно приводимого к FLOAT.

y - выражение числового типа.

Возвращаемое значение имеет тип выражения x.

Функция **SQRT** (x) вычисляет корень квадратный из x, при этом x является выражением типа FLOAT, или неявно приводимого к нему. Результат имеет тип FLOAT.

Функция SQRT является обратной к функции POWER(x,2), т.е. SQRT(POWER(x,2)) должна возвращать x.

Проверим это

```
1.SELECT 3 x, power(3,2) y, sqrt(power(3,2)) sqrt_y;
```

x	y	sqrt_y
3	9	3

Правильно. Однако

```
1.SELECT 3.1 x, power(3.1,2) y, sqrt(power(3.1,2)) sqrt_y;
```

даст

x	y	sqrt_y
3.1	9.6	3,09838667696593

Этот неожиданный результат, вероятно, связан с потерей точности при неявном преобразовании результата функции POWER (который соответствует типу аргумента, т.е. numeric) к типу данных FLOAT.

Действительно,

```
1.SELECT SQL_VARIANT_PROPERTY(3.1,'BASETYPE') basetype
```

basetype
numeric

Если применить эквивалентное преобразование, которое сохраняет тип NUMERIC для возвращаемого результата,

```
1.SELECT 3.1 x, power(3.1,2) y, power(power(3.1,2),0.5)
  sqrt_y;
```

то получим ожидаемый результат

x	y	sqrt_y
3.1	9.6	3.1

Аналогичный результат мы получим, применив преобразование типа данных аргумента функции POWER к FLOAT в примере с использованием SQRT. В этом случае функция POWER вернет значение типа FLOAT, и преобразование не потребуется. Действительно,

```
1.SELECT 3.1 x, power(3.1,2) y, sqrt(power(CAST(3.1 AS
  float),2)) sqrt_y;
```

x	y	sqrt_y

3.1	9.6	3,1

Если же изменить порядок применения функций, то никаких "чудес" не возникает:

```
1.SELECT power(sqrt(9.6),2) power_;
```

power_
9,6

В этом примере функция SQRT возвращает результат типа FLOAT, что не требует преобразования.

Типичные проблемы

В этой главе мы рассмотрим несколько типичных проблем, часто возникающих на практике и вызывающих определенные трудности. Следуя нашей традиции, решаться эти задачи будут одним запросом, то есть без создания временных таблиц и использования курсоров.

Кроме того, здесь повторно рассмотрены некоторые аспекты языка для более глубокого понимания.

Генерация числовой последовательности

Иногда возникает необходимость получить в запросе числовую последовательность. Это может быть самоцелью или же промежуточным

результатом для получения, скажем, последовательности дат. Пусть, например, требуется получить последовательность целых чисел от 1 до 100 с шагом 1. Можно, конечно, строить такую последовательность в «лоб», то есть:

```
1.SELECT 1 AS num
2.UNION ALL
3.SELECT 2
4....
5.UNION ALL
6.SELECT 100;
```

А если потребуется 1000 чисел или больше? Помимо затрат времени на написание такого количества операторов, мы будем неэффективно использовать сетевой трафик, передавая на выполнение серверу запросы такого размера.

Помочь нам уменьшить размер запроса может декартово произведение (**CROSS JOIN**), которое редко когда используется непосредственно, но часто является промежуточным результатом в различных алгоритмах получения осмысленных данных. Существенной особенностью декартового произведения является то, что мощность результата (количество строк) равно произведению мощностей участвующих в декартовом произведении таблиц. Например, если нам нужно получить последовательность 100 чисел, мы можем использовать декартово произведение таблиц, каждая из которых содержит всего по 10 записей. Итак:

```
1.SELECT *
2.FROM (SELECT 1 a UNION ALL SELECT 2 UNION ALL SELECT 3
  UNION ALL SELECT 4
3. UNION ALL SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT
  7
4. UNION ALL SELECT 8 UNION ALL SELECT 9 UNION ALL SELECT
  10
5. ) x CROSS JOIN
6. (SELECT 1 b UNION ALL SELECT 2 UNION ALL SELECT 3 UNION
  ALL SELECT 4
7. UNION ALL SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT
  7
8. UNION ALL SELECT 8 UNION ALL SELECT 9 UNION ALL SELECT
  10
9. ) y;
```

Результатом здесь является двухстолбцовая таблица, содержащая 100 строк. При этом каждое значение из первого подзапроса (числа от 1 до 10) сочетается с каждым значением из второго (аналогичного) подзапроса:

1	1
1	2
…	…
1	10
2	1
2	2
…	…
2	10
…	…

Теперь осталось только вычислить сами значения. Будем считать, что число в первом столбце представляет собой десятки –1, а второй — единицы. Тогда вместо *SELECT * FROM*… в нашем запросе напишем:

```
1.SELECT 10*(a-1)+b
2.FROM …
```

что и даст требуемый результат.

А почему бы не взять три таблицы (подзапроса)? Чем больше размер генерируемой последовательности, тем больше таблиц следует взять, чтобы получить более короткий запрос. Аналогично рассуждая и, исходя из того, что 5 * 5 * 5 = 125, получим:

```
1.SELECT 5*5*(a-1)+5*(b-1) + c AS num
```

```
2.FROM (SELECT 1 a UNION ALL SELECT 2 UNION ALL SELECT 3
3. UNION ALL SELECT 4 UNION ALL SELECT 5
4. ) x CROSS JOIN
5. (SELECT 1 b UNION ALL SELECT 2 UNION ALL SELECT 3
6. UNION ALL SELECT 4 UNION ALL SELECT 5
7. ) y CROSS JOIN
8. (SELECT 1 c UNION ALL SELECT 2 UNION ALL SELECT 3
9. UNION ALL SELECT 4 UNION ALL SELECT 5
10.       ) z
11.      WHERE 5*5*(a-1)+5*(b-1) + c <= 100
12.      ORDER BY 1;
```

Условие

```
1.WHERE 5*5*(a-1)+5*(b-1) + c <= 100
```

использовано для того, чтобы ограничить последовательность значением 100, а не 125.

Рассмотрим «практический» пример. Пусть требуется получить 100 последовательных незанятых номеров моделей, идущих за последним номером модели в таблице Product. Идея такова: находим максимальный номер модели и далее, используя генерацию последовательности, 100 последующих значений с шагом 1.

```
1.SELECT (SELECT MAX(model)
2. FROM Product
3. ) + 5*5*(a-1)+5*(b-1) + c AS num
4.FROM (SELECT 1 a UNION ALL SELECT 2 UNION ALL SELECT 3
5. UNION ALL SELECT 4 UNION ALL SELECT 5
6. ) x CROSS JOIN
7. (SELECT 1 b UNION ALL SELECT 2 UNION ALL SELECT 3
8. UNION ALL SELECT 4 UNION ALL SELECT 5
9. ) y CROSS JOIN
10.       (SELECT 1 c UNION ALL SELECT 2 UNION ALL SELECT 3
11.       UNION ALL SELECT 4 UNION ALL SELECT 5
12.       ) z
13.      WHERE 5*5*(a-1)+5*(b-1) + c <= 100
14.      ORDER BY 1;
```

Результат выполнения этого запроса мы не будем здесь приводить из экономии места. Проверьте самостоятельно, щелкнув по кнопке "Выполнить".

Если ваш сервер поддерживает CTE, то получение числовой последовательности существенно упрощается. Вы можете использовать Консоль учебника, чтобы решить рассмотренную здесь задачу этим способом. За примерами вы можете обратиться к главе, посвященной рекурсивным CTE, и где рассматривается несколько числовых последовательностей.

Вероятно, ввиду часто возникающей потребности в числовых последовательностях, в PostgreSQL имеется функция, которая возвращает такую последовательность:

```
1.generate_series(start, stop [, step])
```

Здесь
start - начальное значение последовательности,
stop - конечное значение последовательности,
step - шаг последовательности (по умолчанию равен 1).

Применение данной функции проще показать на примерах. Начнем с задачи, которая рассматривалась на предыдущей странице:

> *Получить 100 последовательных незанятых номеров моделей, идущих за последним номером модели в таблице Product.*

Решение для PostgreSQL можно записать очень лаконично:

```
1.SELECT CAST(MAX(model) AS INT) + generate_series(1,100)
  AS num FROM Product;
```

Преобразование типа здесь необходимо, поскольку номер модели (model) имеет тип VARCHAR.

Следующий пример - это вывод алфавита, который мы получали с помощью SQL-рекурсии. Применим тот же алгоритм, а именно, к коду первой буквы будем последовательно добавлять единицы, после чего преобразуем полученные коды к символам:

```
1.SELECT CHR(ASCII('A') + generate_series(0,25)) AS letter
  ORDER BY 1;
```

Наконец, рассмотрим довольно часто возникающую необходимость получения последовательности дат. В связи с этим отметим, что третий параметр (step) может быть не только типа INT, но и типа INTERVAL.

Последнее позволит нам непосредственно работать с последовательностями дат, минуя преобразование последовательности чисел к последовательности дат. Итак,

Вывести последовательность дат между датами первого и последнего полета пассажира с id_psg=5.

```
1. SELECT generate_series(MIN(date), MAX(date), '1 day')
2. FROM pass_in_trip WHERE id_psg = 5;
```

Поскольку пока на внутренних страницах учебника есть возможность выполнять скрипты только под SQL Server, вы можете для выполнения приведенных здесь запросов воспользоваться консолью, выбрав PostgreSQL в списке поддерживаемых СУБД.

Нумерация

Обычно необходимость нумерации строк возникает при формировании отчетов. В этом случае нумерацию строк, возвращаемых запросом, обычно реализуют на клиенте. Например, не составляет особого труда перенумеровать строки отчета, подготовленного в MS Access. Однако иногда это необходимо сделать в самом запросе. Этот случай мы сейчас и рассмотрим.

Нумерация строк в соответствии с порядком, заданном значениями первичного ключа

Естественно, нумероваться строки должны в соответствии с некоторым порядком. Пусть этот порядок задается столбцом первичного ключа, то есть в порядке возрастания (или убывания) значений в этом единственном столбце.

Для определенности предположим, что нам нужно перенумеровать модели в таблице Product, где номер модели как раз является первичным ключом. Существенным здесь является то, значения первичного ключа не содержат дубликатов и **NULL**-значений, в результате чего имеется принципиальная возможность установить однозначное соответствие между номером модели и номером строки в заданном порядке сортировки моделей.

Рассмотрим сначала следующий запрос:

```
1. SELECT P1.model, P2.model
2. FROM Product P1 JOIN
3.  Product P2 ON P1.model <= P2.model;
```

Здесь выполняется соединение двух идентичных таблиц по неравенству P1.model <= P2.model, в результате чего каждая модель из второй таблицы (P2.model) будет соединяться только с теми моделями из первой таблицы (P1.model), номера которых меньше или равны номеру этой модели. В результате получим, например, что модель с минимальным номером (1121) будет присутствовать во втором столбце результирующего набора только один раз, так как она меньше или равна только самой себе. На другом конце будет находиться модель с максимальным номером, так как любая модель будет меньше или равна ей. Следовательно, модель с максимальным номером будет сочетаться с каждой моделью, и число таких сочетаний будет равно общему числу моделей в таблице Product.

Из сказанного выше ясно, что это количество раз, которое каждая из моделей встречается во втором столбце результирующего набора как раз и будет порядковым номером модели при сортировке моделей по возрастанию.

Таким образом, чтобы решить нашу задачу нумерации достаточно пересчитать модели в правом столбце, что нетрудно сделать при помощи группировки и использования агрегатной функции **COUNT**:

Решение 8.2.1

```
1. SELECT COUNT(*) no, P2.model
```

```
2.FROM Product P1 JOIN
3. Product P2 ON P1.model <= P2.model
4.GROUP BY P2.model;
```

Не будем экономить место и представим результат выполнения этого запроса:

no	model
1	1121
2	1232
3	1233
4	1260
5	1276
6	1288
7	1298
8	1321
9	1401
10	1408
11	1433
12	1434
13	1750
14	1752
15	2112

Для нумерации в обратном порядке достаточно поменять знак неравенства на противоположный.

Если ваша СУБД поддерживает ранжирующие функции, то пронумеровать строки можно совсем просто:

```
1. SELECT ROW_NUMBER() OVER(ORDER BY model) no, model
2. FROM Product;
```

Несколько усложним задачу, и попытаемся пронумеровать модели каждого производителя отдельно. Воспользуемся предыдущим решением, и внесем в него следующие изменения:

1. Добавим в условие соединения равенство производителей, чтобы выделить модели каждого производителя в отдельную группу.

```
1. SELECT COUNT(*) no, P2.model
2.     FROM Product P1 JOIN
3.      Product P2 ON P1.maker =P2.maker AND P1.model <= P2.model
4.     GROUP BY P2.model
5.     ORDER BY P2.model;
```

В принципе, это и все. Правда, результат не отличается наглядностью.

2. Добавим в вывод производителя, при этом все равно из какой таблицы мы его возьмем в силу равенства. Однако тогда необходимо добавить производителя в столбцы группировки (MySQL не в счет):

```
1. SELECT COUNT(*) no, P1.maker, P2.model
2.     FROM Product P1 JOIN
```

```
3.     Product  P2  ON  P1.maker  =P2.maker  AND  P1.model  <=
   P2.model
4.     GROUP BY P1.maker, P2.model
5.     ORDER BY P2.model;
```

3. Ну и, наконец, добавим сортировку для наглядности результата. Столбец *maker* должен быть первым столбцом сортировки, чтобы каждая группа выводилась отдельно.

```
1. SELECT COUNT(*) no, P1.maker, P2.model
2.     FROM Product P1 JOIN
3.     Product  P2  ON  P1.maker  =P2.maker  AND  P1.model  <=
   P2.model
4.     GROUP BY P1.maker, P2.model
5.     ORDER BY P1.maker, P2.model;
```

no	maker	model
1	A	1232
2	A	1233
3	A	1276
4	A	1298
5	A	1401
6	A	1408
7	A	1752
1	B	1121
2	B	1750
1	C	1321
1	D	1288

2	D	1433
1	E	1260
2	E	1434
3	E	2112
4	E	2113

Я надеюсь, что выполнить раздельную нумерацию моделей по типам продукции вам теперь не составит труда. Сделайте это в качестве самостоятельного задания.

Ранжирующие функции, естественно, упрощают запрос:

```
1. SELECT ROW_NUMBER() OVER(PARTITION BY maker ORDER BY
   model) no, maker, model
2.     FROM Product
3.     ORDER BY maker, model;
```

Нумерация строк при наличии дубликатов в результирующем столбце

Согласно реляционной теории в таблице не может быть одинаковых строк. И хотя реализации допускают построение таблиц, не имеющих первичного ключа, и, как следствие, допускающих наличие одинаковых строк, следует, на наш взгляд, отнести эту ситуацию к ошибкам в проектировании. Кроме того, таблица, не имеющая первичного ключа или уникального индекса, не является обновляемой. Последнее заключение вполне естественно,

так как система не имеет информации о том, какой из дубликатов предпочесть.

Поэтому, говоря о дубликатах, мы имеем в виду дубликаты в результирующем наборе, появление которых может быть обусловлено тем, что первичный ключ весь или частично (в случае составного ключа) отсутствует в результирующем наборе.

Чтобы пояснить сказанное, рассмотрим следующий запрос

```
1.SELECT id_psg
2.FROM pass_in_trip;
```

который вернет номера пассажиров, совершавших полеты, зафиксированные в базе данных. Поскольку один и тот же пассажир может совершить несколько рейсов, мы получаем здесь дубликаты. Однако ни один пассажир не может в один и тот же день более одного раза полететь одним и тем же рейсом, что регламентируется соответствующим первичным ключом — {trip_no, date, id_psg}.

Итак, нам нужно перенумеровать пассажиров, которые могут повторяться. Зададимся для начала порядком, в котором их нужно перенумеровать. Пусть этот порядок соответствует сортировке по трем полям — дате полета, идентификатору пассажира и номеру рейса (по возрастанию).

Чтобы свести задачу к ранее рассмотренной (а это возможно, так как три перечисленных поля представляют собой первичный ключ), сконструируем столбец, который объединял бы информацию из перечисленных полей. Поскольку поля имеют разные типы данных, приведем их к единому символьному представлению и выполним конкатенацию.

При этом нам нужно определиться с количеством символов. Поскольку в представлении даты вылета отсутствует время, ограничимся 11 символами. Номер рейса везде представлен четырехсимвольным числом. Остается идентификатор пассажира. В соответствии с имеющейся базой данных ограничимся двумя символами, что не принижает общности подхода.

Однако для правильности сортировки нужно «односимвольных» пассажиров записывать с лидирующим нулем — 01, 09 и т. д. Иначе пассажир с номером 10 будет предшествовать, скажем, пассажиру с идентификационным номером 2. Выполним соответствующие преобразования:

Преобразования 8.2.2

```
1.Дата: CAST(date AS CHAR(11))
2.Номер рейса: CAST(trip_no AS CHAR(4))
3.Идентификатор пассажира: RIGHT('00'+CAST(id_psg AS
  VARCHAR(2)),2).
```

В последнем преобразовании идентификатора пассажира мы использовали нестандартную функцию **RIGHT** (SQL Server), которая извлекает из строки указанное количество символов справа. Можно было бы применить функцию **SUBSTRING**, однако так получается короче и, кроме того, наверняка в других коммерческих продуктах имеются аналогичные «расширения» стандарта. Соединяя эти выражения в указанном порядке, мы получим уникальный столбец, который и будет служить для нумерации пассажиров в соответствии с возрастанием (убыванием) значений в этом столбце. Используя этот столбец, мы можем воспользоваться решением 8.2.1. Окончательно получим:

```
1.SELECT COUNT(*) num, P2.id_psg
2.FROM (SELECT *, CAST(date AS CHAR(11)) +
3.                RIGHT('00' + CAST(id_psg AS VARCHAR(2)),
  2)+
4.                CAST(trip_no AS CHAR(4)) dit
5.       FROM Pass_in_trip
6.      ) P1 JOIN
7.      (SELECT *, CAST(date AS CHAR(11)) +
8.                 RIGHT('00' + CAST(id_psg AS VARCHAR(2)),
  2)+
9.                 CAST(trip_no AS CHAR(4)) dit
10.            FROM pass_in_trip
11.            ) P2 ON P1.dit <= P2.dit
12.      GROUP BY P2.dit, P2.id_psg
13.      ORDER BY 1;
```

Для нумерации в другом порядке достаточно конкатенировать преобразования 8.2.2 в другой последовательности. Например, чтобы пронумеровать пассажиров в порядке их идентификационных номеров, первым слагаемым должно быть:

```
1.RIGHT('00' + CAST(id_psg AS VARCHAR(2)), 2).
```

В этом примере еще более наглядно, чем для решения 8.2.1, проявляются достоинства ранжирующих функций. Вот как просто и понятно можно переписать решение нашей задачи с их использованием:

```
1.SELECT ROW_NUMBER() OVER(ORDER BY date, id_psg, trip_no)
  num, id_psg
2.FROM Pass_in_trip
3.ORDER BY num;
```

Об использовании оператора CASE

В главе 5 (пункт 5.10) описан синтаксис этого оператора и приведены примеры его использования. Однако эти примеры демонстрируют применение **CASE** только в предложении SELECT. Действительно, это место наиболее частого использования оператора, однако не единственное. Мы можем применять CASE практически везде, где ожидается скалярное значение. Здесь мы приведем несколько примеров его применения в других предложениях оператора SELECT.

Предложение WHERE

Предложение **WHERE** ограничивает выходной набор теми строками, которые удовлетворяют предикату в этом предложении. Предположим, что у нас имеется следующее соответствие между объемом

памяти ПК и типом используемой операционной системы (естественно, условно):

RAM < 64	W95
RAM >=64 и < 128	W98
RAM >= 128	W2k

Если мы захотим отобрать компьютеры по типу ОС (заметим, что такого поля нет в таблице PC), то можно написать следующий оператор:

```
1.SELECT *
2.FROM PC
3.WHERE CASE
4. WHEN ram < 64
5. THEN 'W95'
6. WHEN ram < 128
7. THEN 'W98'
8. ELSE 'W2k'
9. END ='W98';
```

Здесь мы выбираем модели, подходящие для операционной системы W98. Ниже приведен результат выполнения этого запроса.

code	model	speed	ram	hd	cd	price
1	1232	500	64	5	12x	600.0
3	1233	500	64	5	12x	600.0
8	1232	450	64	8	24x	350.0

Это может оказаться полезнее, чем кажется на первый взгляд, если иметь в виду конфиденциальность информации. Например, на клиенте можно формировать запросы, которые будут оперировать такими категориями, как высоко, средне и низко оплачиваемый специалист. То есть сами критерии

(оклады) будут спрятаны, скажем, в хранимой процедуре, а через параметр будет передаваться что-то типа символьной строки «средний».

Предложение GROUP BY

Пусть теперь мы хотим получить количество компьютеров, подходящих по RAM к каждому типу операционных систем. Тогда мы можем написать следующий запрос:

```
1.SELECT
2.       CASE
3.            WHEN ram < 64
4.            THEN 'W95'
5.            WHEN ram < 128
6.            THEN 'W98'
7.            ELSE 'W2k'
8.        END Type,
9.        COUNT(*) Qty
10.      FROM PC
11.      GROUP BY
12.              CASE
13.                   WHEN ram < 64
14.                   THEN 'W95'
15.                   WHEN ram < 128
16.                   THEN 'W98'
17.                   ELSE 'W2k'
18.              END;
```

В результате выполнения запроса получим:

Type	Qty
W2k	5

W95	3
W98	3

Здесь мы дублируем оператор **CASE** в предложении **SELECT**, чтобы получить столбец с наименованием операционной системы.

Использование оператора CASE в предложении GROUP BY позволяет в рамках одного запроса выполнять группировку по разному числу столбцов. Рассмотрим, например, следующую задачу.

Для каждой комбинации скорости процессора и объема жесткого диска определить среднюю цену ПК. Для ПК со скоростью процессора менее 600 выполнять группировку только по скорости процессора.

Сначала для сравнения приведем результаты группировки по одному (*speed*) и двум столбцам (*speed*, *hd*) соответственно.

```
1. SELECT speed,MAX(hd) max_hd, AVG(price) avg_price  FROM
   pc
2. GROUP BY speed;
```

speed	max_hd	avg_price
450	10	350.00
500	10	487.50
600	14	850.00
750	20	900.00
800	20	970.00
900	40	980.00

```
1. SELECT speed, hd max_hd, AVG(price) avg_price  FROM pc
2. GROUP BY speed, hd
```

```
3. ORDER BY speed;
```

speed	max_hd	avg_price
450	8	350.00
450	10	350.00
500	5	600.00
500	10	375.00
600	8	850.00
600	14	850.00
750	14	850.00
750	20	950.00
800	20	970.00
900	40	980.00

Поскольку в каждой группе значения *hd* совпадают, можно *max(hd)* и *hd* использовать равноправно.

Теперь дадим решение нашей задачи:

```
1. SELECT speed, MAX(hd) max_hd, AVG(price) avg_price FROM
   pc
2. GROUP BY speed, CASE WHEN speed >= 600 THEN hd ELSE speed
   END
3. ORDER BY speed;
```

speed	max_hd	avg_price
450	10	350.00

500	10	487.50
600	8	850.00
600	14	850.00
750	14	850.00
750	20	950.00
800	20	970.00
900	40	980.00

Когда *speed >= 600*, выполняется группировка по столбцам *speed, hd*. В противном случае группировка выглядит так:

```
1. GROUP BY speed, speed
```

что эквивалентно группировке одному столбцу. Кстати, оператор CASE в последнем решении можно было написать без ELSE:

```
1. SELECT speed, MAX(hd) max_hd, AVG(price) avg_price  FROM
   pc
2. GROUP BY speed, CASE WHEN speed >= 600 THEN hd END
3. ORDER BY speed;
```

Хотя явная группировка - *GROUP BY speed, NULL* - будет вызывать ошибку.

Предложение HAVING

Пусть требуется найти красные квадраты, т.е. квадраты, окрашенные только баллонами с красной

краской и суммарным количеством краски, равным 255. Речь идёт о базе данных "Окраска".

Эту задачу можно решить разными способами. Здесь мы приведём решение с использованием оператора **CASE** в предложении **HAVING**.

Идея состоит в следующем. Выполняя группировку по идентификатору квадрата, будем суммировать объем краски. При этом краску красного цвета будем добавлять со знаком "+", а остальную - со знаком "-". Поскольку краски каждого цвета на квадрате не может быть больше 255 единиц, то результат, равный в точности 255 говорит о том, что вся эта краска была красного цвета, и никакой другой использовано не было.

А вот и само решение:

```
1. SELECT b_q_id
2. FROM utb JOIN utv ON v_id=b_v_id
3. GROUP BY b_q_id
4. HAVING SUM(CASE WHEN v_color='R' THEN b_vol ELSE -b_vol
   END) = 255;
```

Предложение ORDER BY

Использование оператора **CASE** в предложении **ORDER BY** позволяет выполнить более сложную сортировку, чем та, которая допускается при использовании сортировки по набору столбцов. При сортировке по столбцу ram можно выполнить ее по возрастанию или по убыванию. Если же мы хотим вывести сначала средние модели, то есть те, которые отвечают W98, потом высшие (W2k), а уже потом низшие (W95), то можно поступить так:

```
1. SELECT *
```

```
2.FROM PC
3.ORDER BY CASE
4.  WHEN ram <= 32
5.  THEN '3-W95'
6.  WHEN ram <= 64
7.  THEN '1-W98'
8.  ELSE '2-W2k'
9.  END;
```

Цифры перед названием ОС проставлены в соответствии с желательным порядком сортировки. В противном случае, упорядочение текстовых значений по возрастанию будет следующим: W2k, W95, W98. Вот результат вышеприведенного запроса (заголовок столбца сортировки выделен прописными буквами):

code	model	speed	RAM	hd	cd	price
3	1233	500	64	5	12x	600
1	1232	500	64	5	12x	600
8	1232	450	64	8	24x	350
2	1121	750	128	14	40x	850
4	1121	600	128	14	40x	850
5	1121	600	128	8	40x	850
6	1233	750	128	20	50x	950
11	1233	900	128	40	40x	980
12	1233	800	128	20	50x	970
7	1232	500	32	10	12x	400
9	1232	450	32	10	24x	350
10	1260	500	32	10	12x	350

Есть еще более интересная возможность сортировки, а именно, сортировать по разным столбцам в зависимости от значения в некотором поле. Пусть, например, в группе W95 мы хотим выполнить сортировку по столбцу speed, в группе W98 — по столбцу hd, в группе W2k — по столбцу price. То есть в каждой группе, характеризуемой ОС описанными выше критериями, нам нужно выполнить сортировку по разным столбцам. Эту, на первый взгляд непростую задачу, решает простой запрос с оператором **CASE** в предложении **ORDER BY**:

```
1.SELECT *
2.FROM PC
3.ORDER BY ram,
4. CASE
5. WHEN ram <= 32
6. THEN speed
7. WHEN ram <= 64
8. THEN hd
9. ELSE price
10.       END;
```

Вот результат этого запроса:

code	model	speed	ram	hd	cd	price
9	1232	450	32	10	24x	350
10	1260	500	32	10	12x	350
7	1232	500	32	10	12x	400
1	1232	500	64	5	12x	600
3	1233	500	64	5	12x	600
8	1232	450	64	8	24x	350

2	1121	750	128	14	40x	850
4	1121	600	128	14	40x	850
5	1121	600	128	8	40x	850
6	1233	750	128	20	50x	950
12	1233	800	128	20	50x	970
11	1233	900	128	40	40x	980

Еще о NULL-значениях

Смысл **NULL**-значения — это отсутствие информации или неприменимость данного атрибута в данном кортеже.

Можно спросить: «Зачем иметь атрибут, если его значение неприменимо?». Ответ лежит в области моделирования предметной области. Рассмотрим, например, схему базы данных «Компьютеры». Она представляет собой реляционную модель связи «тип-супертип». Сущностями предметной области здесь являются модели компьютерной продукции (супертип), при этом каждый тип продукции (ПК, портативный компьютер или принтер) отображается в отдельную таблицу со связями «многие к одному» с таблицей Product.

Такая модель обеспечивает высокую степень нормализации (3НФ). Однако это не единственный способ. Можно было бы хранить всю информацию в одной таблице, которая содержала бы как общие для всех моделей атрибуты (например, цена — price), так и атрибуты, которые имеют смысл только для моделей определенного типа (например, цвет — color — для характеристики принтеров). Для такой схемы **NULL**-

значение является вполне оправданным именно в смысле неприменимости характеристики, то есть **NULL** в столбце color, будет говорить нам о том, что эта характеристика не имеет отношения, скажем, к моделям ПК.

Обратимся теперь ко второй ипостаси **NULL**-значений — отсутствию информации. Если мы решим отказаться от использования **NULL**-значений, то должны предложить альтернативу. Естественным путем является применение значения по умолчанию, которое будет подставлено в соответствующий столбец при отсутствии информации. Следует заметить, что таких значений по умолчанию должно быть, по меньшей мере, столько, сколько различных типов данных поддерживается СУБД (целые, строки, дата-время, …).

Рассмотрим, например, таблицу Laptop и поле price (цена). Пусть предметная область такова, что на момент ввода информации о моделях портативных компьютеров их цена не всегда известна. При выборе значения по умолчанию мы должны ограничиться только значениями, допустимыми для поля price. Тип данных для столбца (money) заставляет нас выбирать только из числовых значений, совместимых с данным типом и проверочными ограничениями (ограничение типа **CHECK**), наложенными на допустимые значения для этого столбца. Любое положительное значение в качестве значения по умолчанию будет вызывать путаницу, так как невозможно будет отличить «истинное» значение цены от заменителя отсутствующей цены. Поэтому следует выбрать нуль или любое отрицательное значение. А теперь поговорим о том, чем плоха такая замена.

Для примера рассмотрим информацию о моделях 1298, имеющихся в таблице Laptop. Чтобы познакомиться с данными, выполним запрос:

```
1. SELECT *
2. FROM Laptop
3. WHERE model = 1298;
```

Вот результаты выполнения этого запроса:

code	model	speed	ram	hd	price	screen
1	1298	350	32	4	700.0	11
4	1298	600	64	10	1050.0	15
6	1298	450	64	10	950.0	12

Рассмотрим задачу получения средней цены модели 1298. Пока все цены известны решение этой задачи не вызывает никаких сомнений:

```
1.SELECT model, AVG(price) avg_price
2.FROM Laptop
3.WHERE model = 1298
4.GROUP BY model;
```

model	avg_price
1298	900.0

Пусть теперь цена модели с кодом 4 неизвестна. Если, как было решено ранее, мы будем заменять неизвестное значение, скажем, нулем (*UPDATE Laptop SET price = 0 WHERE code=4*), то получим заведомо неверное среднее значение цены — 550.0

Если же мы будем использовать **NULL**-значение, то результат будет вполне правильным — 825.0, так как **NULL**-значения будут игнорироваться при группировке, в результате чего среднее значение будет вычисляться только по моделям с известной ценой (то есть среднее по двум моделям).

Итак, как мы постарались показать, **NULL**-значение является неотъемлемой особенностью реляционной модели, поэтому рекомендуем научиться корректно работать с такими значениями.

Примечание:

Ради объективности отсылаем вас к аргументированной критике Дейта относительно использования NULL-значений [1]. Коддом было предложено [6] два разных типа NULL-значений, соответствующих как раз тем двум аспектам их применения, о которых шла речь выше.

Сравнение строк, содержащих NULL-значения

Как известно, предикаты простого сравнения с NULL-значениями дают истинностное значение UNKNOWN, т.е. ни TRUE и ни FALSE, что означает "неизвестно". Поэтому не стоит удивляться, что в одних случаях, при сравнении между собой NULL-значений, они считаются равными друг другу, а других случаях - нет. Поясним сказанное на примерах.

Рассмотрим соединение двух одинаковых строк, содержащих NULL-значения, по равенству всех столбцов.

```
1.WITH A AS (
2.SELECT 'a' a, NULL b
3.)
4., B AS (
5.SELECT 'a' a, NULL b
6.)
7.SELECT * FROM A JOIN B ON A.a=B.a AND A.b=B.b;
```

В этом случае соединяться будут только те строки, для которых предикат соединения есть TRUE. Поскольку предикат соединения в примере оценивается как UNKNOWN, в результате мы не получим ни одной строки.

Однако пересечение запросов (так же, как объединение и разность) считает эти строки идентичными:

```
1.WITH A AS (
2.SELECT 'a' a, NULL b
3.)
4., B AS (
5.SELECT 'a' a, NULL b
6.)
7.SELECT * FROM A
8.INTERSECT
```

```
9. SELECT * FROM B;
```

a	b
a	NULL

Можно сделать вывод о том, что при горизонтальных операциях NULL-значения не считаются равными (и неравными, впрочем, тоже), а при вертикальных трактуются как равные. В частности, при группировке по столбцу, содержащему NULL-значения, последние образуют одну группу.

В заключение рассмотрим несколько решений задачи определения количества принтеров с неизвестной ценой. Таблица PrinterN отличается от таблицы Printer тем, что для пары моделей цена установлена в NULL.

(1) От общего числа строк отнимем число строк с известной ценой.

```
1. SELECT COUNT(*) - COUNT(price) qty FROM printerN;
```

(2) Использование предиката IS NULL для подсчёта строк, для которых цена неизвестна.

```
1. SELECT COUNT(*) FROM printerN WHERE price IS NULL;
```

(3) Группируем по цене и отбираем группу с неизвестной ценой, используя HAVING.

```
1. SELECT COUNT(*) FROM printerN GROUP BY price HAVING
   price IS NULL;
```

Трехзначная логика и предложение Where

Рассмотрим следующий пример.

Пусть требуется определить корабли с неизвестным годом спуска на воду (база данных "Корабли").

Если мы напишем

Решение 8.5.1

```
1. SELECT *
2. FROM Ships
3. WHERE launched = NULL;
```

то, как бы ни казалось это странным, мы не получим ни одной записи, даже если такие корабли имеются в таблице Ships (напомним, что столбец launched допускает **NULL**-значения) Поскольку в доступной базе данных нет кораблей с неизвестным годом спуска на воду, давайте их создадим, чтобы вы могли проверить справедливость данного утверждения:

```
1. SELECT *
2. FROM (SELECT name, launched,
3.  CASE
4.  WHEN launched < 1940
5.  THEN NULL
6.  ELSE launched
7.  END year
8.  FROM Ships
9.  ) x
10.      WHERE year = NULL;
```

Здесь мы добавили в подзапросе столбец year, который содержит **NULL**, если корабль был спущен на воду до 1940 года.

Итак, почему мы ничего не получили? Здесь следует вспомнить о том, что в SQL (и вообще в реляционной теории) используется трехзначная логика, то есть истинностным значением операции сравнения может быть не только **TRUE** (истина) и **FALSE** (ложь), но и **UNKNOWN** (неизвестно). Это обусловлено существованием **NULL**-значения, сравнение с которым и дает это истинностное значение. Это интуитивно понятно, если помнить, что **NULL**-значение служит для замены неизвестной информации. Если мы спросим: «Является ли годом спуска на воду корабля Бисмарк 1939 год»? Ответом будет: «Не знаю». Так как у нас нет информации в базе данных о годе спуска на воду этого корабля, это «не знаю» и есть **UNKNOWN**.

Что происходит, если в предложении **WHERE** мы используем сравнение с **NULL**-значением явно или неявно (с **NULL**-значением в сравниваемом столбце)? Запись попадает в результирующий набор, если предикат дает истинностное значение TRUE. И все, то есть при значениях **FALSE** или **UNKNOWN** запись не попадает в результат. Именно поэтому мы ничего и не получили в приведенном выше примере, поскольку для всех строк получаем **UNKNOWN**.

Так как же получить список кораблей с неизвестным годом спуска на воду? Для этого в стандарте SQL имеется специальный предикат **IS NULL** (и обратный ему **IS NOT NULL**). Истинностным значением этого предиката не может быть **UNKNOWN**, то есть год либо известен (**FALSE**), либо неизвестен (**TRUE**). Тогда для решения нашей задачи можно написать:

Решение 8.5.2

```
1. SELECT *
2. FROM Ships
3. WHERE launched IS NULL;
```

Это стандарт. А что же реализации? Все сказанное выше справедливо для SQL Server. Однако это не единственная возможность. Видимо, чтобы сделать программирование на SQL более привычным для тех, кто пользуется традиционными языками программирования, можно отключить стандартную трактовку **NULL**-значений (по умолчанию включено) с помощью соответствующей установки параметра **ANSI_NULLS**:

```
1.SET ANSI_NULLS OFF|ON
```

Напишите в Management Studio (или в Query Analyzer для SQL Server 2000 и ранее) следующий код, и вы все поймете:

```
1.SET ANSI_NULLS OFF;
2.SELECT *
3.FROM (SELECT name, launched,
4. CASE
5. WHEN launched < 1940
6. THEN NULL
7. ELSE launched
8. END year
9. FROM Ships
10.       ) x
11.      WHERE year = NULL;
```

Предикат NOT IN

Рассмотрим еще один пример, позаимствованный мной у Селко [7]. Идея его состоит в использовании предиката **NOT IN** (<список значений, включающий **NULL**>).

Опять таки, для того чтобы вы могли проверить справедливость рассуждений на сайте, давайте искусственно добавим **NULL**-значения в результат запроса:

```
1.SELECT name, launched,
2. CASE
3. WHEN launched < 1915
4. THEN NULL
5. ELSE launched
6. END year
7.FROM Ships
8.WHERE launched <= 1915;
```

Мы специально взяли 1915 год, чтобы результирующий набор был невелик. Вот он:

name	launched	year
Hiei	1914	NULL
Kirishima	1915	1915
Kongo	1913	NULL

А теперь напишем запрос, который должен вернуть все корабли, год спуска на воду не находится в наборе значений столбца year:

```
1.SELECT *
2.FROM Ships
3.WHERE launched <= 1916 AND
4. launched NOT IN(SELECT year
5. FROM (SELECT name, launched,
6. CASE WHEN launched < 1915
7. THEN NULL
8. ELSE launched
9. END year
10.       FROM Ships
11.       WHERE launched <= 1915
12.       )  x
13.       );
```

Запрос

```
1.SELECT *
2.FROM Ships
3.WHERE launched <= 1915;
```

дает нам следующий набор кораблей:

name	class	launched
Hiei	Kongo	1914
Kirishima	Kongo	1915
Kongo	Kongo	1913

Казалось бы, мы должны получить корабли Hiei и Kongo, так как год их спуска на воду известен и не равен 1915. Но нет, мы опять получаем пустой результирующий набор.

Оценим значение предиката для первого из этих кораблей — Hiei (для остальных все будет аналогично). Итак,

```
1.1914 NOT IN (1915, NULL)
```

Еще одно **NULL**-значение мы опустили для краткости. Последний предикат можно заменить следующим:

```
1.1914 <> ALL (1915, NULL)
```

что эквивалентно

```
1.1914 <> 1915
2.AND
3.1914 <> NULL
```

Последнее выражение всегда равно **UNKNOWN**, следовательно, предикат можно переписать в виде:

```
1.1914 <> 1915
2.AND
3.UNKNOWN
```

Следовательно, и все выражение будет равно **UNKNOWN**, так как первое сравнение дает **TRUE**. Если бы первое сравнение было ложным (для 1915 года), то результат всего выражения был бы равен **FALSE**.

Поэтому можно сделать вывод, что при наличии **NULL**-значения в наборе предикат **NOT IN** в предложении **WHERE** всегда будет давать пустой набор записей.

В заключение следует сказать, что если вы выполняете горизонтальную фрагментацию некоторой таблицы, используя некоторое пороговое значение столбца, допускающего **NULL**-значения, то объединение фрагментов типа

```
1.SELECT *
2.FROM Ships
3.WHERE launched <= 1915
4.UNION
5.SELECT *
6.FROM Ships
7.WHERE launched > 1915;
```

не гарантирует восстановления исходной таблицы. Для этого потребуется еще один фрагмент, содержащий в столбце launched **NULL**-значения:

```
1.SELECT *
2.FROM Ships
3.WHERE launched IS NULL;
```

Эти «хитрые» внешние соединения

Пусть требуется для каждого класса определить все корабли с известным годом спуска на воду. Когда говорится «для каждого класса», мы уже знаем, что нужно использовать внешнее соединение, например, левое:

Решение 8.6.1

```
1.SELECT Classes.class, name, launched
2.FROM Classes LEFT JOIN
3. Ships ON Classes.class = Ships.class AND
4. launched IS NOT NULL;
```

Тем самым мы соединяем таблицу Classes с таблицей Ships по столбцу class и отбираем корабли с известным годом спуска на воду. Вот что, помимо прочего, мы имеем в результате:

Class	Name	launched
Bismarck	NULL	NULL

Как же так? Мы же указывали в предикате соединения *launched IS NOT NULL*? В словах «в предикате соединения» как раз и кроется ответ на наш вопрос. Вернемся к определению внешнего левого соединения:

Соединяются все строки из левой таблицы с теми строками из правой, для которых значение предиката истинно. Если для какой-либо строки из левой таблицы нет ни одной соответствующей строки из правой таблицы, то значения столбцов правой таблицы получают значение **NULL**.

В таблице Ships нет ни одного корабля класса Bismarck. Потому мы и получили эту строку, так как класс Bismarck есть в таблице Classes. А если бы такой корабль был? Давайте добавим в таблицу Ships два корабля класса Bismarck — один с известным годом спуска на воду, а другой — с неизвестным:

```
1. SELECT *
2. FROM Ships
3. UNION ALL
4. SELECT 'B_1' AS name, 'Bismarck' AS class, 1941 AS
   launched
5. UNION ALL
6. SELECT 'B_2' AS name, 'Bismarck' AS class, NULL AS
   launched;
```

Перепишем решение 8.6.1 с учетом этих новых кораблей:

Решение 8.6.2

```
1. SELECT Classes.class, name, launched
2. FROM Classes LEFT JOIN
3.  (SELECT *
4.  FROM Ships
5.  UNION ALL
6.  SELECT 'B_1' AS name, 'Bismarck' AS class, 1941 AS
   launched
7.  UNION ALL
8.  SELECT 'B_2' AS name, 'Bismarck' AS class, NULL AS
   launched
9.  ) Ships ON Classes.class = Ships.class AND
10.       launched IS NOT NULL;
```

Теперь получаем ожидаемый результат, а именно, в результирующем наборе будет присутствовать только один корабль класса Bismarck:

Class	Name	launched
Bismarck	B_1	1941

Вывод. Если вам нужно ограничить результирующий набор внешнего соединения, используйте предложение WHERE, которое как раз и служит для этой цели:

Решение 8.6.3

```
1. SELECT Classes.class, name, launched
2. FROM Classes LEFT JOIN
3.  Ships ON Classes.class = Ships.class
4. WHERE launched IS NOT NULL;
```

Предикат же соединения определяет лишь то, какие строки из разных таблиц будут конкатенированы в результирующем наборе.

В заключении замечу, что данный пример не является вполне показательным, так как для решения поставленной задачи вполне подошло бы внутреннее соединение (**INNER JOIN**), несмотря на слова «для каждого класса». Однако гибкость языка SQL позволяет решить задачу разными способами, и использование стереотипов вполне оправдано.

Как правило, в приводимых примерах используются эквисоединения, т.е. соединения по равенству значений (=). Это обусловлено тем, что на практике зачастую используется соединение по внешнему ключу. Подобные примеры можно увидеть на предыдущей странице. Однако предикатом соединения может быть любое логическое выражение. Для иллюстрации рассмотрим следующую задачу.

Найти такие поступления в таблице Income_o, каждое из которых превосходит любой из расходов в таблице Outcome_o.

Решение.

```
1. SELECT Income_o.* FROM Outcome_o RIGHT JOIN Income_o ON
   Outcome_o.out >= Income_o.inc
2. WHERE Outcome_o.out IS NULL;
```

В вышеприведенном решении внешнее соединение выполняется по неравенству *Outcome_o.out >= Income_o.inc*, которому отвечают строки из таблицы Income_o, для которых приход не превышает расхода для каких-либо строк в таблице Outcome_o. Кроме того, во внешнем соединении (в данном случае в правом) будут присутствовать и строки из таблицы Income_o, для которых не нашлось ни одной строки в таблице Outcome_o, делающей истинным значение предиката.

Это и есть строки, являющиеся решением нашей задачи. Чтобы их выбрать, используем тот факт, что отсутствующие значения столбцов из соединяемой таблицы (у нас левой) заполняются NULL-значениями. Соответствующий критерий помещаем в предложение WHERE.

Разумеется, эту задачу можно решить и другими способами, например:

max + подзапрос

```
1. SELECT Income_o.* FROM Income_o
2. WHERE  Income_o.inc  >  (SELECT  MAX(Outcome_o.out)  FROM
   Outcome_o);
```

all + подзапрос

```
1. SELECT Income_o.* FROM Income_o
2. WHERE Income_o.inc > ALL(SELECT Outcome_o.out FROM
   Outcome_o);
```

О неявном преобразовании типов в SQL Server

Помимо типов данных в реляционной теории вводится фундаментальное понятие домена, как множества допустимых значений, которое может иметь атрибут. Можно сказать, что домен представляет собой пару {базовый тип данных, предикат}. При этом значение принадлежит домену только в том случае, если оно имеет соответствующий тип и предикат, вычисленный на этом значении, есть ИСТИНА. Атрибуты (столбцы таблицы) определяются на домене,

то есть помимо контроля типов СУБД при каждом изменении данных должна проверять также значение предиката. Изменение будет отклонено, если сохраняемое значение не удовлетворяет предикату домена.

Домен играет еще одну важную роль, а именно, сравниваться могут только значения, принадлежащие одному домену. Рассмотрим в качестве примера таблицу PC, а именно, столбцы speed (тактовая частота процессора) и hd (объем жесткого диска). Оба эти столбца имеют тип integer (или smallint). Однако это совершенно разные характеристики. Достаточно сказать, что в предметной области для них используются разные единицы измерения — герцы и байты соответственно. Так вот, если мы определим эти столбцы на разных доменах, то сравнение значения одного столбца со значением другого станет недопустимым. Причем контролироваться это будет СУБД. По аналогии с категорной и ссылочной целостностью такой контроль можно было бы назвать доменной целостностью, если бы этот термин не был занят в SQL Server под проверку ограничения **CHECK**, наложенного на столбцы таблицы. А так определенная «доменная целостность» никак не ограничивает сравнения.

Не лишним будет напомнить о важности поддержания целостности на стороне СУБД. Ограничения целостности, как правило, моделируют ограничения, реально существующие в предметной области. Поскольку эти ограничения не зависят от приложений, естественно их проверять (и писать) в том месте, которое является общим для всех приложений, а именно, на стороне СУБД. Это помимо прочего:

- избавляет приложения от необходимости встраивать (и дублировать!) в них необходимые проверки;
- дает более высокий уровень безопасности; ограничения, встроенные в приложения,

легко обойти — достаточно обратиться к базе данных, минуя приложение;

- если ограничения предметной области изменятся, то соответствующие программные изменения нужно будет сделать в одном месте, а не во всех приложениях, работающих с базой данных.

Возвращаясь к доменам, уместно заметить, что и стандарт языка SQL-92 не вкладывает в понятие домена смысла «сравнимости». То, что реализовано стандартом, не более чем возможность один раз записать ограничения, а затем неоднократно применять их при определении спецификаций столбцов, то есть дает возможность избежать дублирования кодов.

В цепочке «Теория > Стандарт > Реализация» последовательно теряется строгость реляционной теории, в результате чего мы не можем вполне прозрачно взаимодействовать с реляционными СУБД разных производителей. Здесь мы хотим показать небольшой пример того, как следует обращаться с типами в SQL Server.

Итак, реально мы имеем то, что сравниваться могут значения одного типа. Для преобразования типов стандарт предлагает функцию **CAST**. То есть в общем случае мы должны преобразовать сравниваемые значения к одному типу, а затем уже выполнять операцию сравнения (или присвоения). Что же произойдет, если мы переменной (или столбцу) одного типа просто присвоим значение другого типа? Рассмотрим простой пример кода на SQL Server и Sybase ASE.T-SQL (в примерах используется SQL Server 2000):

```
1.DECLARE @vc VARCHAR(10), @mn MONEY, @ft FLOAT;
2.SELECT @vc = '499.99';
3.PRINT @vc;
4.SELECT @ft = @vc;
5.PRINT @ft;
```

Здесь мы описывает три переменные соответственно строкового типа (**VARCHAR**), денежного типа (**MONEY**) и числа с плавающей точкой (**float**). Далее строковой переменной присваиваем константу соответствующего типа, а затем присваиваем переменной типа FLOAT значение строковой переменной. В результате получаем два одинаковых результата — 499.99 (оператор **PRINT** осуществляет сообщения вывод на консоль). То, что произошло, называется неявным преобразованием типов, то есть строковое значение — ‘499.99’ было автоматически приведено к типу FLOAT и присвоено переменной @ft.

Добавим в конец кода еще пару строк:

```
1.SELECT @mn = @vc;
2.PRINT @mn;
```

В результате получим два похожих сообщения об ошибке:

Implicit conversion from data type varchar to money is not allowed. Use the CONVERT function to run this query.

и

Implicit conversion from data type money to nvarchar is not allowed. Use the CONVERT function to run this query.

в одном из которых говорится о том, что неявное преобразование типа VARCHAR к типу MONEY не допускается, а в другом — о недопустимости и обратного преобразования. Как и следовало ожидать, нам предлагается использовать явное преобразование с помощью функции **CONVERT**. Однако чтобы быть ближе к стандарту, воспользуемся функцией **CAST**:

```
1.SELECT @mn = CAST(@vc AS MONEY);
2.PRINT @mn;
```

От первого сообщения об ошибке мы избавились. Напрашивается вывод о том, что второе сообщение дает оператор PRINT. Попробуем заглянуть в справку. В BOL об операторе PRINT говорится, что переменная, которая может использоваться в этом операторе, должна быть любого допустимого

строкового типа (CHAR или VARCHAR) или же должна неявно приводиться к этому типу.

Перепишем последнюю строку:

```
1.PRINT CAST(@mn AS VARCHAR);
```

Все работает. Главный вывод, который мы отсюда извлекаем, заключается в том, что не все типы (даже если мы не видим особых причин тому) допускают неявное преобразование.

В частности, не выполняется неявное приведение типа **MONEY** к типам **CHAR**, **VARCHAR**, **NCHAR**, **NVARCHAR** и наоборот.

Примечание:

Следует отметить, что неявное преобразование к типу MONEY уже поддерживается в SQL Server в версиях 2005 и выше.

А теперь о причине, которая заставила нас написать так много слов. Из-за нашей неряшливости оказалось, что в основной и проверочной базе на сайте некоторые эквивалентные столбцы имели разные типы данных. Например, поле price в таблице PC имело тип FLOAT в одной базе и тип MONEY — в другой. В течение очень долгого времени это никак не влияло на работу сайта, но вдруг за последние несколько дней два наших участника решили использовать неявное преобразование типов в своих запросах с уже известным результатом.

Мы решили не ограничиваться извинениями, а написать этот опус об особенностях реализации, надеясь, что это принесет больше пользы. Что же касается типов, то замеченное расхождение уже приведено в соответствие.

Случайная выборка строк из таблицы в SQL Server

В свое время мы использовали случайную выборку записей для формирования списка вопросов теста. Мы делали это на клиенте,

используя функции базового языка, генерирующие псевдослучайное число в заданном диапазоне (например, функция **RND** в Visual Basic).

Однако оказалось, что достаточно просто это можно организовать и на сервере. Причем сделать это можно как аналогичными средствами (с помощью функции **RAND** в T-SQL) так и на основе типа данных uniqueidentifier, который называется глобальным уникальным идентификатором и имеет вид:

```
1.xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx,
```

где каждое x является шестнадцатеричной цифрой в диапазоне 0–9 или a–f. Например, *EEA8BF3A-7633-477A-B8C1-8C60DC9AD20C*.

Для решения нашей задачи важно то, что этот уникальный идентификатор может генерироваться автоматически при использовании функции **NEWID**. Просто напишите в **QA** (Query Analyzer) или в MS (Management Studio)

```
1.SELECT NEWID();
```

или на сайте

```
1.SELECT TOP 1 NEWID() FROM PC;
```

и вы все увидите. Причем, выполняя этот запрос снова и снова, вы будете получать все время разные значения. В этом и заключается уникальность этого идентификатора.

Так вот, идея состоит в том, чтобы в запрос, возвращающий нужное число записей, добавить вычисляемый столбец **NEWID**, по которому должна выполняться сортировка. Поскольку генерируемые значения произвольны, то мы и получим случайную выборку.

Итак, пусть нам нужно выбрать две произвольные модели из таблицы Product:

Решение 8.8.1

```
1.SELECT model
2.FROM (SELECT TOP 2 model, NEWID() [id]
3. FROM Product
4. ORDER BY [id]
5. ) x;
```

Или более просто

Решение 8.8.2

```
1.SELECT TOP 2 model
2.FROM Product
3.ORDER BY NEWID();
```

Выполним этот запрос несколько раз:

--1--
1232
1433
--2--
1276
1260
--3--

1750
1298

Если хотите, можете продолжить.

Теперь вернемся к «традиционному» способу. Функция **RAND** генерирует псевдослучайное число с плавающей точкой в диапазоне от 0 до 1.

Термин «псевдослучайное» означает, что такое число вычисляется с помощью некоторого арифметического алгоритма. То есть при одинаковых начальных (входных) условиях получаемое число будет одним и тем же. Эти начальные условия могут быть заданы явно с помощью аргумента функции, которым может быть любое число типа tinyint, int или smallint, или неявно. В последнем случае аргумент опускается, в результате чего начальное значение будет выбрано SQL Server.

Попробуем выполнить следующий запрос:

```
1. SELECT TOP 1 RAND(), RAND(350)
2. FROM Product;
```

Выполним этот запрос в **QA**. У меня получилось: 0.0485421339242268 и 0.720094900182035З7. Можно с уверенностью утверждать, что первое число у вас другое, однако второе может быть тем же самым, так как во втором случае мы задали начальное значение (350).

Попробуем теперь применить функцию **RAND** для нашей задачи, а именно, для выдачи двух случайным образом выбранных моделей:

```
1. SELECT TOP 2 model
2. FROM Product
3. ORDER BY RAND(model);
```

Получаем первые две модели в порядке возрастания их номеров. Вот где проявилась псевдослучайность. Посмотрим, что за случайные числа мы имеем:

```
1. SELECT model, RAND(model) rnd
2. FROM Product
3. ORDER BY rnd;
```

Для краткости приведу их не все:

Model	rnd
1121	0.734460921022210758
1232	0.736529180831767777
1233	0.73654781380302592
1260	0.73705090402699736
1276	0.7373490315671285
1288	0.73757262722222694
1298	0.73775895693480897
...	

Я не знаю, какой алгоритм используется для вычисления случайного числа, однако могу утверждать, что функция **RAND** ведет себя монотонно в данном диапазоне номеров моделей. Потому у нас ничего и не получилось.

В BOL приводится пример генерации последовательности случайных чисел с использованием системного времени, чтобы динамически менять начальное значение. Вот он:

```
1. SELECT RAND( (DATEPART(mm, GETDATE()) * 100000 )
2.  + (DATEPART(ss, GETDATE()) * 1000 )
3.  + DATEPART(ms, GETDATE()) )
```

```
4. );
```

Однако в таком виде запрос может быть задействован только в медленно выполняющихся пакетах, чтобы неоднократное выполнение запроса происходило не чаще, чем минимальная единица времени, применяемая в запросе (миллисекунды). Очевидно, что использование этой функции в выборке модели не будет отвечать этому условию. Однако если умножить аргумент на некоторый уникальный идентификатор, то мы можем добиться успеха (это решение было предложено Гершовичем В.И.):

```
1. SELECT model, RAND(model*(DATEPART(mm, GETDATE()) * 100000 )
2.  + (DATEPART(ss, GETDATE()) * 1000 )
3.  + DATEPART(ms, GETDATE())
4.  )
5. FROM Product
6. ORDER BY model;
```

Однако тут есть одна проблема, связанная с тем, что аргументом функции **RAND** является целое число. Поэтому если мы превысим максимально допустимое значение для целого числа (для SQL Server оно составляет 2^31-1 или 2 147 483 647), то получим следующее сообщение об ошибке:

Arithmetic overflow error converting expression to data type int.

(«Ошибка переполнения при преобразовании выражения к типу данных int».)

В этом можно убедиться, выполнив вышеприведенный запрос на сайте. Ошибка возникает где-то на номерах моделей, превышающих 2000. В аналогичных случаях нужен еще нормирующий множитель, например,

```
1. CASE
2. WHEN model < 2000
3. THEN model
4. ELSE model/10+model % 10
5. END
```

Здесь добавление model % 10 (остаток от деления на 10) делается для того, чтобы не потерять значащие цифры; в противном случае мы можем получить одинаковые значения для моделей, номера которых отличаются на единицы.

В окончательном виде решение будет выглядеть так (естественно, сортировку нужно делать по rnd, а не по model, которую мы оставили для наглядности результата).

```
1.SELECT model,
2. RAND(CASE
3. WHEN model < 2000
4. THEN model
5. ELSE model/10+model % 10
6. END *
7. (DATEPART(mm, GETDATE()) * 100000 )
8. + (DATEPART(ss, GETDATE()) * 1000 )
9. + DATEPART(ms, GETDATE())
10.       ) rnd
11.      FROM Product
12.      ORDER BY model;
```

А теперь сравните результаты:

model	Rnd
1121	0.40138073102287292
1232	0.48719939613580043
1233	0.98346802618590112
1260	0.38272122312416984
1276	0.3230194099746666
1288	0.27824305011253919

1298	0.24092941689409972

Вывод. Для решения рассматриваемой задачи проще и надежней использовать функцию **NEWID**(), которая гарантирует уникальность значений. Однако эти значения не являются числовыми. Поэтому там, где нужно получить именно число, следует обратить внимание на функцию **RAND**().

Коррелирующие подзапросы

Коррелирующие подзапросы позволяют иногда очень кратко написать запросы, которые могут выглядеть весьма громоздко при использовании других языковых средств. Напомним, что коррелирующий подзапрос — это подзапрос, который содержит ссылку на столбцы из включающего его запроса (назовем его основным). Таким образом, коррелирующий подзапрос будет выполняться для каждой строки основного запроса, так как значения столбцов основного запроса будут меняться.

Пример 8.9.1

Требуется определить дату и рейсы каждого пассажира, совершенные им в свой последний полетный день.

Иными словами, нужно определить максимальную дату полета для каждого пассажира и найти все его рейсы за эту дату. С определением максимальной даты нет никаких проблем:

```
1.SELECT id_psg, MAX(date)
2.FROM pass_in_trip
3.GROUP BY id_psg;
```

Однако тут нет рейса. Если мы попытаемся включить рейс в список вывода:

```
1.SELECT id_psg, trip_no, MAX(date)
2.FROM pass_in_trip
3.GROUP BY id_psg;
```

то получим сообщение об ошибке, так как номер рейса не используется в агрегатной функции и не входит список столбцов группировки. Если включить номер рейса в этот список:

```
1.SELECT id_psg, trip_no, MAX(date)
2.FROM pass_in_trip
3.GROUP BY id_psg, trip_no;
```

мы получим последний полет пассажира каждым рейсом, которым он летал. Это совсем не та задача, которую мы пытаемся решить. Применение коррелирующего подзапроса

```
1.SELECT id_psg, trip_no, [date]
2.FROM pass_in_trip pt_1
3.WHERE [date] = (SELECT MAX([date])
4. FROM pass_in_trip pt_2
5. WHERE pt_1.id_psg = pt_2.id_psg
6. );
```

дает то, что нужно:

id_psg	trip_no	date
10	1187	2003-04-14 00:00:00.000
9	1182	2003-04-13 00:00:00.000
8	1187	2003-04-14 00:00:00.000

6	1123	2003-04-08 00:00:00.000
5	1145	2003-04-25 00:00:00.000
3	1145	2003-04-05 00:00:00.000
3	1123	2003-04-05 00:00:00.000
2	1124	2003-04-02 00:00:00.000
1	1100	2003-04-29 00:00:00.000

Здесь для каждого рейса проверяется, совершен ли он в последний полетный день данного пассажира. При этом если таких рейсов было несколько, мы получим их все.

Очевидным недостатком приведенного решения как раз и является то, что подзапрос должен вычисляться для каждой строки основного запроса. Чтобы избежать этого, можно предложить альтернативное решение, использующее соединение таблицы Pass_in_trip с приведенным в самом начале подзапросом, который вычисляет максимальные даты по каждому пассажиру:

```
1.SELECT pt_1.id_psg, trip_no, [date]
2.FROM pass_in_trip pt_1 JOIN
3. (SELECT id_psg, MAX([date]) md
4. FROM pass_in_trip
5. GROUP BY id_psg
6. ) pt_2 ON pt_1.id_psg = pt_2.id_psg AND
7. [date] = md;
```

Внимание:

Напомним, что приведенные здесь примеры можно выполнить непосредственно на сайте, установив флажок «Без проверки» на странице с упражнениями на SELECT.

Накопительные итоги

Коррелирующие подзапросы можно использовать для вычисления накопительных итогов - задачи, часто возникающей на практике.

В предположении некоторой упорядоченности строк накопительный итог для каждой строки представляет собой сумму значений некоторого числового столбца для этой строки и всех строк, расположенных выше данной.

Другими словами, накопительный итог для первой строки в упорядоченном наборе будет равен значению в этой строке. Для любой другой строки накопительный итог будет равен сумме значения в этой строке и накопительного итога в предыдущей строке.

Рассмотрим, например, такую задачу.

Для пункта 2 по таблице Outcome_o получить на каждый день суммарный расход за этот день и все предыдущие дни.

Вот запрос, который выводит информацию о расходах на пункте 2 в порядке возрастания даты

```
1.SELECT point, date, out
2.FROM Outcome_o o
3.WHERE point = 2
4.ORDER BY date;
```

point	date	out
2	2001-03-22 00:00:00.000	1440.00
2	2001-03-29 00:00:00.000	7848.00

2	2001-04-02 00:00:00.000	2040.00

Фактически, чтобы решить задачу нам нужно добавить еще один столбец, содержащий накопительный итог (run_tot). В соответствии с темой, этот столбец будет представлять собой коррелирующий подзапрос, в котором для ТОГО ЖЕ пункта, что и у ТЕКУЩЕЙ строки включающего запроса, и для всех дат, меньших либо равных дате ТЕКУЩЕЙ строки включающего запроса, будет подсчитываться сумма значений столбца out:

```
1.SELECT point, date, out,
2.  (SELECT SUM(out)
3.   FROM Outcome_o
4.   WHERE point = o.point AND date <= o.date) run_tot
5.FROM Outcome_o o
6.WHERE point = 2
7.ORDER BY point, date;
```

point	date	out	run_tot
2	2001-03-22 00:00:00.000	1440.00	1440.00
2	2001-03-29 00:00:00.000	7848.00	9288.00
2	2001-04-02 00:00:00.000	2040.00	11328.00

Собственно, использование пункта 2 продиктовано желанием уменьшить результирующую выборку. Чтобы получить накопительные итоги для каждого из пунктов, имеющихся в таблице Outcome_o, достаточно закомментировать строку

```
1.WHERE point = 2
```

Ну а чтобы получить "сквозной" накопительный итог для всей таблицы нужно, видимо, убрать условие на равенство пунктов:

```
1.point= o.point
```

Однако при этом мы получим один и тот же накопительный итог для разных пунктов, работавших в один и тот же день. Вот подобный фрагмент из результирующей выборки,

point	date	out	run_tot
1	2001-03-29 00:00:00.000	2004.00	33599.00
2	2001-03-29 00:00:00.000	7848.00	33599.00

Но это не проблема, если понять, что же мы хотим в итоге получить. Если нас интересует накопление расхода по дням, то нужно из выборки вообще исключить пункт и суммировать расходы по дням:

```
1.SELECT date, SUM(out) out,
2.   (SELECT SUM(out)
3.    FROM Outcome_o
4.    WHERE date <= o.date) run_tot
5.FROM Outcome_o o
6.GROUP BY date
7.ORDER BY date;
```

В противном случае, нам нужно указать порядок, в котором к накопительному итогу будут добавляться расходы пунктов в случае, когда у нескольких пунктов совпадает дата. Например, упорядочим их по возрастанию номеров:

```
1.SELECT point, date, out,
2.   (SELECT SUM(out)
3.    FROM Outcome_o
```

```
4.    WHERE date < o.date OR ( date = o.date AND point <=
   o.point)) run_tot
5. FROM Outcome_o o
6. ORDER BY date, point;
```

Расширение поддержки **оконных функций** в **SQL Server** 2012 позволяет решить задачу о накопительных итогах совсем просто.

Применительно к нашей задаче речь идет о следующих появившихся возможностях:
1. Использование сортировки в предложении OVER при применении агрегатных функций.
2. Спецификация диапазона, к значениям которого применяется агрегатная функция. При этом диапазон может быть как ограниченным, так и неограниченным, скажем, от текущей строки до конца или начала отсортированного набора.

Т.е. мы можем получить накопительный итог, упорядочив данные по дате и подсчитав сумму от текущей строки и (неограниченно) выше, причем сделать это с помощью одной функции!

Задачу о накопительных итогах для пункта 2, которая рассматривалась на предыдущей странице, теперь мы можем решить так:

```
1. SELECT point, date, out,
2.     SUM(out) OVER (PARTITION BY point
3.             ORDER BY date -- сортировка по дате
4.             RANGE -- диапазон
5.             UNBOUNDED -- неограниченный
6.             PRECEDING -- от текущей строки и выше
7.             )
8.     FROM Outcome_o o
9.     WHERE point = 2
10.         ORDER BY point, date;
```

Для получения накопительных итогов по каждому пункту отдельно уберем из предыдущего запроса условие отбора по пункту:

```
1. SELECT point, date, out,
2.     SUM(out) OVER (PARTITION BY point ORDER BY date RANGE
   UNBOUNDED PRECEDING)
3.     FROM Outcome_o o
4.     ORDER BY point, date;
```

Представленные здесь решения будут работать в PostgreSQL и Oracle. Что касается MySQL, то там поддержка оконных функций реализована в версии 8.0.

Если нам потребуется подсчитать накопительный итог с учетом не всех предшествующих строк, а, скажем, двух. В этом случае мы можем использовать следующий синтаксис:

```
1. SELECT point,date, out,
2. SUM(out) OVER (ORDER BY date,point ROWS BETWEEN 2
   PRECEDING AND CURRENT ROW)
3. FROM Outcome_o o
4. ORDER BY date,point;
```

Суммирование происходит в окне, которое задается предложением

```
1. ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
```

Здесь задается диапазон строк (rows) между (between) текущей строкой (current row) и двумя строками выше (2 preceding).

В этом примере рассматриваются все строки таблицы, упорядоченные по date, point (сортировка по point устраняет неоднозначность, поскольку несколько пунктов могут иметь отчетность в один и тот же день).

Преобразование даты в строку

Речь пойдет о форматировании даты. Форматирование даты обычно используется для представления даты/времени в отчетах, подготовленных для печати. Как правило, такое форматирование выполняется средствами разработки отчетов. Однако и на уровне СУБД есть подобные возможности. Мы не будем здесь обсуждать вопрос о том, где лучше выполнять данное форматирование. Отметим лишь, что в ряде задач на сайте sql-ex.ru требуется представить результат выполнения запроса в том или ином конкретном формате.

SQL Server

Для форматирования даты в SQL Server используется функция CONVERT.

Будем рассматривать в качестве примера дату, возвращаемую следующим запросом:

```
1. SELECT b_datetime FROM utb WHERE b_datetime='2002-06-01T01:13:39.000';
```

b_datetime
2002-06-01 01:13:39.000

Например, чтобы получить из этого представления временной метки только дату в привычном нам формате "dd-mm-yyyy", достаточно написать

```
1. SELECT CONVERT(varchar, b_datetime, 105)
2. FROM utb WHERE b_datetime='2002-06-01T01:13:39.000';
```

01-06-2002

Для вывода в формате "mm-dd-yyyy" можно в функции CONVERT поменять параметр стиля на 110:

```
1. SELECT CONVERT(varchar,b_datetime,110)
2. FROM utb WHERE b_datetime='2002-06-01T01:13:39.000';
```

06-01-2002

Для вывода одной лишь компоненты времени без миллисекунд используется стиль 108:

```
1. SELECT CONVERT(varchar, b_datetime, 108)
2. FROM utb WHERE b_datetime='2002-06-01T01:13:39.000';
```

01:13:39

MySQL

В MySQL для форматирования даты используется функция **DATE_FORMAT**, в которой вторым параметром является маска, в соответствии с которой форматируется первый параметр типа даты/времени. Рассмотренные выше примеры для MYSQL можно переписать следующим образом:

```
1. SELECT date_format(b_datetime,'%d-%m-%Y')
2. FROM utb WHERE b_datetime='2002-06-01T01:13:39.000';
```

01-06-2002

```
1. SELECT date_format(b_datetime,'%m-%d-%Y')
2. FROM utb WHERE b_datetime='2002-06-01T01:13:39.000';
```

06-01-2002

```
1. SELECT date_format(b_datetime,'%H:%i:%s')
2. FROM utb WHERE b_datetime='2002-06-01T01:13:39.000';
```

01:13:39

Заметим, что "H" используется для представления 24-часового формата времени, а "h" - для 12-часового формата.

PostgreSQL & Oracle

Эти СУБД используют для форматирования функцию **TO_CHAR** с интуитивно понятной маской. Наши примеры для PostgreSQL будут выглядеть так:

```
1. SELECT to_char(b_datetime,'dd-mm-yyyy')
2. FROM utb WHERE b_datetime='2002-06-01T01:13:39.000';
```

01-06-2002

```
1. SELECT to_char(b_datetime,'mm-dd-yyyy')
2. FROM utb WHERE b_datetime='2002-06-01T01:13:39.000';
```

06-01-2002

```
1. --12-часовой формат
2. SELECT to_char(b_datetime,'hh12:mi:ss')
3. FROM utb WHERE b_datetime='2002-06-01T01:13:39.000';
4. --24-часовой формат
5. SELECT to_char(b_datetime,'hh24:mi:ss')
6. FROM utb WHERE b_datetime='2002-06-01T01:13:39.000';
```

01:13:39

Для Oracle принципиальных отличий нет. Чтобы примеры, приведенные для PostgreSQL, работали в Oracle, нам нужно знать, что строковое представление даты нужно явно преобразовывать к соответствующему темпоральному типу (при сравнениях с датой). Кроме того, не поддерживается стандартное представление временной метки с разделителем "T" между датой и временем. Например, последний пример в Oracle будет иметь вид:

```
1. SELECT to_char(b_datetime,'hh24:mi:ss')
2. FROM utb WHERE b_datetime=timestamp'2002-06-01
   01:13:39.000';
```

Новое в стандарте и реализациях языка SQL

Оператор MERGE

Если головной корабль из таблицы Outcomes отсутствует в таблице Ships, добавить его в Ships, приняв имя класса, совпадающим с именем корабля, и год спуска на воду, равным году самого раннего сражения, в котором участвовал корабль. Если же корабль присутствует в Ships, но дата спуска на воду его неизвестна, установить его равным году самого раннего сражения, в котором участвовал корабль.

Эта задача подразумевает выполнение двух разных операторов (INSERT и UPDATE) на одной таблице (Ships) в зависимости от наличия/отсутствия связанных записей в другой таблице (Outcomes).

Для решения подобных задач стандарт предоставляет оператор **MERGE**. Рассмотрим его использование на примере решения данной задачи в SQL Server.

Для начала напишем запрос, который вернет нам головные корабли из таблицы Outcomes, т.е. корабли, у которых имя класса совпадает с именем корабля:

```
1. SELECT ship, ship class FROM Outcomes O JOIN Classes C ON
   C.class=O.ship;
```

ship	class
Bismarck	Bismarck
Tennessee	Tennessee

Теперь добавим соединение с таблицей Battles и выполним группировку, чтобы найти минимальный год сражений каждого такого корабля:

```
1.SELECT year(MIN(date)) min_year, ship, ship class
2.FROM outcomes O JOIN battles B ON O.battle= B.name
3.JOIN Classes C ON C.class=O.ship GROUP BY ship;
```

min_year	ship	class
1941	Bismarck	Bismarck
1944	Tennessee	Tennessee

Исходные данные готовы. Теперь мы можем перейти к написанию оператора MERGE.

```
1.MERGE Ships AS target  -- таблица, которая будет меняться
2.USING (SELECT year(MIN(date)), ship, ship
3.       FROM outcomes O JOIN battles B ON O.battle= B.name
4.               JOIN Classes C ON C.class=O.ship GROUP BY
  ship
5.       ) AS source (min_year,ship, class)  -- источник
  данных, который мы рассмотрели выше
6.ON (target.name = source.ship)  -- условие связи между
  источником и изменяемой таблицей
7.WHEN MATCHED AND target.launched IS NULL -- если головной
  корабль есть в таблице Ships
8.           -- с неизвестным годом
9.    THEN UPDATE SET target.launched = source.min_year --
  обновление
10.     WHEN NOT MATCHED  -- если головного корабля нет в
  таблице Ships
11.         THEN INSERT VALUES(source.ship, source.class,
  source.min_year) -- вставка
12.     OUTPUT $action, inserted.*, deleted.*; -- можно
  вывести измененные строки
```

$action	name	class	launched	name	class	launched
INSERT	Bismarck	Bismarck	1941	NULL	NULL	NULL

Предложение **OUTPUT** позволяет вывести измененные строки. Автоматически создаваемые рабочие таблицы inserted и deleted имеют тот же смысл, что и при использовании в триггерах, т.е. inserted содержит строки, которые были добавлены в изменяемую таблицу, а deleted - удаленные из нее строки.

Поскольку удаления в нашем запросе не было, то соответствующие столбцы имеют значения NULL. Столбец $action содержит название выполненной операции. В нашем случае была выполнена только вставка, поскольку корабль Tennessee содержится в таблице Ships с известным годом спуска на воду:

```
1.SELECT * FROM Ships WHERE name='Tennessee';
```

name	class	launched
Tennessee	Tennessee	1920

Инструкция MERGE может иметь не больше двух предложений **WHEN MATCHED**.

Если указаны два предложения, то первое предложение должно сопровождаться дополнительным условием (что имеет место в нашем случае - AND target.launched IS NULL). Для любой строки второе предложение WHEN MATCHED применяется только в том случае, если не применяется первое.

Если имеются два предложения WHEN MATCHED, одно должно указывать действие UPDATE, а другое — DELETE. Т.е. если мы добавим в оператор предложение

```
1.WHEN MATCHED THEN DELETE
```

то удалим корабль Tennessee:

$action	name	class	launched	name	class	launched
INSERT	Bismarck	Bismarck	1941	NULL	NULL	NULL

DELETE	NULL	NULL	NULL	Tennessee	Tennessee	1920

Инструкцию MERGE нельзя использовать для обновления одной строки более одного раза, а также для обновления и удаления одной и той же строки.

Предложение **WHEN NOT MATCHED [BY TARGET] THEN INSERT** используется для вставки строк из источника, не совпадающих со строками в изменяемой таблице согласно условию связи. В нашем примере такой строкой является строка, относящаяся к кораблю Bismarck. Инструкция MERGE может иметь только одно предложение WHEN NOT MATCHED.

Наконец, оператор MERGE может включать предложение WHEN NOT MATCHED BY SOURCE THEN.

Оно воздействует на те строки изменяемой таблицы, для которых нет соответствия в таблице-источнике. Например, если бы мы хотели удалить из таблицы Ships головные корабли, не принимавшие участие в сражениях, то добавили бы следующее предложение:

```
1.WHEN NOT MATCHED BY SOURCE AND target.name=target.class
  THEN DELETE
```

Результат:

$action	name	class	launched	name	class	launched
DELETE	NULL	NULL	NULL	Iowa	Iowa	1943
DELETE	NULL	NULL	NULL	Kongo	Kongo	1913
DELETE	NULL	NULL	NULL	North Carolina	North Carolina	1941
DELETE	NULL	NULL	NULL	Renown	Renown	1916
DELETE	NULL	NULL	NULL	Revenge	Revenge	1916
DELETE	NULL	NULL	NULL	Yamato	Yamato	1941
INSERT	Bismarck	Bismarck	1941	NULL	NULL	NULL

При помощи этого предложения можно удалять или обновлять строки. Инструкция MERGE может иметь не более двух предложений WHEN NOT MATCHED BY SOURCE. Если указаны два предложения, то первое предложение должно иметь дополнительное условие (как в нашем примере). Для любой выбранной строки второе предложение WHEN NOT MATCHED BY SOURCE применяется только в тех случаях, если не применяется первое. Кроме того, если имеется два предложения WHEN NOT MATCHED BY SOURCE, то одно должно выполнять UPDATE, а другое — DELETE.

Функции ранжирования

Реляционная модель исходит из того факта, что строки в таблице не имеют порядка, являющегося прямым следствием теоретико-множественного подхода. Поэтому наивными выглядят вопросы новичков, спрашивающих: "А как мне получить последнюю добавленную в таблицу строку?" Ответом на вопрос будет "никак", если в таблице не предусмотрен столбец, содержащий дату вставки строки, или не используется последовательная нумерация строк, реализуемая во многих СУБД с помощью столбца с автоинкрементируемым значением. Тогда можно выбрать строку с максимальным значением даты или счетчика.

Вопрос о последней строке имеет смысл только в аспекте выдачи результата выполнения запроса, при этом предполагается некоторая сортировка, которая задается с помощью предложения ORDER BY в операторе SELECT. Если никакая сортировка не задана (предложение ORDER BY отсутствует), то полагаться на то, что порядок вывода строк, полученных при выполнении запроса сегодня, останется таким же и завтра, нельзя, т.к. этот порядок зависит от плана, который выбирает оптимизатор запросов для их выполнения. А план

может меняться, и зависит это от многих причин, которые мы здесь опустим.

Теоретически каждая строка запроса обрабатывается независимо от других строк. Однако на практике часто требуется при обработке строки соотносить ее с предыдущими или последующими строками (например, для получения нарастающих итогов), выделять группы строк, обрабатываемые независимо от других и т.д. В ответ на потребности практики в ряде СУБД в языке SQL появились соответствующие конструкции, в частности, функции ранжирования и оконные (аналитические) функции, которые де-юре были зафиксированы в стандарте SQL:2003. В SQL Server ранжирующие функции появились в версии 2005.

Функция ROW_NUMBER

Функция ROW_NUMBER, как следует из ее названия, нумерует строки, возвращаемые запросом. С ее помощью можно выполнить более сложное упорядочивание строк в отчете, чем то, которое дает предложение ORDER BY в рамках Стандарта SQL-92.

До появления этой функции для нумерации строк, возвращаемых запросом, приходилось использовать довольно сложный интуитивно непонятный алгоритм, изложенный в параграфе. Единственным достоинством данного алгоритма является то, что он будет работать практически на всех СУБД, поддерживающих стандарт SQL-92.

Примечание:

Естественно, можно выполнить нумерацию средствами процедурных языков, используя при этом курсоры и/или

временные таблицы. Но мы здесь говорим о "чистом" SQL.

Используя функцию ROW_NUMBER можно:

- задать нумерацию, которая будет отличаться от порядка сортировки строк результирующего набора;
- создать "несквозную" нумерацию, т.е. выделить группы из общего множества строк и пронумеровать их отдельно для каждой группы;
- использовать одновременно несколько способов нумерации, поскольку, фактически, нумерация не зависит от сортировки строк запроса.

Проще всего возможности функции ROW_NUMBER показать на простых примерах, к чему мы и переходим.

Пример 1.

Пронумеровать все рейсы из таблицы Trip в порядке возрастания их номеров. Выполнить сортировку по {id_comp, trip_no}.

Решение

```
1. SELECT row_number() over(ORDER BY trip_no) num,
2.  trip_no, id_comp
3. FROM trip
4. WHERE ID_comp < 3
5. ORDER BY id_comp, trip_no;
```

Предложение OVER, с которым используется функция ROW_NUMBER задает порядок нумерации строк. При этом используется дополнительное предложение ORDER BY, которое не имеет отношения к порядку вывода строк запроса. Если вы посмотрите на результат, то заметите, что порядок строк в результирующем наборе и порядок нумерации не совпадают:

num	trip_no	id_comp
3	1181	1
4	1182	1
5	1187	1
6	1188	1
7	1195	1
8	1196	1
1	1145	2
2	1146	2

Примечание:

Условие отбора id_comp<3 использовано лишь с целью уменьшения размера выборки.

Конечно, мы можем потребовать выдачу в порядке нумерации, переписав последнюю строку в виде

```
1.ORDER BY trip_no
```

(или, что то же самое, order by num).

Или, наоборот, пронумеровать строки в порядке заданной сортировки:

```
1.SELECT row_number() over(ORDER BY id_comp, trip_no) num,
2. trip_no, id_comp
3.FROM trip
4.WHERE ID_comp<3
```

```
5.ORDER BY id_comp, trip_no;
```

num	trip_no	id_comp
1	1181	1
2	1182	1
3	1187	1
4	1188	1
5	1195	1
6	1196	1
7	1145	2
8	1146	2

А если требуется пронумеровать рейсы для каждой компании отдельно? Для этого нам потребуется еще одна конструкция в предложении OVER - PARTITION BY.

Конструкция PARTITION BY задает группы строк, для которых выполняется независимая нумерация. Группа определяется равенством значений в списке столбцов, перечисленных в этой конструкции, у строк, составляющих группу.

Пример 2.

Пронумеровать рейсы каждой компании отдельно в порядке возрастания номеров рейсов.

```
1.SELECT row_number() over(partition BY id_comp ORDER BY
  id_comp,trip_no) num,
2. trip_no, id_comp
3.FROM trip
4.WHERE ID_comp < 3
5.ORDER BY id_comp, trip_no;
```

PARTITION BY id_comp означает, что рейсы каждой компании образуют группу, для которой и выполняется независимая нумерация. В результате получим:

num	trip_no	id_comp
1	1181	1
2	1182	1
3	1187	1
4	1188	1
5	1195	1
6	1196	1
1	1145	2
2	1146	2

Отсутствие конструкции PARTITION BY, как это было в первом примере, означает, что все строки результирующего набора образуют одну единственную группу.

В **MySQL** ранжирующих/оконных функций не было до версии 8.0, однако была возможность использовать переменные непосредственно в запросе SQL. В частности, с помощью переменных можно решить задачу нумерации строк запроса. Продемонстрируем это на примере, который рассматривался на предыдущей странице.

```
1.SELECT @i:=@i+1 num,
2.      trip_no, id_comp
3.     FROM Trip, (SELECT @i:=0) X
4.     WHERE ID_comp < 3
5.     ORDER BY id_comp, trip_no;
```

num	id_comp	trip_no
1	1	1181
2	1	1182
3	1	1187
4	1	1188
5	1	1195
6	1	1196
7	2	1145
8	2	1146

В третьей строке запроса выполняется инициализация переменной и присваивается ей начальное значение. В итоге каждая строка таблицы Trip будет соединяться со строкой из одного столбца, содержащего 0 (просто декартово произведение).

В первой строке запроса значение переменной инкрементируется на 1, что происходит при вычислении каждой следующей строки в порядке, заданном предложением ORDER BY. В итоге мы получаем нумерацию строк в порядке сортировки.

Если вы опустите инициализацию переменной, то можете получить правильный результат. Но это не гарантировано, в чем можно убедиться, повторно выполнив этот же запрос в текущей сессии соединения с базой данных. Вы должны получить продолжение нумерации с максимального значения переменной @i, достигнутого на предыдущем запуске скрипта.

Мы также можем перенумеровать строки для каждой компании отдельно, т.е. сымитировав поведение **PARTITION BY** в запросе

```
1. SELECT row_number() over(PARTITION BY id_comp ORDER BY
   id_comp,trip_no) num,
```

```
2.      trip_no, id_comp
3.     FROM Trip
4.     WHERE ID_comp < 3
5.     ORDER BY ID_comp, trip_no;
```

Идея решения состоит в следующем. Введем еще одну переменную для хранения номера компании. При инициализации присвоим ей несуществующий номер (например, 0). Затем для каждой строки будем проверять, совпадает ли номер с номером компании текущей строки. Если значения совпадают, будем инкрементировать, если нет, сбрасывать в 1. Наконец, будем присваивать переменной номер компании из текущей строки. Дело в том, что проверка выполняется до присвоения, тем самым мы сравниваем текущее значение номера компании с номером компании из предыдущей строки (в заданном порядке сортировки). Теперь сам запрос.

```
1.SELECT
2.CASE WHEN @comp=id_comp THEN @i:=@i+1 ELSE @i:=1 END num,
3.@comp:=id_comp id_comp, trip_no
4.     FROM Trip, (SELECT @i:=0, @comp:=0) X
5.     WHERE ID_comp < 3
6.     ORDER BY ID_comp, trip_no;
```

num	id_comp	trip_no
1	1	1181
2	1	1182
3	1	1187
4	1	1188
5	1	1195
6	1	1196
1	2	1145
2	2	1146

Или, коль скоро вы отошли от стандарта, можно использовать функцию **IF**, чтобы сократить запись:

```
1.SELECT
2.IF(@comp=id_comp, @i:=@i+1, @i:=1) num,
3.@comp:=id_comp id_comp, trip_no
4.    FROM Trip, (SELECT @i:=0, @comp:=0) X
5.    WHERE ID_comp < 3
6.    ORDER BY id_comp, trip_no;
```

Проверить эти запросы вы можете из консоли, выбрав из списка MySQL.

Функции RANK() и DENSE_RANK()

Эти функции, как и функция ROW_NUMBER(), тоже нумеруют строки, но делают это несколько отличным способом. Это отличие проявляется в том, что строки, которые имеют одинаковые значения в столбцах, по которым выполняется упорядочивание, получают одинаковые номера (ранги). Например, значения (отсортированные по возрастанию)

1
5
6
6
6

получат такие номера:

1	1
5	2
6	3
6	3
6	3

Возникает вопрос, с какого номера продолжится нумерация, если, скажем, в последовательности чисел появится 7 и т.д.? Здесь есть два варианта:

1) с номера 4, т.к. это следующий номер по порядку;

2) с номера 6, т.к. следующая строка будет шестая по счету.

Такая "неоднозначность" и привела к появлению двух функций вместо одной - RANK и DENSE_RANK, первая из которых продолжит нумерацию с 6, а вторая (плотная) - с 4.

Рассмотрим несколько примеров. Начнем с демонстрации отличия в поведении функций RANK и ROW_NUMBER:

```
1.SELECT *, ROW_NUMBER() OVER(ORDER BY type) num,
2.RANK() OVER(ORDER BY type) rnk
3.FROM Printer;
```

Здесь в двух последних столбцах выводятся значения сравниваемых функций при упорядочивании строк по столбцу *type*:

code	model	color	type	price	num	rnk
2	1433	y	Jet	270,00	1	1
3	1434	y	Jet	290,00	2	1
1	1276	n	Laser	400,00	3	3
6	1288	n	Laser	400,00	4	3
4	1401	n	Matrix	150,00	5	5
5	1408	n	Matrix	270,00	6	5

Как и следовало ожидать, ROW_NUMBER пронумеровывает весь набор строк, возвращаемых запросом. Функция RANK, как оказалось, работает по второму из рассмотренных выше варианту, т.е. следующим номером после строк с одинаковым рангом будет номер строки.

А теперь сравним "плотный" и "неплотный" ранги:

```
1.SELECT *, RANK() OVER(ORDER BY type) rnk,
2.DENSE_RANK() OVER(ORDER BY type) rnk_dense
3.FROM Printer;
```

code	model	color	type	price	rnk	rnk_dense
2	1433	y	Jet	270,00	1	1
3	1434	y	Jet	290,00	1	1
1	1276	n	Laser	400,00	3	2
6	1288	n	Laser	400,00	3	2
4	1401	n	Matrix	150,00	5	3

5	1408	n	Matrix	270,00	5	3

Следует также обратить вниманию на порядок, в котором выводятся строки результирующего набора. Поскольку оператор SELECT в нашем примере не имеет предложения ORDER BY, а для вычисления рангов используется одинаковое упорядочивание по столбцу *type*, то и результат выводится в том же порядке. В целях оптимизации, если вам не нужно какое-либо другое упорядочение результирующего набора, используйте этот факт, чтобы не выполнять лишние сортировки, которые ухудшают производительность запроса.

Как и для функции ROW_NUMBER, в предложении OVER может использоваться конструкция PARTITION BY, разбивающая весь набор строк, возвращаемых запросом, на группы, к которым затем применяется соответствующая функция.

Запрос

```
1. SELECT *, RANK() OVER(PARTITION BY type ORDER BY price)
   rnk FROM Printer;
```

позволяет в каждой группе, определяемой типом принтера, ранжировать модели по цене в порядке ее возрастания:

code	model	color	type	price	rnk
2	1433	y	Jet	270,00	1
3	1434	y	Jet	290,00	2
1	1276	n	Laser	400,00	1
6	1288	n	Laser	400,00	1
4	1401	n	Matrix	150,00	1
5	1408	n	Matrix	270,00	2

А вот как можно выбрать самые дешевые модели в каждой категории:

```
1.SELECT model, color, type, price
2.FROM (
3.SELECT *, RANK() OVER(PARTITION BY type ORDER BY price)
  rnk
4. FROM Printer
5.) Ranked_models
6.WHERE rnk = 1;
```

model	color	type	price
1433	y	Jet	270,00
1276	n	Laser	400,00
1288	n	Laser	400,00
1401	n	Matrix	150,00

Запрос можно было бы написать короче, если бы функцию RANK можно было бы применять в предложении WHERE, т.к. само значение ранга нам не требуется. Однако это запрещено (как и для других ранжирующих функций), по крайней мере, в SQL Server.

Наконец, рассмотрим еще один пример.

Пример. Найти производителей, которые производят более 2-х моделей PC.

Эта задача имеет традиционное решение через агрегатные функции:

```
1.SELECT maker FROM Product
2.WHERE type = 'PC'
3.GROUP BY maker
4.HAVING COUNT(*) > 2;
```

Однако эту задачу можно решить и с помощью функции RANK. Идея состоит в следующем: ранжировать модели каждого производителя по уникальному ключу и выбрать только тех производителей, модели которых достигают ранга 3:

```
1. SELECT maker
2. FROM (
3. SELECT maker, RANK() OVER(PARTITION BY maker ORDER BY
   model) rnk
4. FROM Product
5. WHERE type = 'PC'
6. ) Ranked_makers
7. WHERE rnk = 3;
```

И в одном, и в другом случае, естественно, мы получим один и тот же результат:

maker
E

Еще раз повторю: упорядочивание в последнем случае должно быть выполнено по уникальной комбинации столбцов, т.к., в противном случае, моделей может быть больше трех, а ранг меньше (например, 1, 2, 2,...). В нашем случае данное условие выполнено, т.к. упорядочивание выполняется по столбцу model, который является первичным ключом в таблице Product.

Кстати, планы выполнения этих запросов демонстрируют одинаковые стоимости наиболее расходных операций – сканирования таблицы и сортировку (которая я первом случае присутствует неявно и вызвана операцией группировки).

Пример использования DENSE_RANK

Часто встречается задача нахождения N-го по величине значения из набора значений некоторого столбца таблицы, например:

Найти второе по величине значение цены в таблице PC.

Давайте выведем отсортированный список значений цены из таблицы PC для контроля, добавив столбцы со значениями ранжирующих функций:

```
1. SELECT price, DENSE_RANK() OVER(ORDER BY price DESC)
   dense_rnk,
2. RANK() OVER(ORDER BY price DESC) rnk,
3. ROW_NUMBER() OVER(ORDER BY price DESC) rn
4. FROM PC ORDER BY price DESC;
```

price	dense_rnk	rnk	rn
980,00	1	1	1
970,00	2	2	2
950,00	3	3	3
850,00	4	4	4
850,00	4	4	5
850,00	4	4	6
600,00	5	7	7
600,00	5	7	8
400,00	6	9	9
350,00	7	10	10
350,00	7	10	11
350,00	7	10	12

В рамках стандарта **SQL-92** эту задачу можно решить следующим образом:

```
1. SELECT MAX(price) "2nd_price"  FROM PC
2. WHERE price < (SELECT MAX(price) FROM PC);
```

2nd_price
970,00

Т.е. мы находим значение максимума среди всех значений, меньших максимального. А если нам потребуется найти значение третьей по величине цены? Можно поступить аналогично:

```
1. SELECT MAX(price) "3rd_price" FROM PC WHERE price <
2. (
3. SELECT MAX(price) FROM PC
4. WHERE price < (SELECT MAX(price) FROM PC)
5. );
```

3rd_price
950,00

А как найти N-е значение цены? Следуя используемой логике, мы можем добавлять новые "матрешки" к уже существующим вплоть до N-ой. Это решение никак не назовешь универсальным.

Для решения подобных задач хорошо подходит функция **DENSE_RANK**. Например, исходную задачу с помощью этой функции можно решить так:

```
1. SELECT DISTINCT price FROM(
2. SELECT DENSE_RANK() OVER(ORDER BY price DESC) rnk, price
   FROM PC
3. ) X WHERE rnk=2;
```

А чтобы найти любую другую порядковую цену (например, третью), достаточно поменять константу в условиях отбора:

```
1. SELECT DISTINCT price FROM(
2. SELECT DENSE_RANK() OVER(ORDER BY price DESC) rnk, price
   FROM PC
3. ) X WHERE rnk=3;
```

Следует отметить, что использование DENSE_RANK, а не RANK, обусловлено тем, что в случае наличия одинаковых цен, значения, возвращаемые функцией RANK, будут иметь пропуски (рассмотрите задачу нахождения пятой по величине цены). Если же ставить задачу нахождения неуникального N-го значения, то можно использовать функцию ROW_NUMBER (например, третий человек в шеренге по росту). А если значения в таблице уникальны, то решение с помощью любой из этих функций даст один и тот же результат.

Функция NTILE

Задача. Распределить баллончики по 3-м группам поровну. Группы заполняются в порядке возрастания v_id.

Эту задачу решает функция ранжирования **NTILE**, которая появилась в SQL Server 2008.

Эта функция возвращает номер группы, в которую попадает соответствующая строка результирующего набора.

```
1. SELECT *, NTILE(3) OVER(ORDER BY v_id) gr FROM utv ORDER
   BY v_id;
```

Параметром функции NTILE является число групп. Остальное вам уже известно. :-)

Если мы захотим распределить порознь баллончики каждого цвета, то, как и для других функций ранжирования, можно добавить конструкцию **PARTITION BY** в предложение **OVER**:

```
1.SELECT *, NTILE(3) OVER(PARTITION BY v_color ORDER BY
  v_id) gr
2.FROM utv ORDER BY v_color, v_id;
```

Обратите внимание на группы синего цвета (B). В двух первых группах оказалось по 6 баллончиков, а в третьей только 5. В случае, когда число строк не делится нацело на число групп, функция NTILE помещает в последние группы на одну строку меньше, чем в первые.

Наконец, если аргумент функции NTILE окажется больше числа строк, то будет сформировано количество групп, равное числу строк, и в каждой группе окажется по одной строке.

Оконные функции

Фактически мы познакомились с этими функциями, когда рассматривали функции ранжирования. Только сейчас мы будем использовать агрегатные функции вместо функций, которые задают номер/ранг строки. Есть еще одно отличие (в реализации Майкрософт SQL Server 2005/2008) – предложение **OVER**() не содержит дополнительного предложения **ORDER BY**, поскольку значение агрегата не зависит от сортировки строк в «окне».

Как и ранее, предложение **PARTITION BY** определяет «окно», т.е. набор строк, характеризуемых равенством значений списка выражений, указанного в этом предложении. Если предложение PARTITION BY отсутствует, то агрегатные функции применяются ко всему результирующему набору строк запроса. В отличие от классической группировки, где мы получаем на каждую группу одну строку, которая может содержать агрегатные значения, подсчитанные для каждой такой группы, здесь мы можем добавить агрегат к детализированным (несгруппированным) строкам. Рассмотрим несколько примеров использования оконных функций.

Постраничная разбивка записей (пейджинг)

Такая задача часто возникает в тех случаях, когда количество строк, возвращаемых запросом, превышает разумный размер страницы. Примером может служить представление результатов поисковой выдачи или сообщений на форумах сайтов. Результаты сортируются по некоторым критериям (например, по релевантности или по дате сообщения), а затем разбиваются по N строк на страницу. Главная проблема здесь состоит в том, чтобы не загружать на клиента весь набор строк, а выводить только запрашиваемую пользователем страницу (мало кто просматривает все страницы подряд). При отсутствии такой возможности пришлось бы выполнять разбивку по страницам программными средствами клиента, что негативно сказывается на трафике и времени загрузки страницы.

Итак, нам нужно вывести, наряду с детализированными данными, общее число строк (или число страниц) и номер страницы для каждой записи, возвращаемой запросом. Если нам это удастся сделать, то чтобы не возвращать весь результирующий набор на клиента, мы можем на базе этого запроса создать хранимую процедуру, в которую в качестве входных параметров будет передаваться требуемое число записей на странице и номер страницы, а возвращаться набор записей с затребованной страницы. Такой подход будет экономно расходовать трафик, а навигация по страницам будет использовать кэшированный план исполнения хранимой процедуры.

Для примера рассмотрим разбивку по 2 записи на страницу строк из таблицы Laptop, упорядоченных по убыванию цены.

Вот таким образом можно добавить столбец, содержащий общее число строк в таблице:

```
1.SELECT *, COUNT(*) OVER() AS total
2.FROM Laptop;
```

Заметим, что подобное можно было сделать в рамках стандарта SQL-92 с помощью подзапроса:

```
1.SELECT *, (SELECT COUNT(*) FROM Laptop) AS total
2.FROM Laptop;
```

Однако представьте себе, что мы используем не простую таблицу (Laptop), а громоздкий запрос, который может содержать десятки и сотни строк. При этом «оконный» вариант не претерпел бы изменений, а в «классическом» случае пришлось бы полностью дублировать код запроса в подзапросе для вычисления числа строк.

Чтобы посчитать число страниц, воспользуемся следующим простым алгоритмом:

- если число строк запроса нацело делится на число записей на странице, то результат целочисленного деления одного на другое дает число страниц;
- если существует ненулевой остаток целочисленного деления числа строк запроса на число записей на странице, то к результату целочисленного деления добавляем единицу (последняя страница неполная).

Это алгоритм реализуется стандартными средствами с помощью оператора CASE:

```
1. SELECT *,
2.        CASE WHEN total % 2 = 0 THEN total/2 ELSE total/2
   + 1 END AS num_of_pages
3. FROM (
4.        SELECT *, COUNT(*) OVER() AS total
5.        FROM Laptop
6.        ) X;
```

Чтобы получить для каждой строки запроса номер страницы, на которую она должна попасть, мы можем применить аналогичный алгоритм, только применить его нужно не к общему числу строк (total), а к номеру строки. Этот номер строки мы сможем получить с помощью ранжирующей функции ROW_NUMBER, выполнив требуемую по условию сортировку по цене:

```
1. SELECT *,
2.        CASE WHEN num % 2 = 0 THEN num/2 ELSE num/2 + 1 END
   AS page_num,
3.        CASE WHEN total % 2 = 0 THEN total/2 ELSE total/2
   + 1 END AS num_of_pages
4. FROM (
5.        SELECT *, ROW_NUMBER() OVER(ORDER BY price DESC) AS
   num,
6.               COUNT(*) OVER() AS total
7.        FROM Laptop
8.) X;
```

code	model	speed	ram	hd	price	screen	num	total	page_num	num_of_pages
3	1750	754	128	12	1200,00	14	1	6	1	3
5	1752	750	128	10	1150,00	14	2	6	1	3
4	1298	600	64	10	1050,00	15	3	6	2	3
2	1321	500	64	8	970,00	12	4	6	2	3

6	1298	450	64	10	950,00	12	5	6	3	3
1	1298	350	32	4	700,00	11	6	6	3	3

Хранимая процедура, о которой говорилось выше, может выглядеть так:

```
1.CREATE PROCEDURE paging
2.@n int -- число записей на страницу
3., @p int =1 -- номер страницы, по умолчанию - первая
4.AS
5.SELECT * FROM
6. (SELECT *,
7.    CASE WHEN num % @n = 0 THEN num/@n ELSE num/@n + 1 END
  AS page_num,
8.    CASE WHEN total % @n = 0 THEN total/@n ELSE total/@n
  + 1 END AS num_of_pages
9.  FROM
10.        (SELECT *,
11.              ROW_NUMBER() OVER(ORDER BY price DESC) AS
  num,
12.              COUNT(*) OVER() AS total FROM Laptop
13.        ) X
14.       ) Y
15.     WHERE page_num = @p;
16.     GO
```

Таким образом, если нам нужно получить вторую страницу при условии размещения 2-х записей на странице, достаточно написать

```
1.exec paging @n=2, @p=2
```

В результате получим:

code	model	speed	ram	hd	price	screen	num	total	page_num	num_of_pages
4	1298	600	64	10	1050,00	15	3	6	2	3
2	1321	500	64	8	970,00	12	4	6	2	3

А вот так будет выглядеть неполная вторая страница, если число записей на странице будет равно четырем:

```
1.exec paging @n=4, @p=2
```

code	model	speed	ram	hd	price	screen	num	total	page_num	num_of_pages
6	1298	450	64	10	950,00	12	5	6	2	2
1	1298	350	32	4	700,00	11	6	6	2	2

Новые возможности стандарта, которые были реализованы в **SQL Server** 2012, делают разбивку на страницы очень простой операцией. Речь идет о новых необязательных конструкциях в предложении **ORDER BY**, а именно, **OFFSET** и **FETCH**. С их помощью можно указать сколько строк из результата запроса возвращать (FETCH) клиенту и начиная с какой строки (OFFSET) это делать.

Теперь расширенный синтаксис предложения ORDER BY имеет вид:

```
1.ORDER BY <выражение>
2.      [ ASC | DESC ]
3.      [ ,...n ]
4.[
5.OFFSET <целочисленное_выражение_1> { ROW | ROWS }
6.[FETCH { FIRST | NEXT } <целочисленное_выражение_2> { ROW
  | ROWS } ONLY]
```

```
7.]
```

FIRST и NEXT являются синонимами, как и ROW с ROWS, т.е. можно использовать любой из двух вариантов.

целочисленное_выражение_2 определяет число возвращаемых строк, а *целочисленное_выражение_1* - количество строк от начала отсортированного набора, которое следует пропустить перед выводом. Если предложение FETCH отсутствует, то выводиться будут все строки, начиная с *целочисленное_выражение_1* + 1.

С учетом новых возможностей процедуру постраничного вывода строк, которая была рассмотрена выше, можно реализовать совсем просто:

```
1.CREATE PROC paging
2.  @n int =2 -- число записей на страницу, по умолчанию 2
3., @p int =1 -- номер страницы, по умолчанию - первая
4.AS
5.SELECT * FROM Laptop
6.ORDER BY price DESC OFFSET @n*(@p-1) ROWS FETCH NEXT @n
  ROWS ONLY;
```

Заметим, что стандартный синтаксис предложения ORDER BY поддерживает также PostgreSQL.

Другие примеры использования оконных функций

Использование оконных функций, как и CTE , может сократить объем кода. Вернемся к задаче, на которой мы демонстрировали преимущества использования CTE:

Найти максимальную сумму прихода/расхода среди всех 4-х таблиц базы данных "Вторсырье", а также тип операции, дату и пункт приема, когда и где она была зафиксирована.

Используем следующий алгоритм. К результатам запроса, объединяющему все операции из 4 таблиц базы «Вторсырье», добавим столбец, который с помощью оконной функции **MAX** определит максимальную сумму. Затем мы отберем те строки, у которых сумма операции совпадает с этим максимальным значением:

```
1.SELECT max_sum, type, date, point
2.FROM (
3.SELECT MAX(inc) over() AS max_sum, *
4.FROM (
5.   SELECT inc, 'inc' type, date, point FROM Income
6.   UNION ALL
7.   SELECT inc, 'inc' type, date, point FROM Income_o
8.   UNION ALL
9.   SELECT out, 'out' type, date, point FROM Outcome_o
10.        UNION ALL
11.        SELECT out, 'out' type, date, point FROM Outcome
12.      ) X
13.      ) Y
14.     WHERE inc = max_sum;
```

max_sum	type	date	point
18000,00	inc	2001-10-02 00:00:00.000	3

Рассмотрим еще один пример.

Для каждого ПК из таблицы PC найти разность между его ценой и средней ценой на модели с таким же значением скорости ЦП.

Здесь, в отличие от предыдущих задач, требуется выполнить разбиение компьютеров на группы с одинаковым значением speed, которое мы реализуем с помощью предложения **PARTITION BY**. Именно скорость текущей строки таблицы и будет определять группу для вычисления среднего значения. Решение

```
1. SELECT *, price - AVG(price) OVER(PARTITION BY speed) AS
   dprice
2. FROM PC;
```

Другое решение этой задачи можно построить с помощью коррелирующего подзапроса.

```
1. SELECT *, price - (SELECT AVG(price) FROM PC AS PC1 WHERE
   PC1.speed = PC.speed) AS dprice
2. FROM PC;
```

Функции LAG и LEAD

Синтаксис:

```
1.
   LAG | LEAD (< скалярное выражение > [,< сдвиг >] [, <
   значение по умолчанию >])
2.     OVER ( [ < предложение partition BY >] < предложение
   ORDER BY > )
```

Оконные функции **LAG** и **LEAD** появились в SQL Server, начиная с версии 2012.

Эти функции возвращают значение выражения, вычисленного для предыдущей строки (LAG) или следующей строки (LEAD) результирующего набора соответственно. Рассмотрим простой пример запроса, выводящего коды (code) принтеров вместе с кодами из предыдущей и следующей строк:

```
1.SELECT  code,
2.LAG(code) OVER(ORDER BY code) prev_code,
3.LEAD(code) OVER(ORDER BY code) next_code
4.FROM printer;
```

code	prev_code	next_code
1	NULL	2
2	1	3
3	2	4
4	3	5
5	4	6
6	5	NULL

Обратите внимание, что если следующей или предыдущей строки (в порядке возрастания значения *code*) не существует, то используется NULL-значение. Однако такое поведение можно поменять с помощью необязательного (третьего) параметра каждой функции. Значение этого параметра будет использоваться в том случае, если соответствующей строки не существует. В нижеследующем примере используется значение -999, если предыдущей строки не существует, и 999, если не существует следующей строки.

```
1.SELECT  code,
2.LAG(code,1,-999) OVER(ORDER BY code) prev_code,
3.LEAD(code,1,999) OVER(ORDER BY code)  next_code
4.FROM printer;
```

code	prev_code	next_code
1	-999	2
2	1	3
3	2	4
4	3	5
5	4	6
6	5	999

Чтобы указать третий параметр, нам пришлось использовать и второй необязательный параметр с значением 1, которое принимается по умолчанию. Этот параметр определяет, какую из предыдущих (последующих) строк следует использовать, т.е. на сколько данная строка отстоит от текущей. В следующем примере берется строка, идущая через одну от текущей.

```
1.SELECT  code,
2.LAG(code,2,-999) OVER(ORDER BY code) prev_code,
3.LEAD(code,2,999) OVER(ORDER BY code) next_code
4.FROM printer;
```

code	prev_code	next_code
1	-999	3
2	-999	4
3	1	5
4	2	6
5	3	999

6	4	999

В заключение отметим, что порядок, в котором выбираются следующие и предыдущие строки задаётся предложением **ORDER BY** в предложении **OVER**, а не сортировкой, используемой в запросе. Вот пример, который иллюстрирует сказанное.

```
1.SELECT  code,
2.LAG(code) OVER(ORDER BY code) prev_code,
3.LEAD(code) OVER(ORDER BY code) next_code
4.FROM printer
5.ORDER BY code DESC;
```

code	prev_code	next_code
6	5	NULL
5	4	6
4	3	5
3	2	4
2	1	3
1	NULL	2

Чтобы оценить преимущество, которое предоставляет появление в языке SQL данных функций, рассмотрим "классические" решения данной задачи.

Самосоединение

```
1.SELECT p1.code,p3.code,p2.code
```

```
2. FROM printer p1 LEFT JOIN Printer p2 ON p1.code=p2.code-
   1
3. LEFT JOIN Printer p3 ON p1.code=p3.code+1;
```

Коррелирующий подзапрос

```
1. SELECT p1.code,
2. (SELECT MAX(p3.code) FROM Printer p3 WHERE p3.code <
   p1.code) prev_code,
3. (SELECT MIN(p2.code) FROM Printer p2 WHERE p2.code >
   p1.code) next_code
4. FROM printer p1;
```

Функции FIRST_VALUE и LAST_VALUE

Для каждой компании выводить один рейс, выбираемый случайным образом

(база данных аэропорт).

Использование коррелирующего подзапроса

В подзапросе для каждой компании данные сортируются случайным образом при использовании функции newid(), после чего выбирается одна (первая строка) этого отсортированного набора:

```
1. SELECT id_comp,
2.   (SELECT TOP 1 trip_no FROM trip t WHERE c.id_comp =
   t.id_comp ORDER BY NEWID()) trip_no
3. FROM company c
4. ORDER BY id_comp;
```

id_comp	trip_no
1	1188
2	1146
3	1124
4	1101
5	7771

Разумеется, вы скорее всего получите другой результат, но, поскольку данных в таблице немного, рано или поздно вы сможете получить и такой. :-)

Использование функции FIRST_VALUE

Эта оконная функция возвращает первое из упорядоченного набора значений. Теперь мы можем сделать все без подзапросов, выделив в окне набор рейсов для компании из текущей строки запроса с помощью предложения PARTITION BY и упорядочив его, как и в предыдущем примере, случайным образом в предложении ORDER BY:

```
1.SELECT DISTINCT id_comp,
2.FIRST_VALUE(trip_no) OVER(PARTITION BY id_comp ORDER BY
  NEWID()) trip_no
3.FROM trip
4.ORDER BY id_comp;
```

id_comp	trip_no
1	1195
2	1145
3	1124
4	1100

5	8882

Ключевое слово DISTINCT нужно здесь для того, чтобы не повторять одну и ту же компанию для каждого выполняемого ею рейса.

Засада с LAST_VALUE

Казалось бы, какая разница брать первое или последнее значение из случайным образом упорядоченного набора? Но давайте посмотрим, что мы получим, если в предыдущем запросе заменить FIRST_VALUE на LAST_VALUE:

```
1.SELECT DISTINCT id_comp,
2.LAST_VALUE(trip_no) OVER(PARTITION BY id_comp ORDER BY
  NEWID()) trip_no
3.FROM trip
4.ORDER BY id_comp;
```

Я приведу результаты только для id_comp = 1. Вы можете сами выполнить запрос, чтобы убедиться, что будут выводиться абсолютно все рейсы из таблицы Trip.

id_comp	trip_no
1	1181
1	1182
1	1187
1	1188
1	1195
1	1196

Что мы делаем в подобных случаях? Конечно, обращаемся к документации, а там мы читаем... Нет, постойте, сначала полный синтаксис:

```
1.LAST_VALUE | FIRST_VALUE ( [ скалярное_выражение ] ) [ IGNORE NULLS | RESPECT NULLS ]
2.      OVER    (    [    предложение_partition_by    ] предложение_order_by [ предложение_rows_range ] )
```

Здесь
IGNORE NULLS или RESPECT NULLS определяют, будут ли учитываться NULL-значения;
предложение_rows_range задает параметры окна.

А теперь читаем:

Внимание:

Диапазоном по умолчанию является RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

Т.е. окном является диапазон от текущей строки и неограниченно выше. Поскольку мы выбираем последнюю строку диапазона, то всегда будет выводиться текущая строка, как бы не сортировались строки. Т.е. сколько бы строк выше не оказалось при случайной сортировке.

Потому и DISTINCT не помогает, т.к. все выводимые строки оказываются уникальными.

Значит нам просто нужно явно (и правильно!) задать параметры окна, а именно, от текущей строки и неограниченно ниже, поскольку мы выбираем последнее значение:

```
1.SELECT DISTINCT id_comp,
2.LAST_VALUE(trip_no) OVER(PARTITION BY id_comp ORDER BY NEWID()
3.RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) trip_no
4.FROM trip
5.ORDER BY id_comp;
```

id_comp	trip_no
1	1188
2	1145
3	1123
4	1101
5	7773

Остался последний вопрос. Если мы не задавали параметры окна, почему у нас правильно отработал запрос с FIRST_VALUE?

Ответ лежит на поверхности - потому что здесь значение по умолчанию нам подошло, хотя я и не нашел в документации, каким оно должно быть для FIRST_VALUE .

Могу предположить, что тем же, что и для LAST_VALUE.

COUNT DISTINCT и оконные функции

Мы без проблем можем посчитать общее количество ПК для каждого производителя, а также количество уникальных моделей данного производителя в таблице PC:

```
1. SELECT maker, COUNT(*) models, COUNT(DISTINCT pc.model)
   unique_models
2. FROM product p JOIN pc ON p.model=pc.model
3. GROUP BY maker
4. ORDER BY maker;
```

maker	models	unique_models
A	8	2
B	3	1
E	1	1

Если нам требуется получить детальную информацию о каждой модели, наряду с их общим количеством для каждого производителя, то можно использовать оконную функцию:

```
1.SELECT maker, pc.model,pc.price,
2.COUNT(*) over(partition BY maker) models
3.FROM product p JOIN pc ON p.model=pc.model
4.ORDER BY maker, pc.model;
```

maker	model	price	models
A	1232	600,00	8
A	1232	400,00	8
A	1232	350,00	8
A	1232	350,00	8
A	1233	600,00	8
A	1233	950,00	8
A	1233	980,00	8
A	1233	970,00	8
B	1121	850,00	3
B	1121	850,00	3

B	1121	850,00	3
E	1260	350,00	1

Теперь представим, что нам требуется дополнить эту информацию количеством уникальных моделей. Естественная попытка

```
1.SELECT maker, pc.model,pc.price,
2.COUNT(*) over(partition BY maker) models,
3.COUNT(DISTINCT  pc.model)  over(partition  BY  maker)
  unique_models
4.FROM product p JOIN pc ON p.model=pc.model
5.ORDER BY maker, pc.model;
```

терпит неудачу:

Использование ключевого слова DISTINCT не допускается с предложением OVER.

Сообщение об ошибке ясно описывает проблему. Вопрос в том, как её обойти.

Использование подзапроса

```
1.WITH cte AS
2.(SELECT maker, pc.model,pc.price,
3.COUNT(*) over(partition BY maker) models
4. FROM product p JOIN pc ON p.model=pc.model)
5.SELECT maker, model, models,
6.(SELECT COUNT(DISTINCT model)
7.     FROM cte t WHERE t.maker=cte.maker) unique_models
8.FROM cte
9.ORDER BY maker,model;
```

maker	model	models	unique_models
A	1232	8	2
A	1232	8	2

A	1232	8	2
A	1232	8	2
A	1233	8	2
A	1233	8	2
A	1233	8	2
A	1233	8	2
B	1121	3	1
B	1121	3	1
B	1121	3	1
E	1260	1	1

Использование DENSE_RANK

```
1.WITH cte AS
2.(SELECT maker, pc.model,pc.price,
3.COUNT(*) over(partition BY maker) models,
4.DENSE_RANK()  over(partition BY maker ORDER BY pc.model)
  drnk
5.FROM product p JOIN pc ON p.model=pc.model)
6.SELECT maker, model, price, models,
7.MAX(drnk) over(partition BY maker) unique_models FROM cte
8.ORDER BY maker, model;
```

Здесь мы воспользовались тем фактом, что последнее ранговое значение - max(drnk) - оказывается равным числу уникальных моделей.

CROSS APPLY / OUTER APPLY

Оператор **CROSS APPLY** появился в SQL Server 2005. Он позволяет выполнить соединение двух табличных выражений. При этом каждая строка из левой таблицы сочетается с каждой строкой из правой.

Давайте попробуем разобраться в том, какие преимущества дает нам использование этого нестандартного оператора.

Первый пример.

```
1.SELECT * FROM
2.Product
3.CROSS APPLY
4.Laptop;
```

Мы получили просто декартово произведение таблиц Product и Laptop. Аналогичный результат мы можем получить с помощью следующих стандартных запросов:

```
1.SELECT * FROM
2.Product
3.CROSS JOIN
4.Laptop;
```

Или

```
1.SELECT * FROM
2.Product, Laptop;
```

Поставим теперь более осмысленную задачу.

Для каждого ноутбука дополнительно вывести имя производителя.

Эту задачу мы можем решить с помощью обычного соединения:

```
1. SELECT P.maker, L.*  FROM
2. Product P JOIN Laptop L ON P.model= L.model;
```

С помощью CROSS APPLY решение этой же задачи можно написать так:

```
1. SELECT P.maker, L.*  FROM
2. Product P
3. CROSS APPLY
4. (SELECT * FROM Laptop L WHERE P.model= L.model) L;
```

"И что тут нового"? - спросите вы. Запрос стал даже более громоздким. Пока да, можно согласиться. Но уже здесь можно заметить весьма важную вещь, которая отличает CROSS APPLY от других видов соединений. А именно, мы используем коррелирующий подзапрос в предложении FROM, передавая в него значения из левого табличного выражения. В данном примере это значения из столбца P.model. Т.е. для каждой строки из левой таблицы правая таблица будет своя.

Поняв это, мы можем воспользоваться данными преимуществами. Рассмотрим следующую задачу.

Для каждого ноутбука дополнительно вывести максимальную цену среди ноутбуков того же производителя.

Эту задачу мы можем решить с помощью коррелирующего подзапроса в предложении SELECT:

```
1. SELECT *, (SELECT MAX(price) FROM Laptop L2
2. JOIN  Product P1 ON L2.model=P1.model
```

```
3.WHERE maker = (SELECT maker FROM Product P2 WHERE
  P2.model= L1.model)) max_price
4. FROM laptop L1;
```

Пока решение, использующее CROSS APPLY, будет мало чем отличаться от вышеприведенного:

```
1.SELECT *
2. FROM laptop L1
3. CROSS APPLY
4. (SELECT MAX(price) max_price FROM Laptop L2
5.JOIN  Product P1 ON L2.model=P1.model
6.WHERE maker = (SELECT maker FROM Product P2 WHERE
  P2.model= L1.model)) X;
```

А теперь представьте, что нам нужно, помимо максимальной цены, вывести минимальную, среднюю цены и т.д. Поскольку коррелирующий подзапрос в предложении SELECT должен возвращать только одно значение, в первом варианте решения нам придется фактически дублировать код для каждого агрегата:

```
1.SELECT *, (SELECT MAX(price) FROM Laptop L2
2.JOIN  Product P1 ON L2.model=P1.model
3.WHERE maker = (SELECT maker FROM Product P2 WHERE
  P2.model= L1.model)) max_price,
4.(SELECT MIN(price) FROM Laptop L2
5.JOIN  Product P1 ON L2.model=P1.model
6.WHERE maker = (SELECT maker FROM Product P2 WHERE
  P2.model= L1.model)) min_price
7.FROM Laptop L1;
```

и т.д.

А при использовании CROSS APPLY мы просто добавим в подзапрос требуемую агрегатную функцию:

```
1.SELECT *
```

```
2. FROM laptop L1
3. CROSS APPLY
4. (SELECT MAX(price) max_price, MIN(price) min_price  FROM
   Laptop L2
5.JOIN  Product P1 ON L2.model=P1.model
6.WHERE  maker  =  (SELECT  maker  FROM  Product  P2  WHERE
   P2.model= L1.model)) X;
```

Рассмотрим еще один пример.

Соединить каждую строку из таблицы Laptop со следующей строкой в порядке, заданном сортировкой (model, code).

Столбец *code* в сортировке используется для того, чтобы задать однозначный порядок для строк, имеющих одинаковые значения в столбце *model*. С помощью CROSS APPLY мы можем передать в подзапрос параметры текущей строки и выбрать первую строку из тех, которые идут ниже текущей в заданном сортировкой порядке. Итак,

```
1.SELECT * FROM laptop L1
2.CROSS APPLY
3.(SELECT TOP 1 * FROM Laptop L2
4.WHERE L1.model < L2.model OR (L1.model = L2.model AND
   L1.code < L2.code)
5.ORDER BY model, code) X
6.ORDER BY L1.model;
```

Попробуйте решить эту задачу традиционными средствами, чтобы сравнить трудозатраты.

Оператор OUTER APPLY

Как показывают результаты предыдущего запроса, мы "потеряли" последнюю (шестую) строку из таблицы Laptop, поскольку ее не с чем соединять. Другими словами, CROSS APPLY ведет себя как внутренне соединение. Аналогом же внешнего (левого) соединения является оператор OUTER APPLY. Он отличается от CROSS APPLY только тем, что выводит все строки из левой таблицы, заменяя отсутствующие значения из правой таблицы NULL-значениями.

Замена CROSS APPLY на OUTER APPLY в предыдущем запросе иллюстрирует сказанное.

```
1. SELECT * FROM laptop L1
2. OUTER APPLY
3. (SELECT TOP 1 *
4. FROM Laptop L2
5. WHERE L1.model < L2.model OR (L1.model = L2.model AND
   L1.code < L2.code)
6. ORDER BY model, code) X
7. ORDER BY L1.model;
```

Еще одной популярной задачей является вывод по N строк из каждой группы. Примером может служить вывод 5 наиболее популярных товаров в каждой категории. Рассмотрим следующую задачу.

Вывести из таблицы Product по три модели с наименьшими номерами из каждой группы, характеризуемой типом продукции.

Дополним решения, предложенные на сайте sql-ex.ru, решением, использующим CROSS APPLY. Идея заключается в соединении уникальных типов (первый запрос) с запросом, выводящих по 3 модели модели каждого типа из первого запроса в соответствии с требуемой сортировкой.

```
1. SELECT X.* FROM
2. (SELECT DISTINCT type FROM product) Pr1
3. CROSS APPLY
4. (SELECT TOP 3 * FROM product Pr2 WHERE  Pr1.type=Pr2.type
   ORDER BY pr2.model) x;
```

В заключение давайте рассмотрим пример задачи, которая часто встречается на практике, а именно, задачи расположения в столбец значений из строки таблицы. Для конкретизации сформулируем задачу таким образом.

Для таблицы Laptop представить информацию о продуктах в три столбца: code, название характеристики (speed, ram, hd или screen), значение характеристики.

Метод решения состоит в использовании конструктора таблицы, куда с помощью CROSS APPLY будут передаваться значения столбцов. Давайте разберем этот метод подробно.

Конструктор таблицы может использоваться не только в операторе INSERT, но и для задания таблицы в предложении FROM, например,

```
1.SELECT name, value
2.FROM (
3.VALUES('speed', 1)
4.,('ram', 1)
5.,('hd', 1)
6.,('screen', 1)
7.) Spec(name, value);
```

Эта таблица у нас называется Spec и содержит два столбца - *name* (символьные строки) и *value* (числа).

Давайте теперь включим эту таблицу в оператор CROSS APPLY, который будет соединять каждую строку из таблицы Laptop с четырьмя строками из сгенерированной таблицы:

```
1.SELECT code, name, value
2.FROM Laptop
3.CROSS APPLY (
4.VALUES('speed', 1)
5.,('ram', 1)
6.,('hd', 1)
7.,('screen', 1)
8.) Spec(name, value)
9.WHERE code < 4 -- для уменьшения размера выборки
10.      ;
```

Собственно, нам осталось воспользоваться основным свойством оператора CROSS APPLY - коррелированностью табличного выражения - и заменить единички в столбце *value* на имена столбцов из соединяемой таблицы:

```
1.SELECT code, name, value FROM Laptop
2.CROSS APPLY
3.(VALUES('speed', speed)
4.,('ram', ram)
5.,('hd', hd)
6.,('screen', screen)
7.) spec(name, value)
8.WHERE code < 4 -- для уменьшения размера выборки
9.ORDER BY code, name, value;
```

PostgreSQL обладает аналогичной функциональностью. Синтаксические отличия незначительны и состоят в замене CROSS APPLY на CROSS JOIN LATERAL. Сравните три примера, которые рассматривались на предыдущих страницах этой главы.

Пример 1

SQL Server

```
1.SELECT *
2.     FROM laptop L1
3.     CROSS APPLY
4.         (SELECT    MAX(price)    max_price,    MIN(price)
  min_price  FROM Laptop L2
5.    JOIN  Product P1 ON L2.model=P1.model
6.     WHERE maker = (SELECT maker FROM Product P2 WHERE
  P2.model= L1.model)) X;
```

PostgreSQL

```
1.SELECT *
2.     FROM laptop L1
3.     CROSS JOIN LATERAL
4.         (SELECT    MAX(price)    max_price,    MIN(price)
  min_price  FROM Laptop L2
5.    JOIN  Product P1 ON L2.model=P1.model
6.     WHERE maker = (SELECT maker FROM Product P2 WHERE
  P2.model= L1.model)) X;
```

Пример 2

SQL Server

```
1.
  SELECT code, name, value FROM Laptop
2.     CROSS APPLY
3.     (VALUES('speed', speed)
4.     ,('ram', ram)
5.     ,('hd', hd)
6.     ,('screen', screen)
7.     ) spec(name, value)
8.     WHERE code < 4
9.     ORDER BY code, name, value;
```

PostgreSQL

```
1.SELECT code, name, value FROM Laptop
2.     CROSS JOIN LATERAL
3.     (VALUES('speed', speed)
4.     ,('ram', ram)
5.     ,('hd', hd)
6.     ,('screen', screen)
7.     ) spec(name, value)
8.     WHERE code < 4
9.     ORDER BY code, name, value;
```

Пример 3

SQL Server

```
1.SELECT * FROM laptop L1
2.     CROSS APPLY
3.     (SELECT TOP 1 * FROM Laptop L2
4.     WHERE L1.model < L2.model OR (L1.model = L2.model AND
  L1.code < L2.code)
5.     ORDER BY model, code) X
6.     ORDER BY L1.model;
```

PostgreSQL

Дополнительное отличие в этом примере связано не с реализацией CROSS APPLY, а с тем, что для ограничения выборки PostgreSQL вместо конструкции TOP(n) использует LIMIT n в предложении ORDER BY.

```
1. SELECT * FROM laptop L1
2.     CROSS JOIN LATERAL
3.     (SELECT  * FROM Laptop L2
4.     WHERE L1.model < L2.model OR (L1.model = L2.model AND
   L1.code < L2.code)
5.     ORDER BY model, code LIMIT 1) X
6.     ORDER BY L1.model;
```

OUTER APPLY

Для данного "внешнего" соединения в PostgreSQL используется LEFT JOIN LATERAL. Сравните запросы в примере 4.

Пример 4

SQL Server

```
1. SELECT * FROM laptop L1
2.     OUTER APPLY
3.     (SELECT TOP 1 *
4.     FROM Laptop L2
5.     WHERE L1.model < L2.model OR (L1.model = L2.model
   AND L1.code < L2.code)
6.     ORDER BY model, code) X
7.     ORDER BY L1.model;
```

PostgreSQL

Обратите внимание на предикат ON TRUE. Поскольку синтаксис соединения [LEFT|RIGHT [OUTER]] JOIN требует предиката, то для единообразия используется "фиктивный" предикат, имеющий значение ИСТИНА.

```
1. SELECT * FROM laptop L1
2.     LEFT JOIN LATERAL
3.     (SELECT *
4.     FROM Laptop L2
5.     WHERE L1.model < L2.model OR (L1.model = L2.model
   AND L1.code < L2.code)
6.     ORDER BY model, code LIMIT 1) X ON TRUE
7.     ORDER BY L1.model;
```

Функция CONCAT

Для **конкатенации строк** в SQL Server используется оператор "+".

Т.е. если операнды являются числовыми, то выполняется операция сложения, а если – строковыми, то конкатенация:

```
1.SELECT 1+2+3+4 a, '1'+'2'+'3'+'4' b;
```

a	b
10	1234

Если же операнды являются значениями разных типов, то SQL Server выполняет **неявное преобразование типов**. Выполняя следующий запрос

```
1.SELECT hd + ' Gb' volume FROM PC WHERE model = 1232;
```

мы получим сообщение об ошибке:

Error converting data type varchar to real.(Ошибка при преобразовании типа данных varchar к real.)

Существует приоритет типов при их неявном преобразовании, и в соответствии с этим приоритетом сервер пытается преобразовать строку ' Gb' к типу данных столбца *hd* (real).

Разумеется, явное преобразование типа решает проблему:

```
1.SELECT CAST(hd AS VARCHAR) + ' Gb' volume FROM PC WHERE
  model = 1232;
```

volume
5 Gb

10 Gb
8 Gb
10 Gb

В **SQL Server 2012** появилась функция **CONCAT**, которая выполняет конкатенацию, неявно преобразуя типы аргументов к строковому типу данных. С помощью этой функции предыдущий запрос можно переписать так:

```
1. SELECT CONCAT(hd, ' Gb') volume FROM PC WHERE model=1232;
```

Еще одна полезная особенность функции CONCAT состоит в том, что **NULL-значения** неявно преобразуются в пустую строку- ''. Обычная же конкатенация с NULL-значением дает NULL. Вот пример, который это демонстрирует.

```
1. SELECT NULL + 'concatenation with NULL' plus,
2.                CONCAT(NULL, 'concatenation with NULL')
   concat;
```

plus	concat
NULL	concatenation with NULL

Следует отметить, что у функции CONCAT может быть произвольное число аргументов, но не менее двух.

```
1. SELECT 1+2+3+4 a, CONCAT(1, 2, 3, 4) b;
```

a	b
10	1234

MySQL

В MySQL также имеется функция CONCAT, вернее, даже две функции. Первая из них – CONCAT – возвращает NULL, если среди аргументов функции встречается NULL, вторая – CONCAT_WS – опускает аргумент, если его значение NULL. Кроме того, эта функция первым аргументом имеет разделитель, используемый при конкатенации.

```
1.SELECT CONCAT(NULL, 'concatenation with NULL') concat,
2. CONCAT_WS('',NULL, 'concatenation with NULL') concat_ws,
3.CONCAT_WS(', ',1, 2, NULL, 4) concat_ws_null;
```

concat	concat_ws	concat_ws_null
(NULL)	concatenation with NULL	1, 2, 4

Операторы PIVOT и UNPIVOT

Чтобы объяснить, что такое PIVOT, я бы начал с электронных таблиц EXCEL. В версии MS Excel 5.0 появились так называемые сводные таблицы. Сводные таблицы представляют собой двумерную визуализацию многомерных структур данных, применяемых в технологии OLAP для построения хранилищ данных. Правильней даже сказать двумерные сечения трехмерных OLAP-кубов, если иметь в виду наличие на сводной таблице элемента, который называется «страница». Сводные таблицы позволяют выполнять стандартные операции с многомерными структурами, например, упоминавшееся уже сечение куба, свертку и детализацию – операцию обратную свертке.

Следует сказать, что сводная таблица не является реляционной, поскольку имеет не только заголовки столбцов, но и заголовки строк, при этом и те и другие

формируются из данных, находящихся в столбцах обычных реляционных таблиц. Последнее, кстати, означает, что число строк и столбцов заранее неизвестно, т.к. они формируются динамически при выполнении запроса к реляционным данным. Кроме того, заголовки могут иметь многоуровневые подзаголовки, что и позволяет выполнять операции свертки (переход на более высокий уровень иерархии) и детализации (переход на более низкий уровень иерархии).

Такие свойства сводных таблиц позволяют их использовать, наряду со сводными диаграммами, в качестве клиента для визуального отображения многомерных данных, находящихся в хранилищах, поддерживаемых различными СУБД (например, MS SQL Server Analysis Services).

Чтобы пояснить сказанное примером, давайте рассмотрим такой запрос к одной из учебных баз на sql-ex.ru:

```
1. SELECT maker, type
2. FROM product;
```

результатом которого является такая таблица:

maker	type
B	PC
A	PC
A	PC
E	PC
A	Printer
D	Printer

A	Laptop
C	Laptop
A	Printer
A	Printer
D	Printer
E	Printer
B	Laptop
A	Laptop
E	PC
E	PC

Пусть теперь нам требуется получить таблицу со следующими заголовками:

	Типы продукции			
Производители		Laptop	PC	Printer
	A			
	B			
	C			
	D			
	E			

Заголовками строк здесь являются уникальные имена производителей, которые берутся из столбца maker вышеприведенного запроса, а заголовками столбцов – уникальные типы продукции (соответственно, из столбца type). А что должно быть в середине? Ответ очевиден – некоторый агрегат, например, функция count(type), которая подсчитает для каждого производителя отдельно число моделей ПК, ноутбуков и принтеров, которые и заполнят соответствующие ячейки этой таблицы.

Это простейший вариант сводной таблицы, который имеет всего два уровня иерархии по столбцам и строкам. Т.е. выполняя свертку по вертикали, мы получаем количество моделей каждого вида продукции для всех производителей, а по горизонтали – общее число моделей независимо от типа для каждого производителя. Можно было бы добавить дополнительные уровни иерархии, например, для градации принтеров по цветности, а ноутбуков по размеру экрана и т.д.

Можно сказать, что pivot-таблица в SQL – это одноуровневая сводная таблица.

Оператор PIVOT не является стандартным (я не уверен, что он когда-нибудь будет стандартизован ввиду нереляционной природы pivot-таблицы), поэтому я буду использовать в примерах его реализацию в языке T-SQL (SQL Server 2005/2008).

Я могу и ошибиться в хронологии, но мне представляется, что успех реализации сводной таблицы в Excel привел к появлению так называемых перекрестных запросов в Access, и, наконец, к оператору PIVOT в T-SQL.

Особенности: ORACLE

В Oracle операции PIVOT и DRILLDOWN (UNPIVOT) были отданы на откуп средствам визуализации OLAP, т.е. MS Excel, BusinessObjects и пр.

В Oracle 11g появились pivot/unpivot, но в pivot есть "тонкость" - нужно ЯВНО указывать столбцы, если результат нужен в ТАБЛИЧНОМ виде. Получение данных для столбцов, не указанных явно, также возможно, но только в виде XML.

Оператор PIVOT

Давайте рассмотрим такую задачу.

Пример 1.

Для каждого производителя из таблицы Product определить число моделей каждого типа продукции.

Задачу можно решить стандартными средствами с использованием оператора CASE:

```
1.SELECT maker,
2.SUM(CASE type WHEN 'pc' THEN 1 ELSE 0 END) PC
3., SUM(CASE type WHEN 'laptop' THEN 1 ELSE 0 END) Laptop
4., SUM(CASE type WHEN 'printer' THEN 1 ELSE 0 END) Printer
5.FROM Product
6.GROUP BY maker;
```

Теперь решение через PIVOT:

```
1.SELECT maker, -- столбец (столбцы), значения из которого
  формируют заголовки строк
2.[pc], [laptop], [printer] -- значения из столбца, который
  указан в предложении type,
3.-- формирующие заголовки столбцов
4.FROM Product -- здесь может быть подзапрос
5.PIVOT -- формирование пивот-таблицы
6.(COUNT(model) -- агрегатная функция, формирующая
  содержимое сводной таблицы
7.FOR type -- указывается столбец,
8.-- уникальные значения в котором будут являться
  заголовками столбцов
9.IN([pc], [laptop], [printer]) --указываются конкретные
  значения в столбце type,
10.      -- которые следует использовать в качестве
  заголовков,
11.      -- т.к. нам могут потребоваться не все
12.     ) pvt ;-- алиас для сводной таблицы
```

Надеюсь, что комментарии к коду достаточно понятны для того, чтобы написать оператор PIVOT без шпаргалки. Давайте попробуем.

Пример 2.

Посчитать среднюю цену на ноутбуки в зависимости от размера экрана.

Задача элементарная и решается с помощью группировки:

```
1.SELECT screen, AVG(price) avg_
2.FROM Laptop
3.GROUP BY screen;
```

screen	avg_
11	700.00
12	960.00
14	1175.00
15	1050.00

А вот как можно повернуть эту таблицу с помощью PIVOT:

```
1.SELECT [avg_],
2. [11],[12],[14],[15]
3. FROM (SELECT 'average price' AS 'avg_', screen, price
  FROM Laptop) x
4. PIVOT
5. (AVG(price)
6. FOR screen
7. IN([11],[12],[14],[15])
8. ) pvt;
```

avg_	11	12	14	15
average price	700.00	960.00	1175.00	1050.00

В отличие от сводных таблиц, в операторе PIVOT требуется явно перечислить столбцы для вывода. Это серьезное ограничение, т.к. для этого нужно знать характер данных, а значит и применять в приложениях этот оператор мы сможем, как правило, только к справочникам (вернее, к данным, которые берутся из справочников).

Если рассмотренных примеров покажется недостаточно, чтобы понять и использовать без затруднений этот оператор, я вернусь к нему, когда придумаю нетривиальные примеры, где использование оператора PIVOT позволяет существенно упростить код.

Я написал этот опус в помощь тем, кому оператор PIVOT интуитивно непонятен. Могу согласиться с тем, что в реляционном языке SQL он выглядит инородным телом. Собственно, иначе и быть не может ввиду того, что поворот (транспонирование) таблицы является не реляционной операцией, а операцией работы с многомерными структурами данных.

Оператор UNPIVOT

Как следует из названия оператора, UNPIVOT выполняет обратную по отношению к PIVOT операцию, т.е. представляет данные, записанные в строке таблицы, в одном столбце. В примере, рассмотренном в предыдущем параграфе, мы с помощью оператора PIVOT разворачивали в строку таблицу, полученную с помощью следующего запроса:

```
1.SELECT screen, AVG(price) avg_
2.FROM Laptop
3.GROUP BY screen;
```

screen	avg_
11	700.00
12	960.00
14	1175.00

15	1050.00

В результате было получено следующее представление:

avg_	11	12	14	15
average price	700.00	960.00	1175.00	1050.00

Исходный результат мы можем получить, если применим к pivot-запросу unpivot-преобразование:

```
1.SELECT screen -- заголовок столбца, который будет
  содержать заголовки
2.-- строк исходной таблицы
3.,avg__ AS avg_
4.-- заголовок столбца, который будет содержать значения из
  строки исходной таблицы
5.FROM( -- pivot-запрос из предыдущего примера
6.SELECT [avg_], [11],[12],[14],[15]
7.FROM (SELECT 'average price' AS 'avg_', screen, price
8.FROM Laptop) x
9. PIVOT (AVG(price) FOR screen IN([11],[12],[14],[15]) )
10.      pvt
11.      -- конец pivot-запроса
12.      ) pvt
13.      UNPIVOT (avg__ -- заголовок столбца, который будет
  содержать значения
14.      -- из столбцов исходной таблицы, перечисленных ниже
15.       FOR screen IN([11],[12],[14],[15])
16.      ) unpvt;
```

Заметим, что имя avg_ нельзя использовать в операторе UNPIVOT, поскольку оно уже использовалось в операторе PIVOT, поэтому я использовал новое имя avg__, которому затем присвоил алиас, чтобы полностью воссоздать результат, полученный с помощью группировки.

Рассмотрим теперь более содержательный пример. Пусть требуется информацию о рейсе 1100 представить в следующем виде:

trip_no	spec	info
1100	id_comp	4
1100	plane	Boeing
1100	town_from	Rostov
1100	town_to	Paris
1100	time_out	14:30:00
1100	time_in	17:50:00

Поскольку информация из строки таблицы трансформируется в столбец, то напрашивается использование оператора UNPIVOT. Здесь следует сделать одно замечание. Значения в этом столбце должны быть одного типа. Поскольку в этот столбец в нашем примере собираются значения из разных столбцов исходной таблицы, то нужно преобразовать их к единому типу. Более того, должны совпадать не только типы, но и размер.

Общим типом в нашем случае является строковый тип. Поскольку столбцы town_from и town_to уже имеют тип char(25), то приведем все к этому типу:

```
1.SELECT trip_no,CAST(id_comp AS CHAR(25)) id_comp,
2. CAST(plane AS CHAR(25)) plane,town_from,town_to,
3. CONVERT(CHAR(25),time_out, 108) time_out,
4.CONVERT(CHAR(25),time_in,108) time_in
5.FROM Trip WHERE trip_no =1100;
```

trip_no	id_comp	plane	town_from	town_to	time_out	time_in
1100	4	Boeing	Rostov	Paris	14:30:00	17:50:00

Здесь мы заодно преобразовали время вылета/прилета, убрав из него составляющую даты:

```
1.CONVERT(CHAR(25),time_out, 108)
```

Остальное, я надеюсь, понятно из кода:

```
1.SELECT trip_no, spec, info FROM (
2.SELECT trip_no,CAST(id_comp AS CHAR(25)) id_comp,
3.CAST(plane AS CHAR(25)) plane,
4.CAST(town_from AS CHAR(25)) town_from,
5.CAST(town_to AS CHAR(25)) town_to,
6.CONVERT(CHAR(25),time_out, 108) time_out,
7. CONVERT(CHAR(25),time_in,108) time_in
8.FROM Trip
9.WHERE trip_no =1100 ) x
10.      UNPIVOT( info
11.      FOR spec IN(id_comp, plane, town_from, town_to,
  time_out, time_in)
12.      ) unpvt;
```

Столбец с именем spec используется для вывода названий параметров, а столбец info содержит сами параметры. Результат выполнения запроса уже был представлен в условии задачи.

Пусть нам требуется повернуть строку, содержащую NULL-значение в одном из столбцов.

```
1.WITH utest AS
2.(SELECT 1 a, 2 b, NULL c)
3.SELECT * FROM utest;
```

Т.е. вместо результата

a	b	c
1	2	NULL

мы хотим получить

a	1
b	2
c	NULL

Применим оператор **UNPIVOT**:

```
1.WITH utest AS
2.(SELECT 1 a, 2 b, NULL c)
3.SELECT col, value FROM utest
4.UNPIVOT (
5.value FOR col IN (a,b,c)
6.) AS unpvt;
```

Первая неожиданность - ошибка компиляции:

Тип столбца "c" конфликтует с типами других столбцов, указанных в списке UNPIVOT.

Это означает, что сервер неявно не преобразовал тип столбца "c", содержащий NULL, к типу первых двух столбцов, которые могут быть оценены как целочисленные.

Давайте сделаем это явно:

```
1.WITH utest AS
2.(SELECT 1 a, 2 b, CAST(NULL AS INT) c)
3.SELECT col,value FROM utest
4.UNPIVOT (
5.value FOR col IN (a,b,c)
6.) AS unpvt;
```

col	value
a	1
b	2

Вторая неожиданность - оказывается UNPIVOT игнорирует NULL-значения, не выводя их в результирующем наборе.

Первое, что приходит в голову всем, это заменить NULL каким-нибудь валидным значением, заведомо отсутствующим в столбце. Если, в соответствии с ограничениями предметной области, значения c не могут быть отрицательными, заменим NULL на -1:

```
1.WITH utest AS
2.(SELECT 1 a, 2 b,  COALESCE(CAST(NULL AS INT),-1) c)
3.SELECT col, value FROM utest
4.UNPIVOT (
5.value FOR col IN (a, b, c)
6.) AS unpvt;
```

col	value
a	1
b	2

c	-1

Остался последний шаг, о котором многие забывают, решая задачи на сайте sql-ex.ru. А именно, обратное преобразование. Вместо этого пытаются подобрать такое значение, которое позволило бы "удовлетворить" систему проверки. Иногда это получается, например, если сравнение NULL и '' (пустой строки) оценивается на сайте как true. Но понятно, что на это полагаться не стоит. Итак, обратное пребразование:

```
1.WITH utest AS
2.(SELECT 1 a, 2 b,  COALESCE(CAST(NULL AS INT),-1) c)
3.SELECT col, NULLIF(value, -1) value FROM utest
4.UNPIVOT (
5.value FOR col IN (a, b, c)
6.) AS unpvt;
```

col	value
a	1
b	2
c	NULL

Здесь как нельзя более кстати пришлась функция NULLIF.

CROSSTAB в PostgreSQL

Повернуть таблицу в PostgreSQL можно при помощи функции **CROSSTAB**. Эта функция принимает в качестве текстового параметра SQL-запрос, который возвращает 3 столбца:

- идентификатор строки - т.е. этот столбец содержит значения, определяющие результирующую (повернутую) строку;
- категорию - уникальные значения из этого столбца образуют столбцы повернутой таблицы. Нужно отметить, что в отличие от PIVOT сами значения роли не играют; важно лишь их количество, которое определяет максимально допустимое количество столбцов;
- значение категории - собственно значения категорий. Размещение значений по столбцам производится слева направо, и имена категорий роли не играют, а только их порядок, определяемый сортировкой запроса.

Поясним сказанное на примере базы данных "Окраска".

Давайте для каждого квадрата просуммируем количество краски каждого цвета:

```
1.SELECT b_q_id, v_color, SUM(b_vol) qty FROM utb JOIN utv
  ON b_v_id = v_id
2.WHERE b_q_id BETWEEN 12 AND 16
3.GROUP BY b_q_id, v_color
4.ORDER BY b_q_id, CASE v_color WHEN 'R' THEN 1 WHEN 'G'
  THEN 2 ELSE 3 END;
```

Здесь мы ограничились только квадратами с номерами в диапазоне 12-16, чтобы, с одной стороны, уменьшить вывод, а, с другой стороны, сделать вывод презентативным. Сортировка по цветам выполнена в порядке RGB. Вот результат:

b_q_id	v_color	Qty
12	R	255
12	G	255
12	B	255

13	B	123
14	R	50
14	B	111
15	R	100
15	G	100
16	G	100
16	B	150

В терминологии CROSSTAB номера баллонов являются идентификаторами строк, а цвета - категориями. Результат поворота должен быть следующим:

square	R	G	B
12	255	255	255
13			123
14	50		111
15	100	100	
16		100	150

Теперь с помощью CROSSTAB попытаемся написать запрос, который бы дал требуемый результат:

```
1. SELECT * FROM
```

```
2.crosstab(
3.$$select b_q_id, v_color, SUM(b_vol) qty FROM utb JOIN
  utv ON b_v_id = v_id
4.WHERE b_q_id BETWEEN 12 AND 16
5.GROUP BY b_q_id, v_color
6.ORDER BY b_q_id, CASE v_color WHEN 'R' THEN 1 WHEN 'G'
  THEN 2 ELSE 3 END; $$
7.) AS ct(square int, "R" bigint,"G" bigint,"B" bigint)
8.ORDER BY square;
```

Здесь мы должны перечислить список столбцов с указанием их типа. При этом столбцы категорий могут быть перечислены не все. Посмотрим на результат (вы можете проверять запросы в консоли, выбрав для исполнения PostgreSQL

square	R	G	B
12	255	255	255
13	123		
14	50	111	
15	100	100	
16	100	150	

Этот результат не вполне совпадает с ожидаемым. Напомним, что здесь важен только порядок. Если квадрат окрашивался только одним цветом, то значение (суммарный объем краски) попадет в первую категорию (у нас она называется R), каким бы этот единственный цвет ни был. Давайте перепишем запрос таким образом, чтобы он давал значения для всех цветов причем в нужном порядке. При этом отсутствующий цвет будем заменять NULL-значением. Чтобы добиться этого, добавим для каждого квадрата по одной строке каждого цвета со значением объема краски равным NULL:

```
1.SELECT b_q_id, v_color, SUM(b_vol) qty FROM(
2.SELECT b_q_id, v_color, b_vol FROM utb
3.JOIN utv ON b_v_id = v_id
4.UNION ALL --вот эта добавка
5.SELECT * FROM (SELECT DISTINCT b_q_id FROM utb) X
6.CROSS JOIN (SELECT 'R' color, NULL::smallint vol
7.              UNION ALL SELECT 'G', NULL UNION ALL
  SELECT 'B', NULL) Y
8.) X
9.WHERE b_q_id BETWEEN 12 AND 16
10.     GROUP BY b_q_id, v_color
11.     ORDER BY b_q_id, CASE v_color WHEN 'R' THEN 1 WHEN
  'G' THEN 2 ELSE 3 END;
```

b_q_id	v_color	Qty
12	R	255
12	G	255
12	B	255
13	R	
13	G	
13	B	123
14	R	50
14	G	
14	B	111
15	R	100
15	G	100
15	B	
16	R	

16	G	100
16	B	150

Теперь ниже представленный запрос даст требуемый результат.

```
1.SELECT * FROM
2.crosstab(
3.$$select b_q_id, v_color, SUM(b_vol) qty FROM(
4.SELECT b_q_id, v_color, b_vol FROM utb
5.JOIN utv ON b_v_id = v_id
6.UNION ALL
7.SELECT * FROM (SELECT DISTINCT b_q_id FROM utb) X
8.CROSS JOIN (SELECT 'R' color, NULL::smallint vol
9.                UNION ALL SELECT 'G', NULL UNION ALL SELECT
  'B', NULL) Y
10.      ) X
11.      WHERE b_q_id BETWEEN 12 AND 16
12.      GROUP BY b_q_id, v_color
13.      ORDER BY b_q_id, CASE v_color WHEN 'R' THEN 1 WHEN
  'G' THEN 2 ELSE 3 END;$$
14.      ) AS ct(square int, "R" bigint,"G" bigint,"B"
  bigint)
15.      ORDER BY square;
```

Наверное, у вас уже возник вопрос: а нельзя ли это сделать как-нибудь проще?

Ответ положительный. Оказывается, у функции CROSSTAB есть второй необязательный параметр - запрос, возвращающий список категорий в том же порядке, в котором выводятся столбцы. Тогда первый запрос, чтобы он давал правильный результат, мы можем переписать следующим образом:

```
1.SELECT * FROM
2.crosstab(
```

```
3.$$select b_q_id, v_color, SUM(b_vol) qty FROM utb JOIN
  utv ON b_v_id = v_id
4.WHERE b_q_id BETWEEN 12 AND 16
5.GROUP BY b_q_id, v_color
6.ORDER BY b_q_id, CASE v_color WHEN 'R' THEN 1 WHEN 'G'
  THEN 2 ELSE 3 END; $$,
7.$$select 'R' UNION ALL SELECT 'G' UNION ALL SELECT 'B';$$
8.) AS ct(square int, "R" bigint,"G" bigint,"B" bigint)
9.ORDER BY square;
```

Поскольку PostgreSQL допускает использование конструктора таблиц, запрос

```
1.SELECT 'R' UNION ALL SELECT 'G' UNION ALL SELECT 'B';
```

можно заменить на более короткий:

```
1.VALUES('R'),('G'),('B');
```

Если вы при выполнении запроса получаете ошибку, что функции crosstab не существует, это означает, что у вас не установлен модуль tablefunc. Он устанавливается простой командой (начиная с версии 9.1)

```
1.CREATE EXTENSION IF NOT EXISTS tablefunc;
```

для конкретной базы данных. Для версий, предшествующих 9.1, достаточно загрузить в pgadmin следующий файл share/contrib/tablefunc.sql и выполнить его.

Общие табличные выражения (CTE)

Чтобы выяснить назначение общих табличных выражений, давайте начнем с примера.

Найти максимальную сумму прихода/расхода среди всех 4-х таблиц базы данных "Вторсырье", а также тип операции, дату и пункт приема, когда и где она была зафиксирована.

Задачу можно решить, например, следующим способом.

```
1.SELECT inc AS max_sum, type, date, point
2.FROM ( SELECT inc, 'inc' type, date, point
3.FROM Income UNION ALL SELECT inc, 'inc' type, date, point
4.FROM Income_o
5.UNION ALL
6.SELECT out, 'out' type, date, point
7.FROM Outcome_o
8.UNION ALL
9.SELECT out, 'out' type, date, point FROM Outcome ) X
10.     WHERE inc >= ALL( SELECT inc FROM Income
11.                UNION ALL
12.                SELECT inc FROM Income_o
13.                UNION ALL SELECT out FROM Outcome_o
14.                UNION ALL SELECT out FROM Outcome );
```

Здесь мы сначала объединяем всю имеющуюся информацию, а затем выбираем только те строки, у которых сумма не меньше, чем каждая из сумм той же выборки из 4-х таблиц.

Фактически, мы дважды написали код объединений четырех таблиц. Как избежать этого? Можно создать представление, а затем адресовать запрос уже к нему:

```
1.CREATE VIEW Inc_Out AS
2.SELECT inc, 'inc' type, date, point
3.FROM Income
```

```
4.UNION ALL
5.SELECT inc, 'inc' type, date, point
6.FROM Income_o
7.UNION ALL
8.SELECT out, 'out' type, date, point
9.FROM Outcome_o
10.      UNION ALL
11.      SELECT out, 'out' type,date, point
12.      FROM Outcome;
13.      GO
14.      SELECT inc AS max_sum, type, date, point
15.      FROM  Inc_Out  WHERE  inc  >=  ALL(  SELECT  inc  FROM
  Inc_Out);
```

Так вот, **CTE** играет роль представления, которое создается в рамках одного запроса и, не сохраняется как объект схемы. Предыдущий вариант решения можно переписать с помощью CTE следующим образом:

```
1.WITH Inc_Out AS (
2.  SELECT inc, 'inc' type, date, point
3.  FROM Income
4.  UNION ALL
5.  SELECT inc, 'inc' type, date, point
6.  FROM Income_o
7.  UNION ALL
8.  SELECT out, 'out' type, date, point
9.  FROM Outcome_o
10.       UNION ALL
11.       SELECT out, 'out' type,date, point FROM Outcome
  )
12.      SELECT inc AS max_sum, type, date, point
13.      FROM Inc_Out WHERE inc >= ALL ( SELECT inc FROM
  Inc_Out);
```

Как видите, все аналогично использованию представления за исключением обязательных скобок, ограничивающих запрос; формально, достаточно лишь заменить CREATE VIEW на **WITH**. Как и для представления, в скобках после имени CTE может быть указан список столбцов, если нам потребуется включить их не все из подлежащего запроса и/или переименовать. Например,

(я добавил дополнительно определение минимальной суммы в предыдущий запрос),

```
1.WITH Inc_Out(m_sum, type, date, point) AS (
2.   SELECT inc, 'inc' type, date, point
3.   FROM Income
4.   UNION ALL
5.   SELECT inc, 'inc' type, date, point
6.   FROM Income_o
7.   UNION ALL
8.   SELECT out, 'out' type, date, point
9.   FROM Outcome_o
10.        UNION ALL
11.        SELECT out, 'out' type,date, point FROM Outcome
   )
12.      SELECT 'max' min_max,* FROM Inc_Out
13.      WHERE m_sum >= ALL( SELECT m_sum FROM Inc_Out)
14.      UNION ALL
15.      SELECT 'min', * FROM Inc_Out
16.      WHERE m_sum <= ALL( SELECT m_sum FROM Inc_Out);
```

Общие табличные выражения позволяют существенно уменьшить объем кода, если многократно приходится обращаться к одним и тем же производным таблицам.

Заметим, что CTE могут использоваться не только с оператором SELECT, но и с другими операторами языка DML. Давайте решим такую задачу:

Пассажиров рейса 7772 от 11 ноября 2005 года требуется отправить другим ближайшим рейсом, вылетающим позже в тот же день в тот же пункт назначения.

Т.е. эта задача на обновление записей в таблице Pass_in_trip. Я не буду приводить здесь решение этой задачи, которое не использует CTE, но вы можете сами это сделать, чтобы сравнить объемы кода двух решений.

Предлагаю пошагово строить решение и представлять результаты в виде запросов на выборку, чтобы вы могли контролировать результаты, выполняя эти запросы онлайн. Поскольку операторы модификации данных пока

запрещены на сайте, я приведу окончательное решение лишь в самом конце. Начнем с таблицы, которую нужно будет обновить:

```
1.WITH Trip_for_replace AS (
2.  SELECT * FROM Pass_in_trip
3.  WHERE trip_no=7772 AND date='20051129' )
4.SELECT * FROM Trip_for_replace;
```

Поскольку CTE играют роль представлений, то их можно в принципе использовать для обновления данных. Слова «в принципе» означают, что CTE является обновляемым, если выполняются определенные условия, аналогичные условиям обновления представлений. В частности, в определении должна использоваться только одна таблица без группировки и вычисляемых столбцов. Отметим, что необходимые условия в нашем случае выполнены.

Теперь нам нужна информация о рейсе 7772 для того, чтобы найти ближайший к нему подходящий рейс. Добавим еще одно CTE в определение:

```
1.WITH Trip_for_replace AS(
2.  SELECT * FROM Pass_in_trip
3.  WHERE trip_no=7772 AND date='20051129' ),
4.Trip_7772 AS ( SELECT * FROM Trip WHERE trip_no=7772 )
5.SELECT * FROM Trip_7772;
```

Обратите внимание, что в одном запросе можно определить любое количество общих табличных выражений. И что особенно важно, CTE может включать ссылку на другое CTE, чем мы, собственно, сейчас и воспользуемся (обратите внимание на ссылку Trip_7772 в определении Trip_candidates).

```
1.WITH Trip_for_replace AS(
2.  SELECT * FROM Pass_in_trip
3.  WHERE trip_no=7772 AND date='20051129' ),
4.Trip_7772 AS ( SELECT * FROM Trip WHERE trip_no=7772 ),
5.Trip_candidates AS ( SELECT Trip.*
6.      FROM Trip, Trip_7772
```

```
7.           WHERE      Trip.town_from+Trip.town_to      =
  Trip_7772.town_from +
8.           Trip_7772.town_to    AND    Trip.time_out    >
  Trip_7772.time_out )
9.SELECT * FROM Trip_candidates;
```

Trip_candidates – это табличное выражение, которое определяет кандидатов на замену, а именно, рейсы, которые вылетают позже, чем 7772, и которые совершаются между теми же городами. Я использую конкатенацию строк town_from+town_to, чтобы не писать отдельные критерии для пункта отправления и места назначения.

Найдем теперь среди строк-кандидатов наиболее близкий по времени рейс:

```
1.WITH Trip_for_replace AS(
2.  SELECT * FROM Pass_in_trip
3.  WHERE trip_no=7772 AND date='20051129' ),
4.Trip_7772 AS ( SELECT * FROM Trip WHERE trip_no=7772 ),
5.Trip_candidates AS( SELECT Trip.* FROM Trip, Trip_7772
6.WHERE Trip.town_from+Trip.town_to = Trip_7772.town_from
  +
7.           Trip_7772.town_to    AND    Trip.time_out    >
  Trip_7772.time_out ),
8.Trip_replace AS(
9.SELECT * FROM Trip_candidates
10.     WHERE   time_out   <=   ALL(SELECT   time_out   FROM
  Trip_candidates) )
11.     SELECT * FROM Trip_replace;
```

Теперь нам осталось последний оператор SELECT заменить на UPDATE, чтобы решить задачу:

```
1.WITH Trip_for_replace AS(
2.  SELECT * FROM Pass_in_trip
3.  WHERE trip_no=7772 AND date='20051129' ),
4.  Trip_7772 AS ( SELECT * FROM Trip WHERE trip_no=7772 ),
5.Trip_candidates AS(
```

```
6.   SELECT Trip.* FROM Trip, Trip_7772
7.  WHERE Trip.town_from+Trip.town_to = Trip_7772.town_from
  +
8.          Trip_7772.town_to     AND     Trip.time_out     >
  Trip_7772.time_out ),
9.     Trip_replace AS( SELECT * FROM Trip_candidates
10.             WHERE time_out <= ALL(SELECT time_out FROM
  Trip_candidates) )
11.        UPDATE  Trip_for_replace  SET  trip_no  =  (SELECT
  trip_no FROM Trip_replace);
```

Здесь мы исходим из довольно естественного предположения о том, что между заданными городами нет двух рейсов, которые бы отправлялись в одно и то же время в одном направлении. В противном случае, понадобился бы дополнительный критерий для отбора единственного рейса, т.к. наша цель – обновление данных, а не представление всех возможных кандидатов на замену.

С использованием CTE с оператором DELETE вы можете познакомиться на примере удаления дубликатов строк из таблицы.

Запрос, который мы использовали для удаления дубликатов в SQL Server

```
1.WITH CTE AS (
2.     SELECT  name,  ROW_NUMBER()  OVER(PARTITION  BY  name
  ORDER BY name) rnk
3.    FROM T
4.      )
5.DELETE FROM CTE
6.WHERE rnk > 1;
```

в PostgreSQL завершится ошибкой:

ОШИБКА: отношение "cte" не существует

Эта ошибка означает, что мы можем удалять строки из базовых таблиц, но не из CTE. Тем не менее, возможно выполнить удаление дубликатов одним запросом, используя CTE.

Поступим следующим образом:

1. Удалим все строки из базовой таблицы, возвращая их в табличное выражение (первое CTE).
2. Используя результат 1 шага, формируем уникальные строки, которые должны остаться в таблице (второе CTE).
3. Вставляем строки, полученные на шаге 2 в базовую таблицу.

Воспользуемся таблицей из цитируемого примера, чтобы написать запрос:

```
1.CREATE TABLE T (name varchar(10));
2.INSERT INTO T VALUES
3.('John'),
4.('Smith'),
5.('John'),
6.('Smith'),
7.('Smith'),
8.('Tom');
```

Вот и сам запрос

```
1.WITH t_deleted AS
2.(DELETE FROM T returning *), -- 1 шаг
3.t_inserted AS
4.(SELECT name, ROW_NUMBER() OVER(PARTITION BY name ORDER
  BY name) rnk
5.    FROM t_deleted) -- 2 шаг
6.INSERT INTO T SELECT name FROM t_inserted
7.WHERE rnk=1; -- 3 шаг (сюда мы перенесли условие отбора
  из 2 шага для сокращения кода)
```

Если теперь выполнить запрос

```
1.SELECT * FROM T;
```

то получим требуемый результат

name
John
Smith
Tom

Рекурсивные CTE

У CTE есть еще одно важное назначение. С его помощью можно написать рекурсивный запрос, т.е. запрос, который, написанный один раз, будет повторяться многократно пока истинно некоторое условие.

Рекурсивный CTE имеет следующий вид:

```
1.WITH <имя>[(<список столбцов>)]
2.AS(
3.< SELECT... > -- анкорная часть
4.UNION ALL -- рекурсивная часть
5.< SELECT...FROM <имя>… >
6.WHERE <условие продолжения итераций>
7.)
```

От обычного CTE-запроса рекурсивный отличается только рекурсивной частью, которая вводится предложением UNION ALL. Обратите внимание, что в рекурсивной части присутствует ссылка на имя CTE, т.е. внутри CTE ссылается само на себя. Это, собственно, и есть рекурсия. Естественно, анкорная и рекурсивная части должны иметь одинаковый набор столбцов.

В **MySQL** рекурсивные CTE (как и обычные) появились в версии 8. Их синтаксис фактически отличается от синтаксиса SQL Server только одним словом RECURSIVE:

```
1.WITH RECURSIVE <имя>[(<список столбцов>)]
2.AS(... и так далее
```

Перейдем к примеру. Рассмотрим задачу получения алфавита, т.е. таблицы алфавитных символов - прописных латинских букв. Чтобы было с чем сравнивать, решим сначала эту задачу с помощью генерации числовой последовательности, которая рассматривалась в параграфе 8.1.

```
1.SELECT CHAR(ASCII('A')+5*(a-1) + b-1) AS num
2.FROM (SELECT 1 a UNION ALL SELECT 2 UNION ALL SELECT 3
3.UNION ALL SELECT 4 UNION ALL SELECT 5 UNION ALL SELECT 6
4.) x CROSS JOIN
5. (SELECT 1 b UNION ALL SELECT 2 UNION ALL SELECT 3
6.UNION ALL SELECT 4 UNION ALL SELECT 5
7.) y
8.WHERE 5*(a-1) + b <= 26
9.ORDER BY 1;
```

А вот решение с помощью рекурсивного CTE

```
1.WITH Letters AS(
2.SELECT ASCII('A') code, CHAR(ASCII('A')) letter
3.UNION ALL
4.SELECT code+1, CHAR(code+1) FROM Letters
5.WHERE code+1 <= ASCII('Z')
6.)
7.SELECT letter FROM Letters;
```

В запросе анкорной части определяем ASCII-код первой буквы алфавита и соответствующий ему символ. В запросе рекурсивной части мы просто увеличиваем ASCII-код на единицу, обращаясь к CTE в предложении FROM. В результате к строке с первым символом будут последовательно добавляться (UNION ALL) строки со следующими буквами в порядке их ASCII-кодов.

Итерации будут продолжаться до тех пор, пока условие *code +1 <= ascii('Z')* будет истинным, т.е. пока не будет добавлена буква "Z".

Оператор

```
1. SELECT letter FROM Letters
```

собственно и служит для обращения к CTE, запуска рекурсии и вывода результата. Все остальное можно считать определением.

Следует заметить, что по умолчанию допускается 100 итераций. Это значит, что если условие прекращения итераций не выполнено ранее, то рекурсия будет остановлена после выполнения 100 итераций. Максимальное число итераций можно изменить с помощью «хинта»

```
1. OPTION(MAXRECURSION N)
```

где N – максимальное число итераций. Значение 0 не ограничивает число итераций. Нужно с осторожностью использовать это значение, т.к. оно чревато зацикливанием.

Если запрос не был завершен в пределах указанного числа итераций, возникает ошибка (полученные к этому моменту строки возвращаются):

The statement terminated. The maximum recursion N has been exhausted before statement completion. (Выполнение оператора прервано. Достигнут предел максимального числа итераций N до завершения выполнения оператора).

В MySQL наша задача будет иметь аналогичное решение:

```
1. WITH RECURSIVE Letters AS(
2.     SELECT ASCII('A') code, CHAR(ASCII('A')) letter
3.     UNION ALL
4.     SELECT code+1, CHAR(code+1) FROM Letters
5.     WHERE code+1 <= ASCII('Z')
6.     )
7.     SELECT letter FROM Letters;
```

Давайте решим еще одну задачу в качестве ответа на вопрос, который мне неоднократно встречался на просторах Интернет.

Преобразовать текст в столбце таблицы таким образом, чтобы каждое слово начиналось с заглавной буквы.

Вот пример данных и требуемый результат:

name	rep
delta	Delta
KSI PSI	Ksi Psi
alfa beta gamma zeta	Alfa Beta Gamma Zeta
ALfa and omegA	Alfa And Omega

За небольшим числом исключений (среди которых можно упомянуть аббревиатуры и инициалы) можно считать, что слову внутри текста предшествует пробел. Это можно использовать в качестве критерия поиска нужных нам элементов текста. Предлагаю реализовать такой достаточно примитивный алгоритм:

1. Первую букву текста делаем прописной, а остальные - строчными.

2. Затем каждую конструкцию "пробел+буква" переводим в верхний регистр.

С первым пунктом алгоритма все просто:

```
1. SELECT name, UPPER(LEFT(name, 1)) + LOWER(SUBSTRING(name,
   2, LEN(name) - 1)) rep
2. FROM
3. (SELECT 'ALfa and omegA' AS name
4. UNION ALL SELECT 'alfa beta gamma    zeta'
5. UNION ALL SELECT 'KSI PSI'
```

```
6.UNION ALL SELECT 'delta'
7.) X;
```

name	rep
ALfa and omegA	Alfa and omega
alfa beta gamma zeta	Alfa beta gamma zeta
KSI PSI	Ksi psi
delta	Delta

Для реализации второго пункта есть варианты. Поскольку букв латинского алфавита не так много (26), можно просто сделать 26 замен. Я не поленюсь и приведу полный вариант, чтобы вы могли поэкспериментировать с запросом.

Итак,

```
1.SELECT                                                    name,
  REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(
2.REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(
3.REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(
4.REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(
5.REPLACE(REPLACE(rep, ' a', ' A'), ' b', ' B'), ' c', '
  C'), ' d', ' D'),
6. ' e', ' E'), ' f', ' F'), ' g', ' G'), ' h', ' H'), '
  i', ' I'), ' j', ' J'), ' k', ' K'),
7.' l', ' L'), ' m', ' M'),  ' n',' N'), ' o', ' O'), ' p',
  ' P'), ' q', ' Q'), ' r', ' R'),
8. ' s', ' S'), ' t', ' T'), ' u', ' U'), ' v', ' V'),
9. ' w', ' W'), ' x', ' X'), ' y', ' Y'), ' z', ' Z')
10.      FROM(
11.      SELECT      name,      UPPER(LEFT(name,1))      +
  LOWER(SUBSTRING(name, 2, LEN(name)-1)) rep
12.      FROM
13.      (SELECT 'ALfa and omegA' AS name
14.      UNION ALL SELECT 'alfa beta gamma    zeta'
15.      UNION ALL SELECT 'KSI PSI'
```

```
16.      UNION ALL SELECT 'delta'
17.      ) X
18.      ) Y;
```

Нетрудно догадаться, что следующий вариант будет использовать рекурсивный CTE.

Сначала напишем два простых **CTE**, которые формируют наш тестовый пример и определяют ASCII-код первой буквы алфавита (A) - не писать же константу. :-) Далее последует анкорная часть, которая выполняет ранее описанную операцию приведения всего текста к нижнему регистру с заглавной первой буквой. Здесь же выполним замену символа с кодом code и предшествующим ему пробелом на... него же. Пусть вас не смущает такая, казалось бы, бесполезная замена. Дело в том, что для регистронезависимых баз данных символы 'a' и 'A' не различаются. Давайте пока на этом остановимся и посмотрим результат.

```
1.WITH NM(name) AS
2.(SELECT 'ALfa and omegA' AS name
3.UNION ALL SELECT 'alfa beta gamma    zeta'
4.UNION ALL SELECT 'KSI PSI'
5.UNION ALL SELECT 'delta'
6.),
7.Ascii_code AS(
8.SELECT  ASCII('A') AS code
9.),
10.      Repl(name, code, rep) AS
11.      (SELECT name, code, REPLACE(UPPER(LEFT(name, 1)) +
12.      LOWER(SUBSTRING(name,   2,   LEN(name)   -   1)),'
   '+CHAR(code),' '+CHAR(code)) rep
13.      FROM Ascii_code, NM
14.      )
15.      SELECT name, rep FROM Repl;
```

name	rep
ALfa and omegA	Alfa And omega
alfa beta gamma zeta	Alfa beta gamma zeta
KSI PSI	Ksi psi
delta	Delta

Добавим, наконец, рекурсивную часть, в которой мы выполним замену буквы с кодом code+1. Рекурсия будет продолжаться до тех пор, пока не будет нарушено условие *code < ASCII('Z')*, т.е. пока мы не переберем все буквы.

Что же мы получим на выходе? К строкам, которые были получены в результате выполнения анкорной части, на каждой итерации будут добавлены (UNION ALL) те же строки с заменой очередной буквы. Отметим большой объем результата при использовании данного метода; в нашем случае это 4x26 = 104 строки. Из этого множества строк нас интересуют только те, которые получены в результате последней итерации, т.е. когда были выполнены все замены. Этой последней итерации соответствует условие *code = ASCII('Z')*, которое и используется в финальном запросе:

```
1.WITH NM(name) AS
2.(SELECT 'ALfa and omegA' AS name
3.UNION ALL SELECT 'alfa beta gamma    zeta'
4.UNION ALL SELECT 'KSI PSI'
5.UNION ALL SELECT 'delta'
6.),
7.Ascii_code AS(
8.SELECT  ASCII('A') AS code
9.),
10.      Repl(name, code, rep) AS
11.      (SELECT name, code, REPLACE(UPPER(LEFT(name, 1)) +
12.      LOWER(SUBSTRING(name,   2,   LEN(name)   -   1)),'
   '+CHAR(code),'  '+CHAR(code)) rep
13.      FROM Ascii_code, NM
14.      UNION ALL
```

```
15.      SELECT  name,  code+1  code,  REPLACE(rep,'  ' +
  CHAR(code+1), ' ' + char(code + 1)) rep
16.       FROM Repl
17.      WHERE code < ASCII('Z')
18.      )
19.      SELECT name, rep FROM Repl WHERE code=ASCII('Z');
```

Я хотел бы предостеречь вас от чрезмерного увлечения рекурсивными CTE, поскольку они зачастую проигрывают в производительности "традиционным" методам. Я не буду далеко ходить за примерами и сравню два представленных здесь метода. Увеличив количество обрабатываемых строк до 10000, я получил такое время использования CPU:

метод на основе REPLACE: **842** ms

рекурсивный метод: **6615** ms

Безусловно, есть задачи, которые нельзя решить непроцедурно в рамках стандарта SQL-92. В этих случаях использование рекурсивных CTE вполне обоснованно. В остальных случаях я бы рекомендовал выполнять тесты производительности для альтернативных решений.

Кстати, в **Oracle** и **PostgreSQL** есть встроенная функция **INITCAP**, которая решает данную задачу:

```
1.SELECT INITCAP(name)
2.  FROM
3.   (SELECT 'ALfa and omegA' AS name
4.     UNION ALL SELECT 'alfa beta gamma    zeta'
5.     UNION ALL SELECT 'KSI PSI'
6.     UNION ALL SELECT 'delta'
7. ) X;
```

Вы можете использовать **консоль**, чтобы убедиться в этом.

О генерации числовых последовательностей в SQL Server

В очередной реализации MS SQL Server 2005 появилась возможность использования рекурсивных CTE конструкций, позволяющих реализовывать циклы для генерации числовых последовательностей и итерационных вычислений.

Введение в рекурсии MS SQL Server можно найти в соответствующих руководствах Microsoft, учебнике и Интернете. В данном параграфе мы рассмотрим только новые примеры, помогающие в ее практическом освоении. Самым простым примером использования рекурсивного CTE является генерация ограниченной числовой последовательности натурального ряда: 1,2,3, ... N.

```
1.WITH Series(a) AS
2.(
3. SELECT 1
4. UNION ALL
5. SELECT a+1 FROM Series WHERE a < 100
6.)
7.SELECT * FROM Series;
```

Эта конструкция предназначена для генерации последовательности целых чисел (в виде одноколоночной таблицы) от 1 до 100.

А. Итерационные вычисления

В элементарной и высшей математике есть много интересных последовательностей, обладающих замечательными свойствами. Некоторые

последовательности сходятся, и их можно использовать для реализации вычислительных алгоритмов или для вычисления алгебраических и трансцендентных чисел, значений тригонометрических функций, для нахождения корней уравнений, решения систем линейных алгебраических уравнений и пр. Другие последовательности, такие как факториал n!, коэффициенты бинома Ньютона, числа Фибоначчи представляют собой расходящиеся последовательности, которые имеют широкое использование в теории вероятности и математической статистике.

Эти последовательности получаются с помощью итераций (рекурсий SQL Server), например:
A1, A2, A3, A4 = fun(A1, A2, A3).

Здесь A1, A2, A3 - начальные значения для итерационного процесса, fun - функция для вычисления 4-го, пятого и т.д. чисел, которая всегда задействует предыдущие 3 числа. Предположим, что процесс стартует с трех одинаковых чисел A1 = A2 = A3 = 1. Тогда схема реализации с использованием рекурсивных CTE будет иметь следующий вид:

```
1.WITH TABLE(iter, A1, A2, A3, A4) AS
2.(
3. SELECT iter = 1, A1 = 1, A2 = 1, A3 = 1, A4 = fun(1, 1,
  1)
4. UNION ALL
5. SELECT iter + 1, A1 = A2, A2 = A3, A3 = A4, A4=fun(A2,
  A3, A4)
6. FROM TABLE WHERE iter < 50
7.)
8.SELECT * FROM TABLE;
```

Здесь колонка iter введена для вывода номера итерации, колонка [A1] содержит пятьдесят первых членов последовательности, [A2], [A3] и [A4] – вспомогательные колонки для хранения необходимых промежуточных результатов.

Обобщением такого подхода является следующий пример. Пусть n >= 1, m >= 1. Тогда последовательные вычисления

```
A[n+1] = fun(A[1], A[2], …, A[n])
A[n+2] = fun(A[2], A[3], …, A[n+1])
…
A[n+m] = fun(A[m], A[m+1], …, A[m+n-1])
```

приводят к генерации m (m >= 1) новых членов последовательности A[n+1],…, A[n+m].

Мы не будем приводить примера использования рекурсии в общем случае, так как это возможно разве что в виде некоторого псевдокода. Читатели могут попытаться построить конкретные примеры самостоятельно. Отметим, что результирующая таблица выглядит следующим образом:

```
A[1]        A[2]        …        A[n]            A[n+1]
A[2]        A[3]        …        A[n+1]          A[n+2]
…
A[m+1]   A[m+2]   …      A[m+n-1]   A[m+n]
```

B. О последовательности чисел Фибоначчи

Последовательность Фибоначчи может быть вычислена с помощью следующей итерационной формулы:

```
A[i+2] = A[i] + A[i+1];  i = 1,…, n; A[1] = A[2] = 1.
```

Здесь очень легко увидеть аналогию с общим случаем предыдущего пункта, если ввести функцию f(x,y)=x+y. Для хранения номера итерации i можно использовать колонку [iter], для A[i] используется колонка [a], для A[i+1] и A[i+2] - колонки [b] и [c]. Колонка [d] не потребуется. Используя общий подход, расчет чисел Фибоначчи можно выразить следующим кодом:

```
1.WITH Fibonacci(iter,a,b,c) AS
2.(
3. SELECT iter=1, a=1, b=1, c=1+1
4. UNION ALL
5. SELECT iter+1, a=b, b=c, c=b+c
6. FROM Fibonacci WHERE b < 1000
7.)
8.SELECT * FROM Fibonacci;
```

Приведем результат запроса для вычисления чисел Фибоначчи, меньших 1000 (они находятся во втором столбце таблицы).

iter	a	b	c
1	1	1	2
2	1	2	3
3	2	3	5
4	3	5	8
5	5	8	13
6	8	13	21
7	13	21	34
8	21	34	55
9	34	55	89
10	55	89	144
11	89	144	233
12	144	233	377
13	233	377	610
14	377	610	987
15	610	987	1597
16	987	1597	2584

C. Нахождение корней уравнений

Большой раздел математического и функционального анализа посвящен нахождению корней уравнений функции одного и многих переменных. Корнем уравнения g(x) = 0 называется число r (или вектор r),

удовлетворяющее условию: g(r) = 0. Общим методом решения таких уравнений является сведение задачи к задаче о неподвижной точке:

```
x=f(x).
```

Смысл такого сведения состоит в нахождении такой функции f, при которой уравнения g (x) = 0 и x = f(x) являются эквивалентными. Кроме того, оператор f должен быть сжимающим. То есть, если значение r1 находится рядом с решением r, то значение r2=f(r1) должно быть еще ближе к решению: abs(r-r2) < A * abs(r-r1), где положительная константа A меньше единицы (A<1) и не зависит от выбора значения r1.

Сжимающих отображений может быть много. Предпочтительными являются те из них, для которых константа A принимает меньшее значение. Чем меньше константа, тем быстрее сходится процесс нахождения корня уравнения g(x):

```
r2 = f(r1), r3 = f(r2),  r4 = f(r3) …
```

Приведем иллюстрирующий пример на SQL.

D. Итерационное вычисление квадратного корня

Квадратный корень из числа a – это решение уравнения x*x = a. В терминах предыдущего пункта это корень уравнения g(x) = 0, где функция g(x) = x*x - a. Вычисление этих чисел SQL-запросом труда не представляет - есть встроенная функция sqrt(a) или power(a,0.5). Тем не менее, проиллюстрируем на примере нахождения квадратного корня подход, который может использоваться в случаях, когда нет соответствующей встроенной функции, а сжимающее отображение известно.

Итерационный аналитический алгоритм для вычисления квадратного корня известен из курса школьной математики, и изложен, например, здесь.

Его можно записать в виде сжимающего отображения x = f(x), где

```
f(x)=1/2*(x+a/x).
```

Легко убедиться в том, что уравнение x = 1/2*(x + a/x) эквивалентно уравнению x*x = a для x <> 0. Читатель с математическим образованием может попытаться доказать, что это преобразование действительно сжимающее, и, следовательно, может быть использовано для итерационного процесса нахождения корня уравнения.
Для иллюстрации алгоритма приведем пример sql-кода для вычисления квадратного корня из числа a = 3:

```
1.WITH Square3(a,b,c) AS
2.(
3. SELECT 1, CAST(3 AS float(53)), CAST(3 AS float(53))
4. UNION ALL
5. SELECT a+1, 1./2.*(b+3/b), 1./6.*3*(c+3./c) FROM Square3
  WHERE a < 7
6.)
7.SELECT iteration=a, Exact=sqrt(CAST(3 AS float(53))),
  Res1=b, Res2=c
8.FROM Square3;
```

Здесь столбец [a] введен для вывода номера итерации, столбцы [b] и [c] вычисляют квадратный корень двумя арифметически эквивалентными способами. Итерации не используют встроенные операции sqrt или power, но для контроля мы вывели в колонке exact значение квадратного корня, вычисленное с помощью встроенной функции.

iteration	Exact	Res1	Res2
1	1.7320508075688772	3.0	3.0
2	1.7320508075688772	2.0	1.999992
3	1.7320508075688772	1.75	1.7499920000160001
4	1.7320508075688772	1.7321428571428572	1.7321358469571777
5	1.7320508075688772	1.7320508100147274	1.7320438814531474
6	1.7320508075688772	1.7320508075688772	1.7320438793794952
7	1.7320508075688772	1.7320508075688772	1.7320438793795034

Видно, что уже на 6-й итерации вычисления в третьем столбце [Res1] привели к совпадению со значением встроенной функции [Exact] в пределах точности FLOAT(53) для квадратного корня из трех. Вычисления в четвертом столбце [Res2] – нет. В чем же причина таких различий? Не сразу очевидно, но причина в том, что выражение (1./6.) вычисляется с большой ошибкой, так как операнды не приведены к 8-байтовому представлению вещественных чисел (двойная точность). Это повлияло на все вычисления, и мы получили

только 5-6 правильных значащих цифр в результате, что согласуется с теорией вычислений в вещественной арифметике с одинарной точностью.

Функция EOMONTH

Как узнать последний день месяца по заданной дате, например, текущего месяца?

Текущую дату мы можем узнать, используя встроенную функцию current_timestamp:

```
1. SELECT current_timestamp;
```

Чтобы узнать последний день предыдущего месяца, мы можем от текущей даты отнять номер текущего дня месяца, т.е. количество дней, прошедших от начала месяца:

```
1. SELECT dateadd(dd, -day(current_timestamp), current_timestamp);
```

Тогда для текущего месяца нам потребуется предварительно добавить один месяц к текущей дате:

```
1. SELECT dateadd(dd, -day(dateadd(mm, 1, current_timestamp)),
2. dateadd(mm, 1, current_timestamp));
```

Уберем, наконец, из полученного результата компоненту времени:

```
1.SELECT CAST(
2.dateadd(dd,   -day(dateadd(mm,   1,   current_timestamp)),
  dateadd(mm, 1, current_timestamp))
3.AS date);
```

В **SQL** Server 2012 появилась функция **EOMONTH**, которая позволяет сделать то же самое без применения "процедурной" логики:

```
1.SELECT CAST(
2.dateadd(dd, -day(dateadd(mm, 1, current_timestamp)),
3.dateadd(mm, 1, current_timestamp)) AS date
4.) old_way, eomonth(current_timestamp) new_way;
```

Если вы получаете ошибку при выполнении последнего запроса, значит учебник еще не переехал на версию SQL Server, поддерживающую EOMONTH.

Ну, а для времени, когда я написал этот запрос, результаты, естественно, совпали:

old_way	new_way
2016-07-31	2016-07-31

Мы уже знаем, что функция EOMONTH имеет аргументом выражение типа даты. Кроме того, функция имеет также второй (необязательный) целочисленный аргумент, представляющий число месяцев, которые, при наличии, будут добавлены к дате, представленной первым аргументом. Например, следующий запрос даст нам последние дни предыдущего, текущего и следующего месяца для даты '2016-01-28':

```
1.SELECT eomonth('2016-01-28',-1) prev_month,
2.                  eomonth('2016-01-28') this_month,
3.                  eomonth('2016-01-28', 1) next_month;
```

prev_month	this_month	next_month
2015-12-31	2016-01-31	2016-02-29

Функция STRING_AGG

Агрегация текстовых данных

Рассмотрим такую задачу.

Перечислить через запятую все корабли из таблицы Ships, которые принадлежат Японии.

Получить список кораблей Японии труда не составляет:

```
1. SELECT name FROM Ships s JOIN Classes c ON s.class=c.class
2. WHERE country='Japan'
3. ORDER BY name;
```

В MySQL есть замечательная агрегатная функция **GROUP_CONCAT**, которая решает поставленную задачу:

```
1. SELECT GROUP_CONCAT(name) ships_list FROM Ships s JOIN
   Classes c ON s.class=c.class
2. WHERE country='Japan'
3. ORDER BY name;
```

ships_list
haruna,hiei,kirishima,kon,musashi,yamato

По умолчанию в качестве разделителя как раз используется запятая, хотя мы можем выбрать любой символ.

Если выполнить группировку, то легко получить список кораблей для каждой страны:

```
1.SELECT country, GROUP_CONCAT(name) ships_list
2.FROM Ships s JOIN Classes c ON s.class=c.class
3.GROUP BY country
4.ORDER BY country, name;
```

country	ships_list
gt.britain	renown,repulse,resolution,ramillies,revenge,royal oak,royal sovereign
japan	haruna,hiei,kirishima,kongo,musashi,yamato
usa	iowa,missouri,new jersey,wisconsin,north carolina,south dakota,washington,california,tennessee

Для SQL Server решение нашей задачи можно получить менее естественным способом - через представление результата выборки в форме XML:

```
1.SELECT STUFF(
2.(SELECT ','+name AS 'data()' FROM Ships s JOIN Classes c
  ON s.class=c.class
3.WHERE country='Japan'
4.ORDER BY name FOR XML PATH('')
5.),1,1,'');
```

Группировка по стране еще добавит сложности. Но мы не будем этого делать, поскольку в SQL Server, начиная с версии 2017, появилась функция **STRING_AGG**, позволяющая конкатенировать строки. Эта функция

имеет два обязательных аргумента - строковое выражение, которое и будет использоваться для сцепления, и разделитель.

```
1. SELECT country, STRING_AGG(name,',') ships_list
2. FROM Ships s JOIN Classes c ON s.class=c.class
3. GROUP BY country
4. ORDER BY country;
```

country	ships_list
Gt.Britain	Renown,Repulse,Resolution,Ramillies,Revenge,Royal Oak,Royal Sovereign
Japan	Musashi,Yamato,Haruna,Hiei,Kirishima,Kongo
USA	North Carolina,South Dakota,Washington,Iowa,Missouri,New Jersey,Wisconsin,California,Tennessee

Из представленного результата видно, что корабли в списке не отсортированы. Сортировка в стиле GROUP_CONCAT здесь не работает. Чтобы задать порядок сортировки, используется необязательное предложение **WITHIN GROUP**:

```
1. SELECT country, STRING_AGG(name,',') WITHIN GROUP (ORDER
   BY name) ships_list
2. FROM Ships s JOIN Classes c ON s.class=c.class
3. GROUP BY country
4. ORDER BY country;
```

country	ships_list
Gt.Britain	Ramillies,Renown,Repulse,Resolution,Revenge,Royal Oak,Royal Sovereign
Japan	Haruna,Hiei,Kirishima,Kongo,Musashi,Yamato

USA	California,Iowa,Missouri,New Jersey,North Carolina,South Dakota,Tennessee,Washington,Wisconsin

При использовании функции STRING_AGG соединяемые значения преобразуются к типу данных VARCHAR (NVARCHAR).

Типом результата будет VARCHAR(8000) или NVARCHAR(4000), если среди соединяемых значений не будет значений типа VARCHAR(MAX) или NVARCHAR(MAX). В последнем случае результат будет иметь тип VARCHAR(MAX) (или NVARCHAR(MAX) соответственно).

В нижеследующем примере будет получена ошибка, поскольку длина результата - 8011 - превышает значение 8000:

- 8000 (значение 'a' будет дополнено пробелами, т.к. используется тип постоянной длины CHAR),
- 1 символ на запятую-разделитель,
- 10 (длина значения 'bbbbbbbbbb').
-

```
1.declare @a char(8000), @b varchar(10);
2.SELECT @a='a',@b=replicate('b',10);
3.SELECT string_agg(x,',') res
4.FROM (VALUES(@b),(@a)) X(x);
```

Результат агрегирования STRING_AGG превышает предел в 8000 байтов. Используйте типы LOB, чтобы избежать усечения результатов.

Однако если, скажем, для переменной @b мы используем тип VARCHAR(MAX), то код выполнится без ошибки:

```
1.declare @a char(8000), @b varchar(MAX);
2.SELECT @a='a',@b=replicate('b',10);
3.SELECT string_agg(x,',') res
4.FROM (VALUES(@b),(@a)) X(x);
```

res
bbbbbbbbbb,a ...(еще 7999 пробелов)

Функция STRING_SPLIT

Функция **STRING_SPLIT** выполняет операцию, обратную **STRING_AGG**. Она принимает на входе символьную строку и разбивает её на подстроки по заданному вторым параметром разделителю. Эти подстроки формируют значения унарной таблицы. К единственному столбцу этой таблицы можно обратиться по имени *value*.

```
1. SELECT * FROM STRING_SPLIT('0 1 2 3 4 5 6 7 8 9',' ');
```

value
0
1
2
3
4
5
6
7
8
9

Фактически функция STRING_SPLIT играет роль нестандартного конструктора одностолбцовой таблицы. Стандартная альтернатива выглядит так:

```
1. SELECT * FROM
   (VALUES(0),(1),(2),(3),(4),(5),(6),(7),(8),(9))
   X(value);
```

Больший интерес представляет случай, когда на вход функции передается значение столбца некоторой таблицы.

Пусть, например, нам требуется разбить многословные названия кораблей в таблице Ships на отдельные слова. Тогда можно выполнить следующий запрос:

```
1. SELECT name, value FROM Ships CROSS APPLY
   STRING_SPLIT(name,' ')
2. WHERE name LIKE '% %';
```

name	value
New Jersey	New
New Jersey	Jersey
North Carolina	North
North Carolina	Carolina
Royal Oak	Royal
Royal Oak	Oak
Royal Sovereign	Royal
Royal Sovereign	Sovereign

South Dakota	South
South Dakota	Dakota

Функция CHOOSE

Нестандартная функция **CHOOSE** появилась в SQL Server версии 2012.

Функция CHOOSE используется для выбора одного из нескольких вариантов. Выбор осуществляется на основании индекса (номера варианта), который является первым параметром функции. Остальные параметры представляют собой варианты выбора. Будет выбран тот вариант, номер которого в списке параметров, совпадает с индексом.

Например, запрос

```
1.SELECT CHOOSE(2,'PC','Laptop','Printer');
```

вернет Laptop, т.к. это второй элемент списка.

Больший интерес представляет случай, когда индексом является выражение, включающее столбцы таблиц.

Пусть нам требуется вместо номера пункта (в базе данных "Вторсырье") выводить его название.

Поскольку в базе данных названия пунктов не хранятся, будем формировать их "налету", используя функцию CHOOSE:

```
1.SELECT DISTINCT point,
2.CHOOSE(point, 'point A', 'point B', 'point C') point_name
3.FROM outcome;
```

point	point_name
1	point A
2	point B
3	point C

Очевидно, что функция CHOOSE является еще одним частным вариантом выражения CASE. Наш пример с использовании функции CASE можно переписать в виде:

```
1.SELECT DISTINCT point,
2.CASE point
3.      WHEN 1 THEN 'point A'
4.      WHEN 2 THEN 'point B'
5.      WHEN 3 THEN 'point C'
6. END point_name
7.FROM outcome;
```

Если указанному индексу не будет найдено соответствия, то результатом функции CHOOSE будет NULL:

```
1.SELECT CHOOSE(4,'PC','Laptop','Printer');
```

Для обработки этого случая в выражении CASE предусмотрено предложение ELSE. С помощью этого предложения мы можем вместо NULL (по умолчанию) вывести, например, UNKNOWN:

```
1.SELECT CASE 4
2.      WHEN 1 THEN 'PC'
3.      WHEN 2 THEN 'Laptop'
4.      WHEN 3 THEN 'Printer'
5.     ELSE 'UNKNOWN'
```

```
6.END;
```

Мы легко можем преодолеть этот "недостаток" функции CHOOSE, обернув её функцией COALESCE:

```
1.SELECT
  COALESCE(CHOOSE(4,'PC','Laptop','Printer'),'UNKNOWN');
```

но это уже как бы "масло масляное" - case от case.

Можно сказать, что функция CHOOSE имеет специфическое применение, но в этих ограниченных случаях она позволяет упростить запись.

В заключение приведем еще один пример.

Для каждой модели указать, является её номер четным (even) или нечетным (odd).

```
1.SELECT model, CHOOSE(model%2+1, 'EVEN', 'ODD')
2.FROM product
3.WHERE ISNUMERIC(model) =1;
```

Предикат в предложении WHERE используется для того, чтобы ограничиться моделями, номера которых представлены числом. Обратите внимание на тип данных столбца model! Надеюсь, что те, кто решает упражнения на сайте sql-ex.ru уже в курсе. :-) Они также должны быть в курсе, что использование функции ISNUMERIC в данном контексте не является радикальным решением.

Заметки о типах данных

В этом разделе я предполагаю писать о том, что мне показалось странным или необычным в отношении типов данных и их преобразовании, о том, что противоречит стандарту и называется «особенностями реализации».

CHAR и VARCHAR

Недавно мне довелось искать ошибку в решении, которое содержало такое преобразование:

```
1.CAST(model AS VARCHAR)
```

Те, кто изучил схему «Компьютеры», подумают о бессмысленности преобразования типа в тот же самый тип (столбец model определен как VARCHAR(50)). Однако именно это преобразование и делало запрос неверным.

Дело в том, что, если размер типа при преобразовании не указан, то в SQL Server принимается значение по умолчанию, которое для VARCHAR равно 30. При этом если преобразуемая строка имеет больший размер, то отсекаются все символы кроме первых 30-ти. Разумеется, никакой ошибки при этом не возникает. Как раз на «длинных» номерах моделей предложенное решение и давало неверный результат. Как говорится в таких случаях, читайте документацию. Однако интересно, что по этому поводу говорит Стандарт?

Согласно стандарту, если для типов CHAR и VARCHAR размер не указан, то подразумевается CHAR(1) и VARCHAR(1) соответственно. Давайте проверим, как следуют стандарту доступные мне СУБД: SQL Server, MySQL, PostgreSQL.

Тут имеется два аспекта:

1. Преобразование типа

2. Использование типов **CHAR/VARCHAR** при описании схемы (DDL).

Начнем с преобразования типа.

SQL Server 2008

```
1. SELECT
   CAST('11111111112222222222333333333344444444445555555555
   ' AS CHAR) chr,
2. CAST('11111111112222222222333333333344444444445555555555
   ' AS VARCHAR) vchr;
```

В результате получим

chr	vchr
111111111122222222223333333333	111111111122222222223333333333

То есть оба символьных типа усекаются до значения по умолчанию, которое равно 30. Никаких сообщений об ошибках не возникает, что, собственно, соответствует стандарту.

PostgreSQL 8.3

```
1. SELECT
   CAST('11111111112222222222333333333344444444445555555555
   ' AS CHAR) AS chr,
2. CAST('11111111112222222222333333333344444444445555555555
   ' AS VARCHAR) AS vchr;
```

chr	vchr
1	11111111112222222222333333333344444444445555555555

Налицо половинчатое следование стандарту, т.е. соответствие ему в отношении типа CHAR. Что касается типа VARCHAR, то согласно документации, если длина строки не указана, принимается строка любого

размера, т.е. усечения не происходит. *(If character varying is used without length specifier, the type accepts strings of any size.)*

MySQL 5.0

Как говорится, чем дальше, тем «страньше». Оказывается, в MySQL преобразование к типу VARCHAR вообще не поддерживается. Нам остается проверить только преобразование к CHAR:

```
1. SELECT
   CAST('11111111112222222222333333333344444444445555555555
   ' AS CHAR) chr;
```

chr
11111111112222222222333333333344444444445555555555

Т.е. строка не усекается; при этом в документации читаю: «Если при использовании функций CAST и CONVERT размер не указан, то длина по умолчанию – 30.» *(When n is not specified when using the CAST and CONVERT functions, the default length is 30).*

Посмотрим теперь, как обстоят дела с определением данных. Ниже приведен тестовый скрипт.

```
1. CREATE TABLE Test_char( chr CHAR, vchr VARCHAR );

1. DELETE FROM Test_char;

1. INSERT INTO Test_char
2. VALUES
   ('1','11111111112222222222333333333344444444445555555555
   ');
3.
4. INSERT INTO Test_char
```

```
5.VALUES
  ('1111111111222222222233333333334444444444555555555',
  '1');
6.
7.INSERT INTO Test_char
8.VALUES    ('2',CAST('111111111122222222223333333333'   AS
  VARCHAR));
9.
10.     INSERT INTO Test_char
11.     VALUES   (CAST('111111111122222222223333333333'  AS
  CHAR), '2');
12.
13.     INSERT INTO Test_char
14.     VALUES ('3', '3');
```

```
1.SELECT * FROM Test_char;
```

SQL Server 2008

chr	vchr
3	3

Итак, будет вставлена только одна строка, содержащая по одному символу для каждого из столбцов. При вставке остальных строк получаем сообщение об ошибке:

String or binary data would be truncated. The statement has been terminated.

которое означает, что следует уменьшить размер строки.

Хотя здесь выдержано соответствие стандарту, мне кажется, что есть некое противоречие в том, что не работает явное приведение к типу столбца таблицы:

```
1.INSERT INTO Test_char
```

```
2. VALUES (CAST('1111111111222222222233333333333' AS CHAR),
   '2');
```

PostgreSQL 8.3

chr	vchr
1	11111111112222222222333333333344444444445555555555
2	1111111111222222222233333333333
1	2
3	3

Можно отметить последовательность в поведении: VARCHAR имеет произвольный размер; вторая строка не была вставлена из-за ошибки превышения размера (ERROR: value too long for type character(1)); явное преобразование значения к типу столбца таблицы работает, отсекая лишние символы справа.

MySQL 5.0

Не поддерживается тип VARCHAR без указания размера строки. CHAR соответствует CHAR(1) – по стандарту. Поскольку явное преобразование к CHAR оставляет длину строки без изменения, то в таблицу, определенную как

```
1. CREATE TABLE Test_char( chr CHAR, vchr VARCHAR(1) );
```

в итоге, как и в SQL Server, добавится единственная строка:

chr	vchr
3	3

Выводы. По моему скромному мнению, ни одна из упомянутых СУБД не отвечает стандартному поведению в тех случаях, когда размер типа не указывается. Наиболее последовательной в «особенностях реализации» является, на мой взгляд, PostgreSQL. В целях переносимости кода я бы рекомендовал всегда явно указывать размер.

Рассмотрим следующий код.

```
1.DECLARE @ch AS VARCHAR(2)='1';
2.DECLARE @t TABLE(ch VARCHAR(2));
3.INSERT INTO @t VALUES('123');
4.SELECT @ch=CONCAT(@ch,'2','3');
5.SELECT @ch "var",(SELECT ch FROM @t) "col";
```

var	col
12	NULL

В этом коде мы определяем простую переменную типа VARCHAR(2) и таблицу (табличную переменную) с единственным столбцом того же типа данных - VARCHAR(2).

В результате данные в переменной усекаются до требуемого размера - 2 символа, а при вставке в столбец таблицы значения, превышающего допустимый размер, возникает ошибка:

Символьные или двоичные данные могут быть усечены.

Этим объясняется результат - NULL в столбце *col*, т.е. данные не были вставлены в таблицу.

По поводу символьных данных следует сделать еще одно замечание, которое касается их допустимого размера.

```
1.SELECT LEN(REPLICATE('a',100000)) "varchar"
2.,LEN(REPLICATE(CAST('a' AS NVARCHAR),100000)) "nvarchar"
3.,LEN(REPLICATE(CAST('c'     AS     VARCHAR(MAX)),100000))
  "varchar_max";
```

varchar	nvarchar	varchar_max
8000	4000	100000

Данный пример показывает, что простые символьные типы усекаются до размера страницы SQL Server - 8000 байтов (тип NVARCHARиспользует 2 байта на символ). Тип VARCHAR(MAX) позволяет хранить данные до 2Гб.

Это особенно следует иметь в виду при конкатенации данных, когда заранее трудно или невозможно узнать размер получаемой в результате строки, что может привести к неверному анализу данных:

```
1.SELECT  LEN(CONCAT(REPLICATE('a',  7000),  REPLICATE('b',
  7000))) ch;
```

ch
8000

Здесь мы соединяем две строки по 7000 символов в каждой. В результате получаем строку размером 8000 символов, остальное отсекается без получения сообщения об ошибке.

Float(n)

Как-то в одной социальной сети спросили, как убрать концевые нули в десятичных числах. Это было связано с

подготовкой некоего отчета, где денежные суммы конкатенировались с текстом. В постановке задачи указывалось ограничение на два десятичных знака. Вот пример данных и требуемый результат:

дано	получить
0.00	0
10.00	10
2.50	2.5
100.00	100
11.33	11.33

Предлагались решения, построенные на разборе строки. Я тоже впал в подобную ересь и предложил следующее решение.

```
1.SELECT num,
2.CASE WHEN CAST(num AS INT) = num
3.           THEN CAST(CAST(num AS INT) AS VARCHAR)
4.           WHEN CAST(num*10 AS INT) = num*10
5.           THEN LEFT(CAST(num AS VARCHAR), LEN(CAST(num
  AS VARCHAR)) - 1)
6.            WHEN CAST(num*100 AS INT)=num*100
7.            THEN CAST(num AS VARCHAR)
8.END fnum
9.FROM(
10.      SELECT 0.00 AS num
11.      UNION ALL SELECT 10.00
12.      UNION ALL SELECT 2.50
13.      UNION ALL SELECT 100
14.      UNION ALL SELECT 11.33
15.      ) X;
```

Не знаю, сколько бы это еще продолжалось, если бы один участник дискуссии не заметил, что все проблемы решает приведение к типу данных **float**. Действительно,

```
1.SELECT num, CAST(num AS FLOAT) fnum
2.FROM(
3.SELECT 0.00 AS num
4.UNION ALL SELECT 10.00
5.UNION ALL SELECT 2.50
6.UNION ALL SELECT 100
7.UNION ALL SELECT 11.33
8.) X;
```

Однако при этом нужно помнить о приближенном характере данного типа, а именно о величине мантиссы в научном представлении числа.

В соответствии со стандартом, этот тип данных имеет аргумент - FLOAT(n), который может принимать значения от 1 до 53. SQL Server значения аргумента в диапазоне 1 - 24 трактует как 24, что соответствует точности 7 цифр, а в диапазоне 25 - 53 как 53, что соответствует точности 15 цифр. По умолчанию принимается значение 53.

Продемонстрируем сказанное следующим примером.

```
1.SELECT num,
2.CAST(num AS FLOAT(24)) num_24,
3.CAST(num AS FLOAT(53)) num_53
4.FROM(
5.SELECT 1234567.80 AS num
6.UNION ALL SELECT  12345678.90
7.UNION ALL SELECT 123456789012345.60
8.UNION ALL SELECT 1234567890123456.70
9.) x;
```

num	num_24	num_53
1234567.80	1234568	1234567.8
12345678.90	1.234568E+07	12345678.9

123456789012345.60	1.234568E+14	123456789012346
1234567890123456.70	1.234568E+15	1.23456789012346E+15

MySQL (версия 5.0)

Не поддерживается преобразование к типу FLOAT.

PostgreSQL (версия 8.3.6)

Практически аналогичное поведение, за исключением того, что для параметра в диапазоне 1 – 24 точность составляет 6 цифр. Соответственно последние результаты будут выглядеть так:

num	num_24	num_53
1234567.80	1.23457e+006	1234567.8
12345678.90	1.23457e+007	12345678.9
123456789012345.60	1.23457e+014	123456789012346
1234567890123456.70	1.23457e+015	1.23456789012346e+015

Целочисленное деление

Иногда недоумение у начинающих работать с **SQL Server** вызывают результаты подобных запросов:

```
1.SELECT 1/3 AS a, 5/3 AS b;
```

Одни (подозреваю, что это пользователи MySQL или Oracle) ожидают результаты типа

a	b
0.3333	1.6667

т.е. вещественное число, другие –

a	b
0	2

т.е. округления к ближайшему целому. В то время как SQL Server дает

a	b
0	1

Чтобы развеять это недоумение, скажу, что операция "/" просто обозначает целочисленное деление (а именно, дает в результате неполное частное), если операнды являются целыми числами. Т.е. отдельного обозначения для этой операции нет, и используется символ "обычного" деления. Если же вы хотите получить десятичное число, то нужно привести хотя бы один операнд к вещественному типу явно (первый столбец) или неявно (второй столбец):

```
1. SELECT CAST(1 AS DEC(12,4))/3 AS a, 5./3 AS b;
```

a	b
0.333333	1.666666

Операция получения остатка от деления в SQL Server обозначается "%":

```
1.SELECT 1 % 3 AS a, 5 % 3 AS b;
```

a	b
1	2

Теперь что касается некоторых других СУБД.

PostgreSQL ведет себя аналогично SQL Server.

В **MySQL** для получения неполного частного используется специальный оператор **DIV**:

```
1.SELECT 1 DIV 3 AS a, 5 DIV 3 AS b;
```

Остаток от деления можно также получить в стиле а-ля Паскаль:

```
1.SELECT 1 MOD 3 AS a, 5 MOD 3 AS b;
```

Хотя будет работать и "общепринятое"

```
1.SELECT 1 % 3 AS a, 5 % 3 AS b;
```

В **Oracle** вообще нет операции для получения неполного частного, поэтому результат деления

```
1.SELECT 1/3 AS a, 5/3 AS b FROM dual;
```

a	b
.333333333333333	1.66666666666667

потребуется вручную приводить к целому типу с желаемым результатом, например, так:

```
1.SELECT CEIL(1/3) AS a, CEIL(5/3) AS b FROM dual;
```

a	b
1	2

или так

```
1.SELECT FLOOR(1/3) AS a, FLOOR(5/3) AS b FROM dual;
```

a	b
0	1

Для получения остатка от деления в Oracle используется функция **MOD**:

```
1.SELECT MOD(1,3) AS a, MOD(5,3) AS b FROM dual;
```

Наконец, если делитель равен нулю:

```
1.SELECT 1/0 AS a;
```

то MySQL возвращает NULL, в то время как другие рассматриваемые здесь СУБД дают ошибку деления на ноль.

Методы типа данных XML

Тип данных XML впервые появился в SQL Server 2005.

Он может содержать до 2 Гб данных.

В языке Transact-SQL имеется пять методов для работы с типом данных XML:

- query() – используется для извлечения XML фрагментов из XML документов;
- value() – используется для извлечения значений конкретных узлов или атрибутов XML документов;
- exist() – используется для проверки существования узла или атрибута. Возвращает 1, если узел или атрибут найден, и 0, если не найден;
- modify() – изменяет XML документ;
- nodes() – разделяет XML документ на несколько строк по узлам.

Методы типа данных XML принимают на вход выражение XPath или запрос XQuery.

Для примеров в данном учебнике будем использовать базу данных с таблицами, содержащими столбцы типа данных XML.

Таблица *tArtist* содержит информацию о музыкальных группах, исполнителях и их альбомах.

```
1.CREATE TABLE dbo.tArtist (
2.      artistId INT NOT NULL PRIMARY KEY
3.    , name VARCHAR(100) NOT NULL
4.    , xmlData XML NOT NULL
5.);
```

Заполним таблицы тестовыми данными.

```
1.INSERT INTO dbo.tArtist (artistId, name, xmlData) VALUES
2.(1, 'Radiohead',
3.'<albums>
4.    <album title="The King of Limbs">
5.        <labels>
6.            <label>Self-released</label>
7.        </labels>
8.        <song title="Bloom" length="5:15"/>
9.        <song title="Morning Mr Magpie" length="4:41"/>
10.              <song    title="Little    by    Little"
  length="4:27"/>
11.           <song title="Feral" length="3:13"/>
12.           <song title="Lotus Flower" length="5:01"/>
13.           <song title="Codex" length="4:47"/>
14.              <song    title="Give    Up    the    Ghost"
  length="4:50"/>
15.           <song title="Separator" length="5:20"/>
16.                                        <description
  link="http://en.wikipedia.org/wiki/The_King_of_Limbs">
17.           The King of Limbs is the eighth studio album
  by English rock band Radiohead,
18.            produced by Nigel Godrich. It was self-
  released on 18 February 2011 as a
19.           download in MP3 and WAV formats, followed
  by physical CD and 12" vinyl
20.            releases on 28 March, a wider digital
  release via AWAL, and a special
21.            "newspaper" edition on 9 May 2011. The
  physical editions were released
22.           through the band''s Ticker Tape imprint on
  XL in the United Kingdom,
```

```
23.                 TBD in the United States, and Hostess Entertainment in Japan.
24.             </description>
25.         </album>
26.         <album title="OK Computer">
27.             <labels>
28.                 <label>Parlophone</label>
29.                 <label>Capitol</label>
30.             </labels>
31.             <song title="Airbag" length="4:44"/>
32.             <song title="Paranoid Android" length="6:23"/>
33.             <song title="Subterranean Homesick Alien" length="4:27"/>
34.             <song title="Exit Music (For a Film)" length="4:24"/>
35.             <song title="Let Down" length="4:59"/>
36.             <song title="Karma Police" length="4:21"/>
37.             <song title="Fitter Happier" length="1:57"/>
38.             <song title="Electioneering" length="3:50"/>
39.             <song title="Climbing Up the Walls" length="4:45"/>
40.             <song title="No Surprises" length="3:48"/>
41.             <song title="Lucky" length="4:19"/>
42.             <song title="The Tourist" length="5:24"/>
43.             <description link="http://en.wikipedia.org/wiki/OK_Computer">
44.             OK Computer is the third studio album by the English alternative rock band
45.             Radiohead, released on 16 June 1997 on Parlophone in the United Kingdom and
46.             1 July 1997 by Capitol Records in the United States. It marks a deliberate
47.             attempt by the band to move away from the introspective guitar-oriented
48.             sound of their previous album The Bends. Its layered sound and wide range
49.             of influences set it apart from many of the Britpop and alternative rock
50.             bands popular at the time and laid the groundwork for Radiohead''s later,
51.             more experimental work.
52.             </description>
```

```
53.         </album>
54.     </albums>'),
55.     (2, 'Guns N'' Roses',
56.     '<albums>
57.         <album title="Use Your Illusion I">
58.             <labels>
59.                 <label>Geffen Records</label>
60.             </labels>
61.              <song title="Right Next Door to Hell"
   length="3:02"/>
62.                  <song title="Dust N'' Bones"
   length="4:58"/>
63.              <song title="Live and Let Die (Paul
   McCartney and Wings cover)"
64.             length="3:04"/>
65.             <song title="Don''t Cry (original version)"
   length="4:44"/>
66.             <song title="Perfect Crime" length="2:24"/>
67.               <song title="You Ain''t the First"
   length="2:36"/>
68.             <song title="Bad Obsession" length="5:28"/>
69.                  <song title="Back Off Bitch"
   length="5:04"/>
70.               <song title="Double Talkin'' Jive"
   length="3:24"/>
71.             <song title="November Rain" length="8:57"/>
72.              <song title="The Garden (featuring Alice
   Cooper and Shannon Hoon)"
73.             length="5:22"/>
74.                  <song title="Garden of Eden"
   length="2:42"/>
75.                  <song title="Don''t Damn Me"
   length="5:19"/>
76.             <song title="Bad Apples" length="4:28"/>
77.             <song title="Dead Horse" length="4:18"/>
78.             <song title="Coma" length="10:13"/>
79.                                         <description
   link="http://ru.wikipedia.org/wiki/Use_Your_Illusion_I">
80.              Use Your Illusion I is the third studio
   album by GnR. It was the first of two
81.             albums released in conjunction with the Use
   Your Illusion Tour, the other
82.             being Use Your Illusion II. The two are thus
   sometimes considered a double album.
```

```
83.                 In fact, in the original vinyl releases,
  both Use Your Illusion albums are
84.                 double albums. Material for all two/four
  discs (depending on the medium) was
85.                 recorded at the same time and there was some
  discussion of releasing a
86.                 ''quadruple album''. The album debuted at
  No. 2 on the Billboard charts, selling
87.                 685,000 copies in its first week, behind
  Use Your Illusion II''s first week sales
88.                 of 770,000. Use Your Illusion I has sold
  5,502,000 units in the U.S. as of 2010,
89.                 according to Nielsen SoundScan. It was
  nominated for a Grammy Award in 1992.
90.                 </description>
91.             </album>
92.             <album title="Use Your Illusion II">
93.                 <labels>
94.                     <label>Geffen Records</label>
95.                 </labels>
96.                 <song title="Civil War" length="7:42"/>
97.                 <song title="14 Years" length="4:21"/>
98.                 <song title="Yesterdays" length="3:16"/>
99.                 <song title="Knockin'' on Heaven''s Door
  (Bob Dylan cover)" length="5:36"/>
100.                <song title="Get in the Ring"
  length="5:41"/>
101.                <song title="Shotgun Blues" length="3:23"/>
102.                <song title="Breakdown" length="7:05"/>
103.                <song title="Pretty Tied Up"
  length="4:48"/>
104.                <song title="Locomotive (Complicity)"
  length="8:42"/>
105.                <song title="So Fine" length="4:06"/>
106.                <song title="Estranged" length="9:24"/>
107.                <song title="You Could Be Mine"
  length="5:43"/>
108.                <song title="Don''t Cry (Alternate lyrics)"
  length="4:44"/>
109.                <song title="My World" length="1:24"/>
110.                <description
  link="http://ru.wikipedia.org/wiki/Use_Your_Illusion_II"
  >
111.                Use Your Illusion II is the fourth studio
  album by GnR. It was one of two albums
```

```
112.                released in conjunction with the Use Your
    Illusion Tour, and as a result the two
113.                albums are sometimes considered a double
    album. Bolstered by lead single ''You
114.                Could Be Mine'', Use Your Illusion II was
    the slightly more popular of the two
115.                albums, selling 770,000 copies its first
    week and debuting at No. 1 on the U.S.
116.                charts, ahead of Use Your Illusion I''s
    first week sales of 685,000.
117.            </description>
118.        </album>
119.    </albums>');
```

Метод query()

В общем случае этот метод принимает на вход выражение XPath и возвращает новый XML документ.

Выражение XPath '/albums/album[2]/labels/label[1]' указывает, что мы хотим получить первый лейбл второго альбома для каждого исполнителя.

Метод **query()** возвращает фрагмент XML документа, содержащий всё между начальным и конечным тегами элемента "label", включая и сам элемент.

```
1.SELECT name,
  xmlData.query('/albums/album[2]/labels/label[1]') AS
  SecondAlbumLabel
2.FROM dbo.tArtist;
```

name	SecondAlbumLabel
Radiohead	Parlophone
Guns N' Roses	Geffen Records

Рассмотрим другой пример. Нужно найти такие альбомы, в описании которых есть слово "record".

```
1. SELECT name,
2. xmlData.query ('/albums/album[description[contains(.,
   "record")]]') AS ContainsRecord
3. FROM dbo.tArtist;
```

Если детально разобрать выражение XPath, то можно его описать следующей фразой: найти в списке альбомов (albums) такие альбомы (album), у которых имеется описание (description), содержащее слово "record" (contains(., "record")). Точка в функции contains() означает обращение к содержимому текущего элемента, в данном случае это элемент description.

name	ContainsRecord
Radiohead	
Guns N' Roses	…

Видим, что такой альбом нашёлся только у группы Guns N' Roses. Но у группы Radiohead в описании альбома OK Computer встречается слово "Records". Этот альбом не нашёлся, т.к. XQuery чувствителен к регистру. Чтобы данный альбом тоже был найден, используем функцию lower-case(), приводящую обрабатываемый текст к нижнему регистру.

```
1. SELECT name,
2. xmlData.query
   ('/albums/album[description[contains(lower-case(.),
   "record")]]')
3. AS ContainsRecord
4. FROM dbo.tArtist;
```

name	ContainsRecord
Radiohead	…
Guns N' Roses	…

Метод value()

Метод value() позволяет извлекать из XML документа содержимое единичного элемента или атрибута с указанием его типа данных.

Если выражение XPath указывает на несколько узлов, то будет возвращено сообщение об ошибке. Например,

```
1.SELECT name,
2.xmlData.value('/albums/album[2]/labels/label[1]/text()',
  'varchar(100)')
3.AS SecondAlbumLabel
4.FROM dbo.tArtist;
```

*XQuery [dbo.tArtist.xmlData.value()]: 'value()' requires a singleton (or empty sequence), found operand of type 'xdt:untypedAtomic *'*

Эта ошибка сообщает, что данное XPath-выражение может вернуть более одной сущности. Например, если бы в исходном XML документе было несколько элементов "albums" или более одного элемента "labels".

Чтобы исправить эту ошибку, замените выражение XPath на одно из следующих:

- '/albums[1]/album[2]/labels[1]/label[1]/text()[1]'
- '(/albums/album[2]/labels/label/text())[1]'

Результат будет одинаковым:

name	SecondAlbumLabel
Radiohead	Parlophone
Guns N' Roses	Geffen Records

Ещё один пример с типом данных "time". Допустим, мы хотим получить для каждого исполнителя название первого альбома, первой песни из этого альбома и её длительность.

```
1.SELECT name
2.        ,     xmlData.value('/albums[1]/album[1]/@title',
  'varchar(100)') AS FirstAlbum
3.                                                             ,
  xmlData.value('/albums[1]/album[1]/song[1]/@title',
  'varchar(100)') AS FirstSongTitle
4.                                                             ,
  xmlData.value('/albums[1]/album[1]/song[1]/@length',
  'time(0)') AS FirstSongLength
5.FROM dbo.tArtist;
```

name	FirstAlbum	FirstSongTitle	FirstSongLength
Radiohead	The King of Limbs	Bloom	05:15:00
Guns N' Roses	Use Your Illusion I	Right Next Door to Hell	03:02:00

К сожалению, поле "FirstSongLength" распознaлось в формате *чч:мм:сс*, тогда как мы уверены, что в XML-документе время записано в формате *мм:сс*. Чтобы исправить эту ошибку, можно использовать обычный синтаксис T-SQL, а можно воспользоваться XQuery:

```
1.SELECT name AS artist
2.       ,      xmlData.value('/albums[1]/album[1]/@title',
  'varchar(100)') AS FirstAlbum
3.   ,  xmlData.value('/albums[1]/album[1]/song[1]/@title',
  'varchar(100)') AS FirstSongTitle
4.                ,              xmlData.value('concat("00:",
  /albums[1]/album[1]/song[1]/@length)', 'time(0)')
5.AS FirstSongLength
6.FROM dbo.tArtist;
```

artist	FirstAlbum	FirstSongTitle	FirstSongLength
Radiohead	The King of Limbs	Bloom	00:05:15
Guns N' Roses	Use Your Illusion I	Right Next Door to Hell	00:03:02

Метод exist()

Данный метод не используется для получения каких-либо данных из XML документа. Он используется только для того, чтобы убедиться, что в нём присутствуют требуемые данные. Метод exist() проверяет существование указанного выражения XPath.

В данном примере метод exist() будет использован для определения, есть ли определённая песня в списке композиций. Пусть песня, которую мы ищем, называется "Garden of Eden".

```
1. SELECT name, xmlData.exist('/albums[1]/album/song[@title="Garden of Eden"]') AS SongExists
2. FROM dbo.tArtist;
```

name	SongExists
Radiohead	0
Guns N' Roses	1

Как можно видеть, песня с таким названием присутствует среди композиций группы Guns N' Roses (1 эквивалентно 'True'), и отсутствует среди композиций группы Radiohead (0 эквивалентно 'False').

Теперь проверим наличие песни длительностью более 10 минут.

```
1. SELECT name
2.     ,xmlData.exist('/albums[1]/album/song[@length>"10:00"]') AS LongSongExists
3. FROM dbo.tArtist;
```

name	SongExists
Radiohead	1
Guns N' Roses	1

Из полученного результата мы видим, что оба исполнителя имеют в своём репертуаре как минимум одну песню длительностью более 10 минут. Но это не так. На самом деле все песни группы Radiohead длятся менее 10 минут.

Проблема заключается в том, что значение атрибута @length по умолчанию принимает строковый тип данных. Получается, что мы сравнивали строки, а не время.

Чтобы решить эту проблему, мы можем преобразовать значение атрибута @length и значение, с которым оно сравнивается, к типу данных time. Оба они должны быть приведены к формату "чч:мм:сс" до преобразования типов данных. Для этого будем использовать функцию xs:time(), принимающую на вход строку и возвращающую время.

```
1. SELECT name
2.     , xmlData.exist('
3.         /albums[1]/album/song/@length[
4.                                         (
5.                                         if (string-length(.)
   = 4)
6.                                                             then
   xs:time(concat("00:0", .))
7.                                                             else
   xs:time(concat("00:", .))
8.                                         )
9.                                         > xs:time("00:10:00")
10.                                                     ]')   AS
   LongSongExists
11.       FROM dbo.tArtist;
```

Теперь мы видим, что в репертуаре группы Radiohead нет ни одной песни длительностью более 10 минут, в то время как в репертуаре группы Guns N' Roses есть как минимум одна такая песня.

name	SongExists
Radiohead	0
Guns N' Roses	1

В данном примере функция time() используется в пространстве имён xs. Все встроенные функции преобразования типов должны использоваться в этом пространстве имён (xs:string, xs:boolean, xs:decimal, xs:float, xs:double, xs:dateTime, xs:time, xs:date и т.д.). Прочие встроенные функции используются в пространстве имён fn, но его указание не обязательно. То есть string-length(.) и fn:string-length(.) эквивалентны.

Метод modify()

Метод modify() позволяет изменять значения непосредственно в XML-документе.

Так же, как и другие методы, он требует выражение XPath, чтобы понять, какое значение нужно изменить. Однако, в отличие от других методов, modify() работает с оператором UPDATE (и не работает с оператором SELECT). Также, modify() может работать только с одним значением за раз. В математике и программировании это называется одноэлементным множеством.

Так как не существует никакого ограничения на количество элементов, содержащихся внутри другого элемента, выражение XPath может возвращать много дочерних элементов. Например, если имеется такой XML документ:

```
<week>
  <day>Monday</day>
  <day>Tuesday</day>
  <day>Wednesday</day>
</week>
```

выражение XPath /week/day возвращает три элемента, которые не являются одноэлементным множеством:

```
<day>Monday</day>
<day>Tuesday</day>
<day>Wednesday</day>
```

Однако, если заменить выражение XPath на (/week/day)[1], то будет возвращён только один элемент "Monday".

```
<day>Monday</day>
```

Попробуем получить название первого лейбла альбома "OK Computer" группы "Radiohead".

```
1. SELECT xmlData.query('(/albums/album[@title="OK Computer"]/labels/label/text())[1]')
2. AS FirstLabelText
3. FROM dbo.tArtist
4. WHERE name = 'Radiohead';
```

FirstLabelText
Parlophone

Метод modify() имеет единственной целью изменение значений в XML-документе, что является полезной возможностью. Представьте, что XML-документ уже импортирован на SQL Server, и обнаружилась опечатка или нужно обновить только одно значение. Чтобы не загружать заново уже исправленный XML-документ, можно просто использовать метод modify() и изменить необходимые значения непосредственно в сохранённом XML-документе на SQL Server.

В следующем примере мы поменяем местами названия лейблов альбома "OK Computer" группы "Radiohead". Первый лейбл должен быть заменён на "Capitol", а второй – на "Parlophone".

Метод modify() может быть использован в предложении SET оператора UPDATE по отношению к переменной или столбцу типа данных XML.

```
1. UPDATE dbo.tArtist
2. SET xmlData.MODIFY('replace value of
3. (/albums/album[@title="OK
   Computer"]/labels/label/text())[1] with "Capitol"')
4. WHERE name = 'Radiohead';

1. UPDATE dbo.tArtist
```

```
2.SET xmlData.MODIFY('replace value of
3.(/albums/album[@title="OK
  Computer"]/labels/label/text())[2] with "Parlophone"')
4.WHERE name = 'Radiohead';
```

Отлично – исправленный запрос отработал корректно, что подтверждается сообщениями:

(1 row(s) affected)

(1 row(s) affected)

Теперь вернёмся к исходному запросу (с использованием оператора SELECT) и выполним его, чтобы проверить, что названия лейблов были обновлены корректно.

```
1.SELECT            xmlData.query('/albums/album[@title="OK
  Computer"]/labels')
2.FROM dbo.tArtist
3.WHERE name = 'Radiohead';
```

```
<labels>
  <label>Capitol</label>
  <label>Parlophone</label>
</labels>
```

Эта задача может быть решена по-другому. Можно поменять местами элементы "label" без изменения их значений. Просто вставим копию первого лейбла в конец списка лейблов.

```
1.UPDATE dbo.tArtist
2.SET   xmlData.MODIFY('insert   (/albums/album[@title="OK
  Computer"]/labels/label)[1] as last
3.into (/albums/album[@title="OK Computer"]/labels)[1]')
4.WHERE name = 'Radiohead';
```

После чего удалим первый лейбл.

```
1.UPDATE dbo.tArtist
2.SET   xmlData.MODIFY('delete   (/albums/album[@title="OK
  Computer"]/labels/label)[1]')
3.WHERE name = 'Radiohead';
```

Чтобы удостовериться, что лейблы поменялись местами корректно, выполним следующий запрос:

```
1.SELECT           xmlData.query('/albums/album[@title="OK
  Computer"]/labels')
2.FROM dbo.tArtist
3.WHERE name = 'Radiohead';
```

```
<labels>
  <label>Parlophone</label>
  <label>Capitol</label>
</labels>
```

Теперь попробуем отметить песню "Perfect Crime" из альбома "Use Your Illusion I" группы Guns N' Roses, как популярную. Для этого добавим в XML-документ в соответствующий элемент атрибут isPopular со значением 1.

```
1.UPDATE dbo.tArtist
2.SET xmlData.MODIFY('
3.    insert attribute isPopular { "1" }
4.     into  (/albums[1]/album[@title="Use  Your  Illusion
  I"]/song[@title="Perfect Crime"])[1]
5.')
6.WHERE name = 'Guns N'' Roses';
```

В качестве ещё одного примера добавим информацию о дате выпуска и дате записи песни "Estranged" из альбома "Use Your Illusion II" группы Guns N' Roses. Обратите внимание, что в данном примере в элемент добавляется сразу два атрибута.

```
1.UPDATE dbo.tArtist
2.SET xmlData.MODIFY('
3.    insert (
4.        attribute Recorded { "1991" },
5.        attribute Released { "1994-01-17" }
6.    )
7.     into (/albums[1]/album[@title="Use Your Illusion
  II"]/song[@title="Estranged"])[1]
8.')
9.WHERE name = 'Guns N'' Roses';
```

Метод nodes()

Разбивает XML-структуру на одно или несколько поддеревьев, в соответствии с указанным выражением XPath.

Выполним следующий запрос:

```
1.SELECT name AS artist
2.    , album.query('.') AS album
3.FROM dbo.tArtist A
4.CROSS APPLY
  A.xmlData.nodes('/albums[1]/album')col(album);
```

Данный запрос разобьет исходную структуру на строки, по количеству элементов "album" и вернет по две строки для каждого исполнителя:

artist	album
Radiohead	<album title="The King of Limbs"><labels><label>S...

Radiohead	<album title="OK Computer"><labels><label>Parlop...
Guns N' Roses	<album title="Use Your Illusion I"><labels><label>G...
Guns N' Roses	<album title="Use Your Illusion II"><labels><label>G...

Разберем текст запроса подробнее.

CROSS APPLY A.xmlData.nodes('/albums[1]/album') – разбивает каждую строку таблицы на столько строк, сколько элементов "album" было найдено.

col – это имя производной таблицы, а album – это имя столбца. Они нужны будут для дальнейшей работы с результатами.

album.query('.') – здесь осуществляется запрос к каждой строке результатов, при помощи псевдонима. Данный подзапрос просто выбирает все данные из поддерева.

Разберём другой пример. Допустим мы хотим получить в виде таблицы первые две песни каждого альбома для каждой из групп.

```
1.SELECT name AS artist
2.     , song.value('../@title', 'varchar(50)') AS album
3.     , song.value('@title', 'varchar(100)') AS song
4.FROM dbo.tArtist A
5.CROSS APPLY
  A.xmlData.nodes('/albums[1]/album/song[position()<=2]')c
  ol(song);
```

artist	album	song
Radiohead	The King of Limbs	Bloom
Radiohead	The King of Limbs	Morning Mr Magpie

Radiohead	OK Computer	Airbag
Radiohead	OK Computer	Paranoid Android
Guns N' Roses	Use Your Illusion I	Right Next Door to Hell
Guns N' Roses	Use Your Illusion I	Dust N' Bones
Guns N' Roses	Use Your Illusion II	Civil War
Guns N' Roses	Use Your Illusion II	14 Years

[position()<=2] – указывает, что нам нужны только два первых элемента "song" внутри каждого элемента "album".

'../@title' – обращается к родительскому элементу и берёт его атрибут @title.

Результат отобразился корректно, но при больших объёмах данных такой запрос будет крайне медленным и неэффективным. Причина заключается в том, что для каждой песни среда выполнения запроса ищет родительский элемент и считывает его атрибут. Перепишем запрос следующим образом:

```
1.SELECT name AS artist
2.    , album.value('@title', 'varchar(50)') AS album
3.    , song.value('@title', 'varchar(100)') AS song
4.FROM dbo.tArtist A
5.CROSS APPLY A.xmlData.nodes('/albums[1]/album')c1(album)
6.CROSS APPLY
  c1.album.nodes('song[position()<=2]')c2(song);
```

Результат будет аналогичным, однако теперь логика запроса изменена следующим образом: сначала выбираются все альбомы, а потом для каждого альбома будут присоединяться песни.

Поскольку песен всегда будет намного больше, чем альбомов, выполнение данного запроса покажет существенный прирост производительности по сравнению с предыдущим.

Язык определения данных (SQL DDL)

Язык определения данных (Data Definition Language) предназначен для создания, изменения и удаления объектов базы данных. Основными объектами реляционной базы данных являются таблицы. С них и начнем.

Таблицы бывают базовые (постоянные) и временные. Временные таблицы существуют в течение сеанса пользователя, в котором он их создал. Если в этом сеансе таблицы не удаляются явно, то они будут удалены автоматически по завершении сеанса. Базовые таблицы предназначены для долговременного хранения информации в базе данных.

Создание базовых таблиц

Базовые таблицы создаются оператором **CREATE TABLE**:

```
1. CREATE TABLE <имя таблицы>(<список спецификаций столбцов
   и ограничений>);
```

Спецификация столбца включает имя столбца и тип данных значений, которые могут находиться в этом столбце. Кроме того, некоторые ограничения могут быть заданы не только отдельными спецификациями, но и в спецификации столбца. Примерами могут служить ограничения первичного и внешнего ключей (простых, не составных), а также ограничение NOT NULL.

Изучать аспекты языка, имеющие отношение к таблицам, мы будем на примерах таблиц учебных баз данных.

В таблице Product из схемы «Компьютерная фирма» имеется три столбца – *maker, model, type* все строкового типа данных **VARCHAR**(N). Чтобы создать эту таблицу мы можем написать следующий оператор:

```
1. CREATE TABLE Product (maker varchar(10), model
   varchar(50), type varchar(50));
```

Значение N указывает максимальное число символов, которое могут содержать данные в данном столбце. VARCHAR является переменным типом, это означает, что если мы зададим значение с числом символов меньше N, то записано на диск будет именно заданное количество символов. Альтернативой служит точный строковый тип **CHAR**; для него отсутствующие символы дополняются пробелами справа, т.е. на диск будет всегда записано N символов.

Если значение N не указано, то по умолчанию подразумевается 1, т.е. один символ.

Как только таблица создана, в нее могут быть помещены данные с помощью оператора INSERT. Сделаем это:

```
1. INSERT INTO product VALUES
2. ('A',     '1232',     'PC'),
3. ('A',     '1232',     'Printer'),
4. (NULL,     NULL,     NULL);
```

Данные успешно вставлены, но какие-то они неправильные. Во-первых, непонятно чем является модель 1232 – принтером или ПК? Во-вторых, имеется у нас еще одна модель, о которой вообще ничего неизвестно.

Здесь следует сделать небольшое отступление, чтобы сказать о том, что, создавая таблицы, мы создаем реляционную модель предметной области. Нашей предметной областью является учет товаров в компьютерной фирме. Чтобы модель данных была адекватна предметной области, требуется так спроектировать таблицы, чтобы то, что происходит с объектами в реальном мире, могло быть отражено в модели, а то, чего не может быть в предметной области, не должно иметь места и в модели.

Итак, в реальном мире модель не может быть одновременно и принтером, и ПК, а у нас это получилось. И тут мы приходим к понятию **целостности данных** и ее реализации посредством ограничений.

Категорная целостность или целостность сущностей

Категорная целостность означает, что каждый объект, определяемый строкой в таблице, должен быть отличим от любого другого объекта. Иными словами, должен быть такой набор атрибутов, уникальная комбинация значений которого позволит нам отличить один объект от другого. Такой набор атрибутов, однозначно идентифицирующий объект, называется потенциальным ключом. Он не может содержать NULL-значения, поскольку это не даст нам возможности идентифицировать объект (как, скажем, книгу с неизвестным названием).

Потенциальных ключей у таблицы может быть несколько. Например, человека можно идентифицировать по номеру паспорта, номеру страхового свидетельства, ИНН, номеру водительских прав и т.д.

Для обеспечения категорной целостности в языке SQL существуют спецификации **PRIMARY KEY** (первичный ключ) и **UNIQUE** (уникальный ключ). Первичный ключ может быть только один в таблице, уникальных же ключей может быть несколько. Т.е. у нас есть возможность для одного из потенциальных ключей задать спецификацию PRIMARY KEY, а для остальных – UNIQUE.

Что в нашем случае может послужить первичным ключом? Поскольку у одного производителя может быть много моделей, и он, соответственно, неоднократно

может присутствовать в данных таблицы Product, то столбец *maker* не может являться кандидатом на роль первичного ключа. Аналогично свойством уникальности не обладает и атрибут *type*.

Уникальным же является номер модели. Это единственный кандидат на роль первичного ключа. Других потенциальных ключей в таблице нет. Чтобы доказать это, можно рассмотреть все остальные комбинации столбцов и показать, что они не гарантируют нам идентификации объекта. В частности, комбинация значений из трех столбцов в приведенном выше примере данных является уникальной, но, тем не менее, не идентифицирует модель 1232.

Давайте создадим первичный ключ. В языке SQL есть возможность изменить структуру существующей таблицы при помощи команды **ALTER TABLE**. Однако давайте будем дозировать новую информацию, поэтому сейчас будет проще пересоздать таблицу, т.е. удалить ее и создать заново с первичным ключом. Удалить таблицу просто (как говорится, ломать – не строить), для этого достаточно выполнить оператор **DROP TABLE** <имя таблицы>. Итак,

```
1.DROP TABLE Product;
2.CREATE TABLE Product (
3.maker varchar(10),
4.model varchar(50) PRIMARY KEY,
5.type varchar(50)
6.);
```

Мы включили спецификацию первичного ключа в определение столбца. Но можно было это сделать отдельным ограничением:

```
1.CONSTRAINT <имя ограничения> PRIMARY KEY (<список
  столбцов, являющихся первичным ключом>)
```

При этом наш код будет выглядеть так (опять предварительно удаляем ранее созданную таблицу

```
1.DROP TABLE Product;
2.CREATE TABLE Product (
3.maker varchar(10),
4.model varchar(50),
5.type varchar(50),
6.CONSTRAINT product_PK PRIMARY KEY (model)
7.);
```

Теперь сама СУБД будет следить за тем, чтобы значения первичного ключа не повторялись и не содержали NULL. Если мы выполним вставку при помощи ранее приведенного оператора INSERT, то получим такое сообщение об ошибке:

Violation of PRIMARY KEY constraint 'product_PK'. Cannot insert duplicate key in object 'Product'. The duplicate key value is (1232).

(Нарушение ограничения первичного ключа 'product_PK'. Нельзя вставить дубликат ключа в объект 'Product’. Дублирующееся значение ключа (1232).)

Если мы исправим ошибку ввода и укажем правильный номер модели принтера:

```
1.INSERT INTO product VALUES
2.('A',    '1232',    'PC'),
3.('A',    '3001',    'Printer'),
4.(NULL,    NULL,    NULL);
```

То получим другую ошибку, связанную с неопределенностью значения первичного ключа:

Cannot insert the value NULL into column 'model', table 'Product'; column does not allow nulls. INSERT fails.

(Невозможно вставить NULL в столбец 'model', table 'Product'; столбец не допускает NULL-значения. INSERT не выполнен.)

Оператор

```
1. INSERT INTO product VALUES
2. ('A',    '1232',    'PC'),
3. ('A',    '3001',    'Printer'),
4. (NULL,    '2000',    NULL);
```

отработает без ошибок.

Как вы должно быть заметили, в сообщении об ошибке нарушения ограничения первичного ключа фигурирует имя ограничения ('product_PK'). А что будет в случае, если имя не задано? Так было у нас, когда спецификация **PRIMARY KEY** была включена в определение столбца *model*. Кстати, во втором варианте мы тоже можем не указывать имя (тогда и ключевое слово **CONSTRAINT** также опускается

```
1. DROP TABLE Product;
2. CREATE TABLE Product (
3. maker varchar(10),
4. model varchar(50),
5. type varchar(50),
6. PRIMARY KEY (model)
7. );
```

И так ли важно нам знать имя ограничения? Помимо того, что оно может помочь нам понять причину ошибки, имя требуется при удалении ограничения, когда мы меняем структуру существующей таблицы. Если мы не задаем имя ограничения, СУБД сама присвоит его. И это имя, которое должно быть уникальным в пределах базы данных, мы можем узнать из информационной схемы – стандартного представления метаданных. Например, чтобы узнать имя ограничения первичного ключа, созданного в последнем скрипте, можно выполнить обычный запрос на выборку из таблицы (представления) информационной схемы:

```
1. SELECT CONSTRAINT_NAME
2. FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS
3. WHERE TABLE_NAME='Product' AND CONSTRAINT_TYPE ='primary
   key';
```

У меня это имя получилось таким: PK__Product__0B7E269E30F848ED. У вас, вероятно, будет другим, поскольку его генерирует СУБД.

Примером составного ключа может послужить первичный ключ в таблице Outcomes (база данных «Корабли»). Здесь только пара {*корабль, сражение*} может быть уникальной, поскольку корабль может принять участие в нескольких сражениях, а в одном сражении участвует несколько кораблей. Корабль же в отдельной битве не может быть упомянут дважды, что и запретит сделать первичный ключ. Создать таблицу Outcomes с упомянутым первичным ключом можно следующим образом:

```
1.CREATE TABLE Outcomes (
2.ship varchar(50),
3.battle varchar(20),
4.result varchar(10),
5.PRIMARY KEY(ship, battle)
6.);
```

Следует заметить, что мы не можем написать так:

```
1.CREATE TABLE Outcomes (
2.ship varchar(50) PRIMARY KEY,
3.battle varchar(20) PRIMARY KEY,
4.result varchar(10)
5.);
```

поскольку спецификация PRIMARY KEY может быть только одна.

Если первичный ключ может быть только один в таблице, то как быть в том случае, если в нашей модели должны быть уникальны разные комбинации атрибутов? Другими словами, как создать альтернативные ключи, например, если нам потребуется добавить уникальный столбец *out_id* в таблицу Outcomes?

Для этой цели в языке SQL имеется спецификация **UNIQUE**. Вот так мог бы выглядеть запрос на создание таблицы Outcomes с дополнительным столбцом *out_id*:

```
1. CREATE TABLE Outcomes (
2. ship varchar(50),
3. battle varchar(20),
4. result varchar(10),
5. out_id int,
6. PRIMARY KEY(ship, battle),
7. UNIQUE (out_id)
8. );
```

Как уже упоминалось выше, ограничений UNIQUE может быть создано несколько для одной таблицы.

Есть еще одно отличие этого ограничения от ограничения PRIMARY KEY. Столбец, на котором создано ограничение UNIQUE, может содержать NULL-значение, но только одно. Вы можете спросить, - как же в этом случае быть с идентификацией объекта? Вспомним пример с книгами, когда мы ищем книгу по уникальному названию. Пусть мы имеем ограничение UNIQUE на столбце названия книги. Поскольку NULL-значение может быть только одно, то найти книгу с неизвестным названием мы можем, исключив все книги с известными названиями.

Если же нам потребуется альтернативный ключ, не допускающий NULL-значений, то совместно с UNIQUE мы можем наложить ограничение NOT NULL. К этому мы сейчас и переходим.

Проверочные ограничения

Проверочное ограничение имеет вид:

```
1. CHECK (<предикат>)
```

Предикат может принимать значения TRUE, FALSE или UNKNOWN. Предикат считается нарушенным, когда он принимает значение FALSE; при

этом действие, которое привело к нарушению предиката, не выполняется, и генерируется сообщение об ошибке.

Вернемся к нашей проблеме, а именно вставке строки (NULL, '2000', NULL) в таблицу Product. Понятно, что подобных строк следует избегать, т.к. неизвестно, чем является модель 2000 и каким производителем она выпускается. Мы можем использовать предикат IS NOT NULL в проверочных ограничениях для столбцов maker и type:

```
1.DROP TABLE Product;
2.CREATE TABLE Product (
3.maker varchar(10),
4.model varchar(50),
5.type varchar(50),
6.CONSTRAINT product_PK PRIMARY KEY (model),
7.CONSTRAINT maker_ch CHECK(maker IS NOT NULL),
8.CONSTRAINT type_ch CHECK(type IS NOT NULL)
9.);
```

Теперь при попытке вставить строку (NULL, '2000', NULL) мы получим сообщение об ошибке:

Cannot insert the value NULL into column 'model', table 'Product'; column does not allow nulls. INSERT fails.

(Значение NULL не может быть вставлено в столбец model таблицы Product; столбец не допускает NULL-значений. INSERT не выполнен)

И строка вставлена не будет, вернее, будет отклонен весь оператор INSERT.

Заметим, что ограничение **NOT NULL** (как и PRIMARY KEY для простого ключа) может быть записано непосредственно в определение столбца:

```
1.DROP TABLE Product;
2.CREATE TABLE Product (
3.maker varchar(10) NOT NULL,
4.model varchar(50) PRIMARY KEY,
5.type varchar(50) NOT NULL
6.);
```

Более того, любое ограничение уровня столбца можно записать непосредственно в определении столбца. Ниже приводится пример, добавляющий новый столбец (*available*) в таблицу Product:

```
1.ALTER TABLE Product ADD available VARCHAR(20) DEFAULT
  'Yes';
```

Чтобы ограничить допустимые значения в данном столбце значениями 'Yes' и 'No', этот оператор можно было бы переписать в виде:

```
1.ALTER TABLE Product ADD available VARCHAR(20)
2.CHECK(available IN('Yes','No')) DEFAULT 'Yes';
```

Оператор ALTER TABLE

Можно выделить следующие уровни проверочных ограничений:

- уровень атрибута (столбца),
- уровень кортежа (строки),
- уровень отношения (таблицы).

В ограничении уровня столбца проверяется значение только одного отдельного столбца, другими словами, в ограничении данного типа имеется ссылка только на один столбец той таблицы, в определении которой содержится данное ограничение. Чтобы привести пример такого ограничения, вернёмся к схеме «Компьютерная фирма». В таблице Product в столбце *type* может находиться одно из трех значений. Мы можем запретить ввод любой другой информации в этот столбец при помощи такого ограничения:

```
1.CHECK (type IN('printer', 'pc', 'laptop'))
```

Давайте сделаем отступление, чтобы познакомиться с оператором **ALTER TABLE**, который позволит нам изменять структуру таблицы, не пересоздавая её всякий раз заново. Это тем более важно, что изменение структуры может потребоваться тогда, когда таблица уже содержит данные.

С помощью оператора ALTER TABLE можно добавить или удалить столбцы, значения по умолчанию, а также ограничения.

В настоящий момент нас интересует добавление ограничения на столбец *type*, поэтому вначале приведём синтаксис оператора для добавления ограничения:

```
1.ALTER TABLE <имя таблицы>
2.ADD CONSTRAINT <имя ограничения> <ограничение>;
```

Давайте теперь добавим наше ограничение и проверим, как оно работает.

```
1.ALTER TABLE Product
2.ADD CONSTRAINT chk_type CHECK (type IN('pc', 'laptop',
  'printer'));
```

Чтобы убедиться в том, что ограничение работает как мы того ожидаем, попробуем добавить модель нового типа:

```
1.INSERT INTO Product VALUES('A', 1122, 'notebook');
```

Как и ожидалось, в ответ мы получим сообщение об ошибке:

The INSERT statement conflicted with the CHECK constraint "chk_type". The conflict occurred in database "learn", table "dbo.product", column 'type'. The statement has been terminated.

(Конфликт инструкции INSERT с ограничением CHECK "chk_type". Конфликт произошел в базе данных "learn", таблица "dbo.product", столбец 'type'. Выполнение данной инструкции было прервано.)

Как легко догадаться, ограничение уровня строки содержит ссылки на несколько столбцов. При этом ограничение проверяется для каждой изменяемой строки отдельно. Строка может быть добавлена (или изменена), если ограничение не нарушено.

В качестве примера давайте запретим производителю Z выпускать что-либо помимо принтеров.

```
1.ALTER TABLE Product
2.ADD constraint chk_maker_Z CHECK ((maker='Z' AND type=
  'printer') OR maker <>'Z');
```

Итак, ограничение проверяет, что модель в таблице Product должна быть принтером производителя Z (maker='Z' and type= 'printer') или любого другого производителя (но не Z).

Если мы попытаемся добавить модель ПК производителя Z,

```
1.INSERT INTO Product VALUES('Z', 1122, 'PC');
```

то получим приведенное выше сообщение об ошибке, в котором вместо имени ограничения chk_type будет упомянуто ограничение chk_maker_Z. При этом модель принтера будет добавлена без проблем:

```
1.INSERT INTO Product VALUES('Z', 1122, 'Printer');
```

Разумеется, другой производитель сможет выпускать все, что угодно:

```
1.INSERT INTO Product VALUES('V', 1123, 'PC');
```

Значения по умолчанию

Для столбца может быть задано значение по умолчанию, т.е. значение, которое будет подставляться в том случае, когда оператор вставки не предоставляет значения для этого столбца. Как правило, значением по умолчанию выбирается наиболее часто встречающееся значение.

Пусть для нашей базы данных наибольшая часть моделей представляет собой ПК. Давайте установим для столбца *type* значение по умолчанию ‘PC’. Добавить значение по умолчанию можно с помощью оператора **ALTER TABLE**. Согласно стандарту, оператор для нашего примера имел бы вид:

```
1.ALTER TABLE Product
2.ALTER COLUMN "type" SET DEFAULT 'PC';
```

Однако **SQL Server** не поддерживает в данном случае стандартный синтаксис; в диалекте **T-SQL** аналогичную операцию можно выполнить так:

```
1.ALTER TABLE Product
2.ADD DEFAULT 'PC' FOR type;
```

Теперь при добавлении в таблицу Product модели ПК мы можем не указывать тип.

```
1.INSERT INTO Product(maker, model) VALUES('A', '1124');
```

Заметим, что значением по умолчанию может быть не только литеральная константа, но и функция без параметров. В частности, мы можем

использовать функцию **CURRENT_TIMESTAMP**, возвращающую текущее значение даты-времени. Давайте добавим столбец в таблицу Product, который будет содержать время, соответствующее выполнению операции добавления модели в БД.

```
1.ALTER TABLE Product
2.ADD add_date DATETIME DEFAULT CURRENT_TIMESTAMP;
```

Добавим модель 1125 производителя A

```
1.INSERT INTO Product(maker, model) VALUES('A', '1125');
```

и посмотрим на результат

```
1.SELECT * FROM Product WHERE model ='1125';
```

maker	model	type	add_date
A	1125	PC	2015-08-24 22:21:23.310

Замечания.

1. Если значение по умолчанию не указано, то подразумевается default NULL, т.е. NULL-значение. Естественно, это значение по умолчанию может быть использовано только в том случае, если на столбце нет ограничения NOT NULL.

2. Если добавить столбец в существующую таблицу, то он, согласно стандарту, будет заполнен значениями по умолчанию для имеющихся строк. В SQL Server поведение при добавлении столбца несколько отличается от стандартного. Если выполнить запрос

```
1.ALTER TABLE Product ADD available VARCHAR(20) DEFAULT
  'Yes';
```

который добавляет в таблицу Product столбец available со значением по умолчанию ‘yes’, то, как это ни странно, столбец будет заполнен NULL-значениями. Чтобы «заставить» сервер заполнить столбец значениями ‘yes’, можно использовать один из двух способов:

a). Запретить NULL, т.е. написать такой запрос:

```
1.ALTER TABLE Product ADD available VARCHAR(20) NOT NULL
  DEFAULT 'Yes';
```

Ясно, что этот способ не годится, если столбец допускает значения NULL.

b). Использовать специальное предложение **WITH VALUES**:

```
1.ALTER TABLE Product ADD available VARCHAR(20) DEFAULT
  'Yes' WITH VALUES;
```

Ссылочная целостность: внешний ключ (FOREIGN KEY)

Внешний ключ – это ограничение, которое поддерживает согласованное состояние данных между двумя таблицами, обеспечивая так называемую ссылочную целостность. Этот тип целостности означает, что всегда есть возможность получить полную информацию об объекте, распределенную по нескольким таблицам.

Причины такого распределения, связанные с принципами проектирования реляционной модели, мы рассмотрим в дальнейшем.

Связь между таблицами не является равноправной. В ней всегда есть главная таблица и таблица подчиненная. Связи бывают двух типов: «один к одному» и «один ко многим». Связь «один к одному» означает, что строке главной таблицы соответствует не более одной строки (т.е. одна или ни одной) в подчиненной таблице. Связь «один ко многим» означает, что одной строке главной таблицы отвечает любое число строк (в том числе и 0) в подчиненной таблице.

Связь устанавливается посредством равенства значений определенных столбцов в главной и подчиненной таблицах. При этом столбец (или набор столбцов в случае составного ключа) в подчиненной таблице, который соотносится со столбцом (или набором столбцов) в главной таблице, и называется внешним ключом.

Поскольку главная таблица всегда находится со стороны «один», то столбец, участвующий в связи по внешнему ключу, должен иметь ограничение **PRIMARY KEY** или **UNIQUE**. Внешний же ключ задается при создании или изменении структуры подчиненной таблицы при помощи спецификации **FOREIGN KEY**:

```
1. FOREIGN KEY(<список столбцов 1> REFERENCES <имя главной
   таблицы>(<список столбцов 2>)
```

Количество столбцов в списках 1 и 2 должно быть одинаковым, а типы данных этих столбцов должны быть попарно совместимы.

Вот как можно создать внешний ключ в таблице PC:

```
1. ALTER TABLE PC
```

```
2.ADD CONSTRAINT fk_pc_product
3.FOREIGN KEY(model) REFERENCES Product(model);
```

Замечание. Для главной таблицы можно не указывать столбец в скобках, если он является первичным ключом, т.к. он может быть только один. В нашем случае так и есть, поэтому последнюю строку можно было написать в виде

```
1.FOREIGN KEY(model) REFERENCES Product;
```

Аналогичным образом создаются внешние ключи в таблицах Printer и Laptop.

Теперь пора разобраться с тем, как работает ограничение внешнего ключа.

Поскольку это ограничение поддерживает согласованность данных в двух таблицах, оно препятствует возникновению таких строк в подчиненной таблице, для которых нет соответствующих строк в главной таблице. Рассогласование могло бы возникнуть в результате выполнения следующих действий.

1. Добавление в подчиненную таблицу строки, для которой нет соответствия в главной таблице. В нашем случае внешние ключи не позволят добавить ни в одну из продукционных таблиц (PC, Laptop или Printer) изделия, модели которого нет в таблице Product. Например, мы получим ошибку при попытке выполнить такой оператор:

```
1.INSERT INTO pc VALUES(13, 1126, 500, 64, 10, '24x', 650);
```

т.к. модели 1126 нет в таблице Product.

The INSERT statement conflicted with the FOREIGN KEY constraint "fk_pc_product". The conflict occurred in database "learn", table "dbo.product", column 'model'. The statement has been terminated.

(Конфликт инструкции INSERT с ограничением FOREIGN KEY " fk_pc_product ". Конфликт произошел в базе данных "learn", таблица

"dbo.product", столбец 'model'. Выполнение данной инструкции было прервано.)

2. Изменение существующего значения внешнего ключа на значение, которого нет в соответствующем столбце главной таблицы. В нашем примере ограничение не позволит выполнить такой оператор UPDATE

```
1.UPDATE pc SET model = 1126 WHERE model = 1121;
```

и вернёт аналогичную ошибку.

3. Удаление строки из главной таблицы, для которой есть связанные строки в подчиненной таблице. Согласованность данных здесь может поддерживаться разными способами, в соответствии со значением опции в необязательном предложении

```
1.ON DELETE <опция>
```

Возможны следующие значения опции:

- **CASCADE** каскадное удаление, т.е. при удалении строки из главной таблицы будут удалены также связанные строки из подчиненной таблицы. Например, при удалении модели 1121 из таблицы Product будут удалены строки с кодами 2, 4 и 5 из таблицы PC;
- **SET NULL** - при удалении строки из главной таблицы значение внешнего ключа становится неопределенным для тех строк из подчиненной таблицы, которые связаны с удаляемой строкой. Естественно, этот вариант подразумевает, что на внешнем ключе нет ограничения NOT NULL. В нашем примере с удалением модели 1121 из таблицы Product значение столбца model в таблице PC примет значение NULL для строк с кодами 2, 4 и 5;
- **SET DEFAULT** – действие аналогичное предыдущему варианту, только вместо NULL будет использовано значение по умолчанию;
- **NO ACTION** (принимается по умолчанию) – операция выполнена не будет, если для удаляемой строки существуют связанные строки в подчиненной таблице. Если связанных строк нет, то удаление будет выполнено.

Поскольку при создании внешнего ключа для таблицы PC мы не указали никакой опции, то будет использоваться NO ACTION – опция, принимаемая по умолчанию. Чтобы изменить поведение, скажем, на каскадное удаление, мы должны переписать ограничение внешнего ключа. Сделать это можно следующим образом:
- удалить существующее ограничение;
- создать новое ограничение.

Для удаления ограничения также используется оператор **ALTER TABLE**:

```
1.ALTER TABLE <имя таблицы>
2.DROP CONSTRAINT <имя ограничения>;
```

Вот где нам понадобилось имя ограничения! Давайте удалим внешний ключ из таблицы PC.

```
1.ALTER TABLE PC
2.DROP CONSTRAINT fk_pc_product;
```

Примечание:

При удалении внешнего ключа сами столбцы не удаляются, удаляется лишь ограничение. Это также справедливо и для других ограничений.

Создадим теперь новое ограничение, использующее каскадное удаление:

```
1.ALTER TABLE PC
2.ADD CONSTRAINT fk_pc_product
3.FOREIGN KEY(model) REFERENCES Product ON DELETE CASCADE;
```

4. Изменение значений столбцов в главной таблице, с которыми связан внешний ключ в подчиненной таблице, т.е. тех столбцов, которые указаны в предложении **REFERENCES** ограничения **FOREIGN KEY**. Здесь действуют те же варианты, что и в случае с удалением строки из главной таблицы, только опция вводится предложением

```
1.ON UPDATE <опция>
```

При помощи внешнего ключа, как и других ограничений, мы моделируем связи, которые существуют в предметной области. Поэтому выбор опций определяется именно предметной областью. В нашем случае при изменении номера модели в таблице Product естественно создать ограничение с опцией **CASCADE**, чтобы это изменение проникало в продукционные таблицы, удаляя изделия аннулированной модели, т.е. для таблицы PC нам следует написать:

```
1.ALTER TABLE PC
2.ADD CONSTRAINT fk_pc_product
3.FOREIGN KEY(model) REFERENCES Product
4.            ON DELETE CASCADE
5.            ON UPDATE CASCADE;
```

Однако для другой предметной области каскадное удаление может привести к ошибочной потере данных. Пусть, например, для таблиц Сотрудники и Отделы существует связь по номеру отдела. Если при удалении (расформировании) отдела сотрудники не увольняются, а переводятся в другие отделы, то каскадное удаление ошибочно привело бы к удалению информации о сотрудниках, работавших в этом отделе. Здесь подошел бы вариант NO ACTION – чтобы сначала распределить сотрудников по другим отделам, а потом удалить «пустой» отдел; или вариант SET NULL, т.е. сначала удаляем отдел, а потом занимаемся трудоустройством сотрудников, не приписанных ни к какому отделу. Еще раз повторю, что выбор варианта зависит не от предпочтений программиста, а от процессов, имеющих место в реальном мире.

Замечания

1. Между таблицами Product и PC выше мы реализовали связь «один ко многим». Связь «один к одному» создается в случае, когда в подчиненной таблице внешним ключом является уникальный столбец или уникальная комбинация столбцов. В ряде случаев связь «один к одному» является ошибкой проектирования, поскольку фактически одна сущность разбивается на две. Однако для такого разделения иногда имеются веские основания, например, когда с целью повышения производительности или обеспечения безопасности приходится выполнить вертикальное секционирование (partitioning) таблицы.

2. При удалении ограничения необходимо знать его имя. Однако, как мы уже знаем, можно создать ограничение, не давая ему имени. Как быть в этом случае? Если мы явно не указываем имя ограничения, его генерирует система. Поэтому имя всегда есть. Другой вопрос, что мы его не знаем. Тут уместно сказать, что в реляционных системах метаданные хранятся так же, как и данные, т.е. в таблицах. Стандартным представлением метаданных является информационная схема, к которой можно адресовать обычные запросы на выборку. Не углубляясь в детали, напишем запрос, который вернет нам имя ограничения внешнего ключа для таблицы PC:

```
1. SELECT CONSTRAINT_NAME
2. FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS
3. WHERE TABLE_NAME ='PC' AND CONSTRAINT_TYPE ='FOREIGN
   KEY';
```

Вложенные запросы в проверочных ограничениях

Мы уже многое сделали, чтобы наша реляционная модель соответствовала предметной области. Однако некоторые проблемы, нарушающие согласованность данных, остались. Например, мы можем добавить в таблицу PC модель (1288), которая в таблице Product объявлена как принтер:

```
1. INSERT INTO PC VALUES(13, 1288, 500, 64, 10, '24x', 650);
```

Более того, ничто не мешает нам добавить эту модель во все продукционные таблицы – PC, Laptop, Printer.

Итак, нам требуется ограничение, которое бы запретило иметь в подчиненных таблицах продукты несоответствующего типа.

Сформулируем проверочное ограничение, которое будет определять тип модели по таблице Product и сравнивать его с типом продукционной таблицы. Например, для таблицы PC такое ограничение могло бы иметь вид:

```
1.ALTER TABLE PC
2.ADD CONSTRAINT model_type
3.CHECK('PC' = (SELECT type FROM Product pr WHERE pr.model
  = pc.model));
```

При попытке выполнить вышеприведенный код (вполне легитимный с точки зрения стандарта SQL-92) получаем сообщение об ошибке:

Subqueries are not allowed in this context. Only scalar expressions are allowed.

(Вложенные запросы в данном контексте запрещены. Допускаются только скалярные выражения.)

Другими словами, SQL Server не допускает использования подзапросов в ограничении CHECK. Что касается реализаций, то это, кстати, больше правило, чем исключение. Что касается MySQL, то эта СУБД вообще не поддерживает ограничений CHECK.

Восполнить этот пробел в SQL Server позволяет использование функций пользователя (**UDF**). Трюк состоит в следующем.

Поскольку, как это следует из сообщения об ошибке, в ограничении **CHECK** допускаются лишь скалярные выражения, напишем скалярнозначную функцию, которая будет принимать номер модели и возвращать ее тип, указанный в таблице Product. Затем эту функцию мы будем использовать в ограничении. Итак,

```
1.CREATE FUNCTION get_type(@model VARCHAR(50))
2.RETURNS VARCHAR(50)
3.AS
4.BEGIN
5.RETURN (SELECT type FROM Product WHERE model=@model)
6.END;
```

Теперь добавим ограничение:

```
1. ALTER TABLE PC
2. ADD CONSTRAINT model_type CHECK('PC' = dbo.get_type(model));
```

Теперь при попытке вставить в таблицу PC модель принтера, например:

```
1. INSERT INTO PC VALUES(13, 1288, 500, 64, 10, '24x', 650);
```

мы получаем следующее сообщение об ошибке:

The INSERT statement conflicted with the CHECK constraint "model_type". The conflict occurred in database "learn", table "dbo.pc", column 'model'.

(Конфликт инструкции INSERT с ограничением CHECK "model_type". Конфликт произошел в базе данных "learn", таблица "dbo.pc", столбец 'model'. Выполнение данной инструкции было прервано.)

Модель же соответствующего типа можно добавить в таблицу:

```
1. INSERT INTO PC VALUES(13, 1260, 500, 64, 10, '24x', 650);
```

Надеюсь, что вам не составит труда написать подобные ограничения и для остальных таблиц этой схемы.

Проверочное ограничение уровня таблицы

В описании схемы «Окраска» утверждается, что объем краски одного цвета на квадрате не может превышать 255 единиц. Как реализовать это ограничение? Рассмотренные ранее варианты нам

не подойдут, т.к. каждая строка таблицы utB может отвечать всем ограничениям на отдельную окраску, но суммарный объем при этом может превысить допустимый предел. Ограничение подобного типа называется ограничением уровня таблицы, т.е. при проверке оно адресуется не к отдельной строке, которой коснулось изменение, а ко всей таблице.

Поскольку тут нам опять потребуется запрос в ограничении CHECK, что не реализовано, напишем сначала пользовательскую функцию, которая будет возвращать 1, если объем какой-либо краски на каком-либо квадрате превысил 255 единиц, и ноль – в противном случае. Лежащий в основе UDF запрос достаточно прост – группировка по ИД квадрата и цвету с фильтрацией в предложении HAVING по условию, что сумма краски превысила 255. Если такой запрос будет содержать строки, то функция вернет 1, иначе – 0. Собственно функция:

```
1.CREATE FUNCTION check_volume()
2.RETURNS INT
3.AS
4.BEGIN
5.DECLARE @ret int
6.IF EXISTS(SELECT SUM(B_VOL)
7.FROM utB JOIN utV ON b_v_id=v_id
8.GROUP BY b_q_id, V_COLOR
9.HAVING SUM(B_VOL) > 255)
10.     SELECT @ret =1 ELSE SELECT @ret = 0;
11.     RETURN @ret;
12.     END;
```

Осталось написать совсем простое ограничение – возвращаемое функцией значение равно 0 или не равно 1, - это кому как нравится:

```
1.ALTER TABLE utB
```

```
2. ADD CONSTRAINT square_volume CHECK(dbo.check_volume() =
   0);
```

Попробуем теперь добавить какой-нибудь краски к белому квадрату (т.е. квадрату, который уже окрашен по максимуму всеми цветами), например, квадрату с b_q_id=1:

```
1. INSERT INTO utB VALUES(CURRENT_TIMESTAMP, 1, 4, 10);
```

В результате мы получим ошибку:

The INSERT statement conflicted with the CHECK constraint "square_volume". The conflict occurred in database "learn", table "dbo.utb". The statement has been terminated.

(Конфликт инструкции INSERT с ограничением CHECK "square_volume". Конфликт произошел в базе данных "learn", таблица "dbo.utb". Выполнение данной инструкции было прервано.)

В качестве упражнения напишите ограничение, которое запретит использование пустых баллончиков, т.е. когда объем краски, израсходованной из баллончика, оказывается более 255.

INFORMATION_SCHEMA и Oracle

Информационная схема (**INFORMATION_SCHEMA**) является стандартным представлением метаданных в языке SQL. Исторически каждый производитель реляционных СУБД предоставлял системные таблицы, которые содержали мета-информацию - имена таблиц, столбцов, ограничений, типы данных

столбцов и т.д. Структура и состав системных таблиц могут меняться в разных версиях продукта, однако поддержка информационной схемы дает возможность менять структуру системных таблиц без изменения способа доступа к метаданным. Другим преимуществом применения INFORMATION_SCHEMA является то, что запросы к метаданным не зависят от используемой СУБД.

Из ведущих производителей, пожалуй, только **Oracle** не поддерживает INFORMATION_SCHEMA. Справедливости ради следует сказать, что Oracle предоставляет возможность использовать системные представления вместо непосредственного обращения к системным таблицам, что также позволяет безопасно изменять структуру системных таблиц.

Приведем ряд типичных запросов в стандартном варианте и для Oracle.

1. Список таблиц базы данных (текущей)

Стандартное

```
1.SELECT table_name FROM information_schema.TABLES;
```

Oracle

```
1.SELECT table_name FROM all_tables WHERE owner='ИМЯ СХЕМЫ';
```

2. Список столбцов заданной таблицы (например, PC)

Стандартное

```
1. SELECT column_name FROM information_schema.COLUMNS WHERE
   table_name='PC';
```

Oracle

```
1. SELECT    column_name    FROM    all_tab_columns    WHERE
   table_name='PC';
```

3. Тип данных и размер заданного символьного столбца таблицы (столбец model таблицы PC)

Стандартное

```
1. SELECT column_name, data_type, CHARACTER_MAXIMUM_LENGTH
2. FROM information_schema.COLUMNS WHERE table_name='PC' AND
   column_name = 'model';
```

Oracle

```
1. SELECT      column_name,data_type,data_length      FROM
   all_tab_columns
2. WHERE table_name='PC'AND column_name='MODEL';
```

4. Имя ограничения первичного ключа таблицы PC

Стандартное

```
1. SELECT         table_name,constraint_name          FROM
   INFORMATION_SCHEMA.TABLE_CONSTRAINTS
2. WHERE table_name='PC' AND CONSTRAINT_TYPE='Primary key';
```

Oracle

```
1. SELECT table_name,constraint_name FROM all_constraints
2. WHERE table_name='PC' AND constraint_type='P';
```

5. Список столбцов, составляющих первичный ключ таблицы Utb

Стандартное

```
1. SELECT table_name, column_name
2. FROM   INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE   WHERE
   table_name='utb'
3. AND    constraint_name=(SELECT    constraint_name    FROM
   INFORMATION_SCHEMA.table_constraints
4. WHERE   table_name='utb'   AND   CONSTRAINT_TYPE='Primary
   key');
```

Здесь мы в основном запросе получаем столбцы, участвующие в ограничениях, и выбираем те из них, у которых имя ограничения совпадает с именем ограничения первичного ключа для той же таблицы. В принципе, условие *table_name='utb'* основного запроса является избыточным с точки зрения получения правильного ответа, поскольку имя ограничения является уникальным для всей базы. Однако ранняя фильтрация существенно ускоряет выполнение запроса.

Оракл

```
1. SELECT table_name, column_name FROM all_ind_columns WHERE
   table_name='UTB'
2. AND     index_name=(SELECT      constraint_name      FROM
   all_constraints
3. WHERE table_name='UTB' AND constraint_type='P');
```

Вопросы оптимизации

Настройка производительности СУБД - многоуровневый процесс, и оптимизация запросов является лишь одним из его аспектов.

Под оптимизацией запроса понимаются действия, приводящие к тому, что оптимизатор запросов выбирает наилучший процедурный план его выполнения. Зачастую это сводится к созданию или перестройке соответствующих индексов, обновлению статистики и т.д., т.е. действиям, не приводящим к переписыванию самого запроса.

Чтобы выполнять такого рода оптимизацию, необходимо умение читать план выполнения запроса и понимать, как выполняются физические операторы, фигурирующие в плане. Этим вопросам посвящены статьи и переводы, которые публикуются в блогах на sql-ex.ru

Здесь же мы будем рассматривать несколько иные вопросы, которые более соответствуют тематике настоящего учебника. Эти вопросы связаны с тем, что одну и ту же практическую задачу можно решить по разному, т.е. использовать различные алгоритмы, реализованные затем в SQL. Грубо говоря, можно так написать запрос, что никакая оптимизация не повысит его производительности до требуемого уровня. Многочисленные примеры сказанного вы можете найти на форумах sql-ex.ru, где публикуются разнообразные решения одной и той же задачи. А имеющийся на сайте оптимизационный этап непосредственно связан с написанием производительных запросов.

Основные операции плана выполнения SQL Server

Данная статья представляет собой описание основных операций, отображаемых в планах выполнения запросов СУБД MS SQL Server.

Index Seek

Поиск по некластеризованному индексу. В большинстве случаев является хорошим для производительности, так как представляет собой прямой доступ SQL Server к требуемым строкам данных. Однако это вовсе не означает, что он всегда работает быстро, например, если он возвращает большое число строк, то по производительности он будет практически равен Index Scan.

Подробнее...

Index Scan

Сканирование некластеризованного индекса. Обычно наличие этой операции плохо отражается на производительности, поскольку она предполагает последовательное чтение индекса для извлечения большого числа строк, приводя к более медленной обработке. Но бывают исключения, например, применение директивы TOP, ограничивающей число возвращаемых записей; если возвращать всего несколько строк, то операция сканирования будет выполняться достаточно быстро, и вы не сможете получить лучшую производительность, чем ту, которую уже имеете, даже если вы попытаетесь перестроить запрос/индексы, чтобы добиться операции Index Seek.

Подробнее...

RID Lookup

Поиск идентификатора записи, является узким местом производительности запроса. Но это легко исправить: если вы видите этот оператор, это означает, что у вас отсутствует кластеризованный индекс на таблице. По крайней мере, вы должны добавить кластеризованный индекс, и тут же получите некоторый рост производительности для большинства ваших запросов.

Подробнее...

Key Lookup

Поиск ключей. Возникает, когда SQL Server предполагает, что он с большей эффективностью может использовать некластеризованный индекс, а затем перейти к кластерзованному индексу для поиска оставшихся значения строк, которые отсутствуют в некластеризованном индексе. Это не всегда плохо: обращение SQL Server к кластеризованному индексу для извлечения недостающих значений довольно эффективный метод по сравнению с необходимостью создавать и поддерживать совершенно новые индексы.

Однако, если все, что нужно SQL Server от операции Key Lookup, это единственный столбец данных, гораздо проще добавить этот столбец в ваш существующий некластеризованный индекс. Размер индекса увеличится на один столбец, но SQL Server сможет избежать необходимости обращаться к двум индексам для извлечения всех необходимых данных и это в целом окажется более эффективным решением.

Подробнее...

Sort

Сортировка является одной из наиболее дорогих операций, которые могут быть в плане выполнения, поэтому лучше избегать ее, насколько это возможно.

Простой способ избежать оператора сортировки – иметь данные, хранящиеся в предварительно упорядоченном виде. Это может быть выполнено созданием индекса с ключевыми столбцами, перечисленными в том же самом порядке, который использует оператор сортировки.

Если SQL Server должен выполнить сортировку одних и тех же данных в одном и том же порядке несколько раз в плане выполнения, то еще одним выходом является разбиение запроса на несколько этапов при использовании временных индексированных таблиц для сохранения данных между этапами. В таком случае,

если вы будете повторно использовать временную таблицу в плане выполнения вашего запроса, то вы получите чистую экономию.

Подробнее...

Spool

Спулы бывают разных типов, но большинство из них можно сформулировать как операторы, которые сохраняют промежуточную таблицу результатов в tempdb.

SQL Server часто использует спул для обработки сложных запросов, преобразуя данные во временную таблицу в базе tempdb для использования её данных в последующих операциях. Побочным эффектом здесь является необходимость записи данных на диск.

Для ускорения выполнения запроса можно попытаться найти способ его перезаписи таким образом, чтобы избежать спула. Если это не получается, использую метод "разделяй и властвуй" для временных таблиц, который может также заменить спул, обеспечивая больший контроль по сравнению с тем, как SQL Server записывает и индексирует данные в tempdb.

Подробнее...

Merge Join

Соединение слиянием. Редко встречаются в реальных запросах, как правило, являются наиболее эффективными из операторов логического соединения.

Оптимизатор выбирает использование соединение слиянием, когда входные данные уже отсортированы или SQL Server может выполнить сортировку данных с относительно небольшой стоимостью.

Операция неприменима, если входные данные не отсортированы.

Подробнее...

Nested Loops Join

Соединение вложенными циклами. Встречаются очень часто. Выполняют довольно эффективное соединение относительно небольших наборов данных.

Соединение вложенными циклами не требует сортировки входных данных. Однако производительность можно улучшить при помощи сортировки источника входных данных; SQL Server сможет выбрать более эффективный оператор, если оба входа отсортированы.

Операция неприменима, если данные слишком велики для хранения в памяти.

Подробнее...

Hash Match Join

Операция используется всегда, когда невозможно применить другие виды соединения. Она выбираются оптимизатором запросов по одной из двух причин:

1. Соединяемые наборы данных настолько велики, что они могут быть обработаны только с помощью Hash Match Join.
2. Наборы данных не упорядочены по столбцам соединения, и SQL Server думает, что вычисление хэшей и цикл по ним будет быстрей, чем сортировка данных.

При первом сценарии трудно оптимизировать выполнение запроса, если только не найти способа соединять меньшие объемы данных.

При втором же сценарии, если есть некоторый способ получить данные в упорядоченном виде до соединения, типа предопределенного порядка сортировки в индексе, то возможно, что SQL Server выберет вместо этой операции более быстрый алгоритм соединения.

Операторы Hash Match Join достаточно эффективны тогда, когда не сбрасывают данные в tempdb.

Подробнее...

Parallelism

Операторы параллелизма обычно считаются хорошими вещами: SQL Server дробит ваши данные на множество частей для асинхронной обработки на множестве процессоров, сокращая общее время работы, требуемое для выполнения вашего запроса.

Однако параллелизм может стать плохим, если большинство запросов используют его. При параллелизме процессоры по-прежнему выполняют тот же самый объем работы, что и без него, тем самым отнимая ресурсы у других запросов, которые могут быть запущены, плюс накладывается дополнительная нагрузка на SQL Server по дроблению и последующему объединению всех данных из множества нитей выполнения.

Если параллелизм является узким местом производительности, можно рассмотреть вопрос об изменении порогового значения стоимости для настройки параллелизма, если оно установлено слишком низким.

Подробнее...

Stream Aggregate

Статистическое выражение потока. Группирует строки в один или несколько столбцов и вычисляет одно или несколько агрегатных выражений (пример: COUNT, MIN, MAX, SUM и AVG), возвращенных запросом. Выход этого оператора может быть использован последующими операторами запроса, возвращен клиенту или то и другое. Оператору Stream Aggregate необходимы входные данные, упорядоченные по группируемым столбцам. Оптимизатор использует перед этим оператором оператор Sort, если данные не были ранее отсортированы оператором Sort или используется упорядоченный поиск или просмотр в индексе.

Compute Scalar

Оператор Compute Scalar вычисляет выражение и выдает вычисляемую скалярную величину. Затем эту величину можно вернуть пользователю или сослаться на нее в каком-либо запросе, а также выполнить эти действия одновременно. Примерами одновременного использования этих возможностей являются предикаты фильтра или соединения. Всегда возвращает одну строку. Часто применяется для того, чтобы конвертировать результат Stream Aggregate в ожидаемый на выходе тип int (когда Stream Aggregate возвращает bigint в случае с COUNT, AVG при типах столбцов int).

Concatenation

Оператор просматривает несколько входов, возвращая каждую просмотренную строку. Используется в запросах с UNION ALL. Копирует строки из первого входного потока в выходной поток и повторяет эту операцию для каждого дополнительного входного потока.

Filter

Оператор просматривает входные данные и возвращает только те строки, которые удовлетворяют критерию фильтрации (предикату).

Описание операций плана выполнения в Postgresql

Данная статья представляет собой описание основных операций, отображаемых в планах выполнения запросов СУБД PostgreSQL.

Seq Scan

Операция сканирует всю таблицу в порядке, в котором она хранится на диске. Обычно наличие этой операции плохо отражается на производительности, поскольку она предполагает последовательное чтение для извлечения большого числа строк, приводя к более медленной обработке. Но бывают исключения, например, применение директивы Limit, ограничивающей число возвращаемых записей.

Index Scan

Сканирование индекса выполняет обход индекса, просматривает конечные узлы, чтобы найти все соответствующие записи, и извлекает соответствующие данные из таблицы. В большинстве случаев является хорошей для производительности, так как представляет собой прямой доступ к требуемым строкам данных.

Bitmap Index Scan

В то время как Index Scan извлекает один указатель кортежа за раз из индекса и немедленно переходит к этому кортежу в таблице, Bitmap Index Scan извлекает все указатели кортежей из индекса за один раз, сортирует их, используя структуру данных "битовая карта (bitmap)" в оперативной памяти, а затем просматривает кортежи таблиц в порядке физического расположения кортежей.

Merge Join

Соединение слиянием объединяет два отсортированных списка. Обе стороны объединения должны быть предварительно отсортированы.

Nested Loops

Соединение вложенными циклами объединяет две таблицы, выбирая результат из одной таблицы и запрашивая другую таблицу для каждой строки из первой. Встречается очень часто. Выполняет довольно эффективное соединение относительно небольших

наборов данных. Соединение вложенными циклами не требует сортировки входных данных.

Hash Join

Хеш-соединение загружает записи-кандидаты с одной стороны соединения в хеш-таблицу, которая затем проверяется для каждой строки с другой стороны соединения. Операция используется всегда, когда невозможно применить другие виды соединения: если соединяемые наборы данных достаточно велики и/или наборы данных не упорядочены по столбцам соединения.

Sort

Сортирует набор по столбцам, указанным в Sort Key. Операция сортировки требует больших объемов памяти для материализации промежуточного результата.

Aggregate

Появляется в плане, если в запросе есть агрегатная функция, используемая для вычисления отдельных результатов из нескольких входных строк: COUNT, SUM, AVG, MAX, MIN и прочие.

GroupAggregate

Группирует предварительно отсортированный набор в соответствии с предложением GROUP BY. Эта операция не буферизует промежуточный результат.

HashAggregate

Использует временную хэш-таблицу для группировки записей. Операция HashAggregate не требует предварительно отсортированного набора данных, вместо этого она использует большие объемы памяти для материализации промежуточного результата. Вывод не упорядочен каким-либо значимым образом.

Unique

Удаляет дублирующиеся данные. Практически не требует памяти: сравнивает значение в предыдущей строке с текущим и, если они одинаковые, отбрасывает его. Данные должны быть предварительно отсортированны.

Filter

Применяет фильтр к набору строк.

Limit

Прерывает выполение операций, когда было выбрано нужное количество строк.

Append

Запускает множество субопераций и возвращает все возвращенные ими строки в виде общего результата. Используется в запросах, содержащих UNION или UNION ALL.

HashSetOp

Операция используется операциями INTERSECT и EXCEPT (с опциональным модификатором ALL). Она работает следующим образом: запускает субоперации Append для пары подзапросов, а затем, на основании результата и опционального модификатора ALL, решает, какие строки нужно вернуть.

Materialize

Операция получает данные из нижележащей операции и размещает их в памяти (или частично в памяти), чтобы ими можно было быстрее воспользоваться, или добавляет им дополнительные свойства, которые предыдущая операция не предоставляет.

CTE Scan

Схожа с Materialize. Операция запускает часть запроса и сохраняет ее вывод, чтобы он мог быть использован другой частью (или частями) запроса.

SubPlan

Операция означает подзапрос, в котором есть ссылки на основной запрос. Вызывается, чтобы посчитать данные из подзапроса, которые зависят от текущей строки.

InitPlan

Операция означает подзапрос, у которого нет ссылок на основной запрос. Операция появляется в плане, когда есть часть запроса, которую можно (или нужно) вычислить прежде всего, и она не зависит ни от чего в остальной части запроса.

Subquery Scan

Операция означает подзапрос, входящий в UNION.

Описание операций плана выполнения в Oracle

Данная статья представляет собой описание основных операций, отображаемых в планах выполнения запросов СУБД Oracle RDBMS.

Index Unique Scan

Выполняет только обход B-дерева. Эта операция используется, если уникальное ограничение гарантирует, что критерии поиска будут соответствовать не более чем одной записи.

Index Range Scan

Выполняет обход B-дерева и просматривает цепочки конечных узлов, чтобы найти все подходящие записи.

Index Full Scan

Читает индекс целиком (все строки) в порядке, представленном индексом. В зависимости от различной системной статистики СУБД может выполнять эту операцию, если нужны все строки в порядке индекса, например, из-за соответствующего предложения ORDER BY. Вместо этого оптимизатор может также использовать операцию Index Fast Full Scan и выполнить дополнительную операцию сортировки.

Index Fast Full Scan

Читает индекс целиком (все строки) в порядке, хранящемся на диске. Эта операция обычно выполняется вместо полного сканирования таблицы, если в индексе доступны все необходимые столбцы. Подобно операции TABLE ACCESS FULL, INDEX FAST FULL SCAN может извлечь выгоду из многоблочных операций чтения.

Table Access By Index ROWID

Извлекает строку из таблицы, используя ROWID, полученный из предыдущего поиска по индексу.

Table Access Full

Полное сканирование таблицы. Читает всю таблицу (все строки и столбцы), в порядке, хранящемся на диске. Хотя многоблочные операции чтения значительно повышают скорость сканирования полной таблицы, это все еще одна из самых дорогих операций. Помимо высоких затрат времени ввода-вывода, полное сканирование таблицы должно проверять все строки таблицы, что также занимает значительное количество процессорного времени.

Merge Join

Соединение слиянием объединяет два отсортированных списка. Обе стороны объединения должны быть предварительно отсортированы.

Nested Loops

Соединение вложенными циклами объединяет две таблицы, выбирая результат из одной таблицы и запрашивая другую таблицу для каждой строки из первой. Встречается очень часто. Выполняет довольно эффективное соединение относительно небольших наборов данных. Соединение вложенными циклами не требует сортировки входных данных.

Hash Join

Хеш-соединение загружает записи-кандидаты с одной стороны соединения в хеш-таблицу, которая затем проверяется для каждой строки с другой стороны соединения. Операция используется всегда, когда невозможно применить другие виды соединения: если соединяемые наборы данных достаточно велики и/или наборы данных не упорядочены по столбцам соединения.

Sort Unique

Сортирует строки и устраняет дупликаты.

Hash Unique

Более эффективная реализация алгоритма сортировки и устранения дупликатов с использованием хэш-таблицы. Заменяет операцию Sort Unique в определенных обстоятельствах.

Sort Aggregate

Вычисляет суммарные итоги с использованием агрегатных функций SUM, COUNT, MIN, MAX, AVG и пр.

Sort Order By

Сортирует результат в соответствии с предложением ORDER BY. Эта операция требует больших объемов памяти для материализации промежуточного результата.

Sort Group By

Сортирует набор записей по столбцам GROUP BY и агрегирует отсортированный результат на втором этапе. Эта операция требует больших объемов памяти для материализации промежуточного результата.

Sort Group By Nosort

Агрегирует предварительно отсортированный набор записей в соответствии с предложением GROUP BY. Эта операция не буферизует промежуточный результат.

Hash Group By

Группирует результат, используя хеш-таблицу. Эта операция требует больших объемов памяти для материализации промежуточного набора записей. Вывод не упорядочен каким-либо значимым образом.

Filter

Применяет фильтр к набору строк.

View

Создает промежуточное представление данных.

Count Stopkey

Прерывает выполение операций, когда было выбрано нужное количество строк.

Sort Join

Сортирует набор записей в столбце соединения. Используется в сочетании с операцией Merge Join для выполнения сортировки соединением слияния.

Intersection

Выполняет операцию пересечения между двумя источниками.

Union-All

Выполняет операцию объединения всех записей между двумя таблицами. Дублирующиеся строки не удаляются.

Load As Select

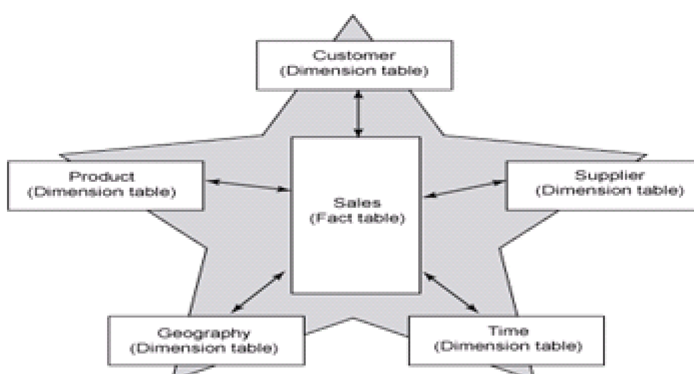
Прямая загрузка с использованием оператора SELECT в качестве источника.

Temp Table Generation/Transformation

Создает/преобразует временную таблицу. Используется в специфичных для Oracle преобразованиях типа Star.

Optimizer Transformations: Star Transformation

Star transformation was introduced in Oracle 8i to process star queries efficiently. These queries are commonly used in data warehouse applications that follow the Star Schema data model. The Star Schema is so called because the data model diagram resembles a star. The center of the star consists of one or more fact tables and the points of the star are the dimension tables.



The basic idea of this transformation is to steer clear of using a full table scan access method on large tables, referred to as **fact** tables in the Star Schema. In a typical star query, the fact table is joined to several much smaller **dimension** tables. The fact table typically contains one key (referred to as foreign key) for every dimension table as well as a number of measure columns such as sales amount. The corresponding key in the dimension table is referred to as the primary key. The join is based on a foreign key of the fact table with the corresponding primary key of the dimension table. The query also contains filter predicates on other columns of the dimension tables that typically are very restrictive. The combination of these filters help to dramatically reduce the data set processed from the fact table. The goal of star transformation is to access only this reduced set of data from the fact table.
Consider the following star query Q1. The query is to find the total sales amount in all cities in California for quarters Q1 and Q2 of year 1999 through the Internet.

Q1:

Copy code snippet

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

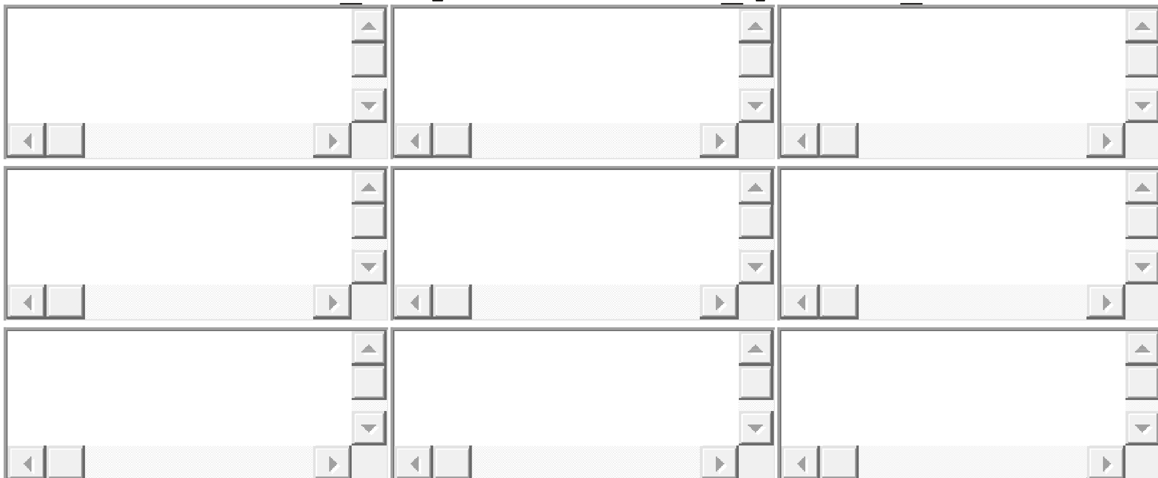
Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

```
SELECT c.cust_city,
       t.calendar_quarter_desc,
       SUM(s.amount_sold) sales_amount
FROM   sales     s,
       times     t,
       customers c,
       channels  ch
WHERE  s.time_id = t.time_id
AND    s.cust_id = c.cust_id
AND    s.channel_id = ch.channel_id
AND    c.cust_state_province = 'CA'
AND    ch.channel_desc = 'Internet'
AND    t.calendar_quarter_desc IN ('1999-01','1999-02')
GROUP BY c.cust_city, t.calendar_quarter_desc;
```



Sales is the fact table while the other tables are considered as dimension tables. The *Sales* table contains one row for every sale of a product and thus it may contain billions of sales records. However only a few of them are sold to customers in California through the Internet for the specified quarters. The query is transformed into Q2.

Q2:

Copy code snippet

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

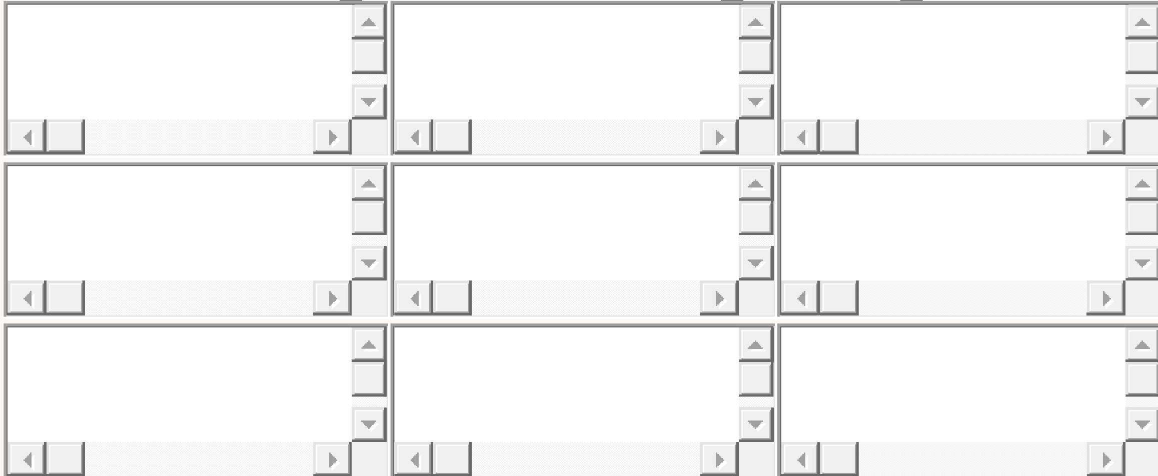
Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

```
SELECT c.cust_city,
       t.calendar_quarter_desc,
       SUM(s.amount_sold) sales_amount
FROM sales     s,
     times     t,
     customers c
WHERE s.time_id = t.time_id
  AND s.cust_id = c.cust_id
  AND c.cust_state_province = 'CA'
  AND t.calendar_quarter_desc IN ('1999-01','1999-02')
  AND s.time_id IN (SELECT time_id
                            FROM times
                            WHERE calendar_quarter_desc
IN('1999-01','1999-02'))
  AND s.cust_id IN (SELECT cust_id
```

```
                            FROM customers
                            WHERE
cust_state_province='CA')
  AND s.channel_id IN (SELECT channel_id
                                  FROM channels
                                  WHERE channel_desc =
'Internet')
GROUP BY c.cust_city, t.calendar_quarter_desc;
```



Star transformation is essentially about adding subquery predicates corresponding to the constraint dimensions. These subquery predicates are referred to as bitmap semi-join predicates. The transformation is performed when there are indexes on the fact join columns (*s.timeid, s.custid*...). By driving bitmap AND and OR operations (bitmaps can be from bitmap indexes or generated from regular B-Tree indexes) of the key values supplied by the subqueries, only the relevant rows from the fact table need to be retrieved. If the filters on the dimension tables filter out a lot of data, this can be much more efficient than a full table scan on the fact table. After the relevant rows have been retrieved from the fact table, they may need to be joined back to the dimension tables, using the original predicates. In some cases, the join back can be eliminated. We will discuss this situation later.

Table 1 shows the query plan for the transformed query. Note that the *sales* table has a bitmap access path instead of a full table scan. For each key value coming from the subqueries (lines 11, 16, 21), the bitmaps are retrieved from the fact table indexes (lines 12, 17, 22). Each bit in the bitmap corresponds to a row in fact table. The bit is set if the key value from the subquery is same as the value in the row of fact table. For example, the bitmap [1][0][1][0][0][0]...(all 0s for remaining rows) indicate that rows 1 and 3 of fact table has matching key value from subquery. Lets say the above bitmap is for a key value from customers table subquery.

The operations in lines 9, 14, 19 iterates over the keys from the subqueries and get the corresponding bitmaps. Lets say the customers subquery produces one more key value with the bitmap [0][1][0][0][0][0]...

The bitmaps for each subquery are merged (ORed) (lines 8, 13 and 18). In the above example, it will produce a single bitmap [1][1][1][0][0][0]... for customers subquery after merging the two bitmaps.

The merged bitmaps are ANDed (line 7). Lets say the bitmap from channels is [1][0][0][0][0][0]... If you AND this bitmap with the bitmap from customers subquery it will produce [1][0][0][0][0]...

The corresponding rowids of the final bitmap are generated (line 6). The fact table rows are retrieved using the rowids (line 5). In the above example, it will generate only 1 rowid corresponding to the first row and fetches only a single row instead of scanning the entire fact table.
The representation of bitmaps in the above example is for illustration purpose only. In Oracle, they are represented and stored in a compressed form.

Table 1: The plan of the transformed query...
Copy code snippet

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

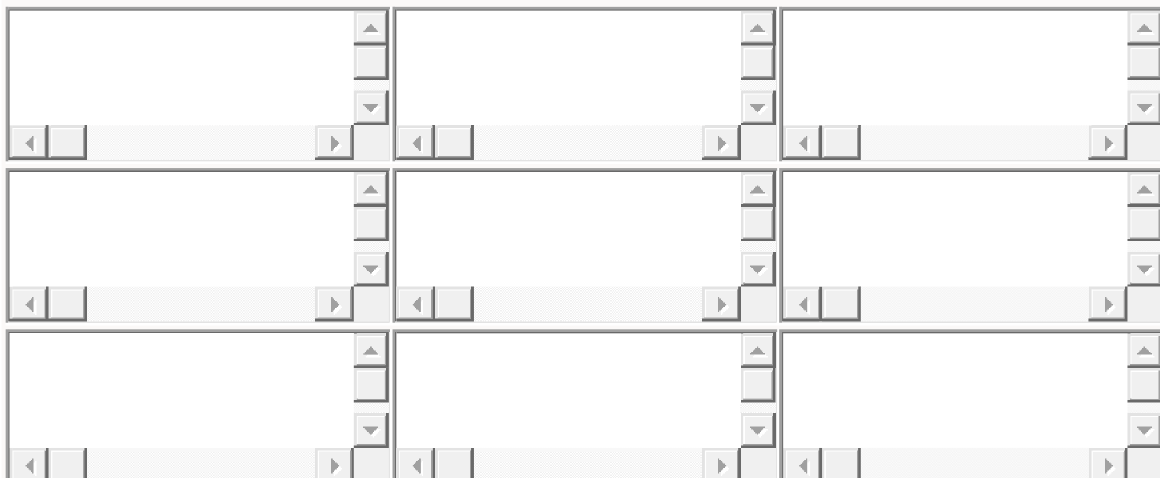
Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

```
-------------------------------------------------------
-------
| Id  | Operation                          | Name
|
-------------------------------------------------------
-------
|   0 | SELECT STATEMENT                   |
|
|   1 |  HASH GROUP BY                     |
|
|*  2 |   HASH JOIN                        |
|
|*  3 |    TABLE ACCESS FULL               | CUSTOMERS
|
|*  4 |    HASH JOIN                       |
|
|*  5 |     TABLE ACCESS FULL              | TIMES
|
|   6 |     VIEW                           |
VW_ST_B1772830   |
|   7 |      NESTED LOOPS                  |
|
|   8 |       PARTITION RANGE SUBQUERY     |
|
|   9 |        BITMAP CONVERSION TO ROWIDS|
|
|  10 |         BITMAP AND                 |
|
|  11 |          BITMAP MERGE              |
|
|  12 |           BITMAP KEY ITERATION     |
|
|  13 |            BUFFER SORT             |
|
```

```
|* 14 |              TABLE ACCESS FULL     | CHANNELS
|
|* 15 |             BITMAP INDEX RANGE SCAN|
SALES_CHANNEL_BIX|
|  16 |           BITMAP MERGE             |
|
|  17 |            BITMAP KEY ITERATION    |
|
|  18 |             BUFFER SORT            |
|
|* 19 |              TABLE ACCESS FULL     | TIMES
|
|* 20 |             BITMAP INDEX RANGE SCAN|
SALES_TIME_BIX   |
|  21 |           BITMAP MERGE             |
|
|  22 |            BITMAP KEY ITERATION    |
|
|  23 |             BUFFER SORT            |
|
|* 24 |              TABLE ACCESS FULL     | CUSTOMERS
|
|* 25 |             BITMAP INDEX RANGE SCAN|
SALES_CUST_BIX   |
|  26 |        TABLE ACCESS BY USER ROWID  | SALES
|
-------------------------------------------------------
-------
```



Join back elimination

The subqueries and their bitmap tree only filter the fact table based on the dimension filters, so it may still be necessary to join to the dimension table. The join back of the dimension table is eliminated when all the predicates on dimension tables are part of the semijoin subquery predicate,

the column(s) selected from the subquery are unique and the dimension columns are not in select list, group by etc. In the above example, the table *channels* is not joined back to the *sales* table since it is not referenced outside and *channel_id* is unique.

Temporary table transformation

If the join back is not eliminated, Oracle stores the results of the subquery in a temporary table to avoid re-scanning the dimension table (for bitmap key generation and join back). In addition to this, the results are materialized if the query is run in parallel, so that each slave can select the results from the temporary tables instead of executing the subquery again.

For example, if Oracle materializes the results of the subquery on *customers* into a temporary table, the transformed query Q3 will be as follows.

Q3:

Copy code snippet

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

```
SELECT t1.c1 cust_city,
       t.calendar_quarter_desc calendar_quarter_desc,
       sum(s.amount_sold) sales_amount
FROM sales s,
     sh.times t,
     sys_temp_0fd9d6621_e7e24 t1
WHERE s.time_id=t.time_id
  AND s.cust_id=t1.c0
  AND (t.calendar_quarter_desc='1999-q1' OR
t.calendar_quarter_desc='1999-q2')
  AND s.cust_id IN    (SELECT  t1.c0
                       FROM sys_temp_0fd9d6621_e7e24
t1)
  AND s.channel_id IN (SELECT  ch.channel_id
                       FROM    channels ch
                       WHERE
ch.channel_desc='internet')
  AND s.time_id IN (SELECT t.time_id
                    FROM  times t
                    WHERE
t.calendar_quarter_desc='1999-q1'
                    OR
t.calendar_quarter_desc='1999-q2')
GROUP BY t1.c1,  t.calendar_quarter_desc;
```

Note that *customers* is replaced by the temporary table *sys_temp_0fd9d6621_e7e24* and references to columns *cust_id* and *cust_city* are replaced by the corresponding columns of

the temporary table. The temporary table will be created with 2 columns - (*c0 number*, *c1 varchar2(30)*). These columns corresponds to *cust_id* and *cust_city* of customers table. The table will be populated using the following query Q4 at the beginning of the execution of the statement Q3.

Q4:

Copy code snippet

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

```
SELECT c.cust_id, c.cust_city
FROM   customers
```

```
WHERE  c.cust_state_province = 'CA'
```

Table 2 shows the plan for the transformed query.

Table 2: Plan with temporary table transformation...

Copy code snippet

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

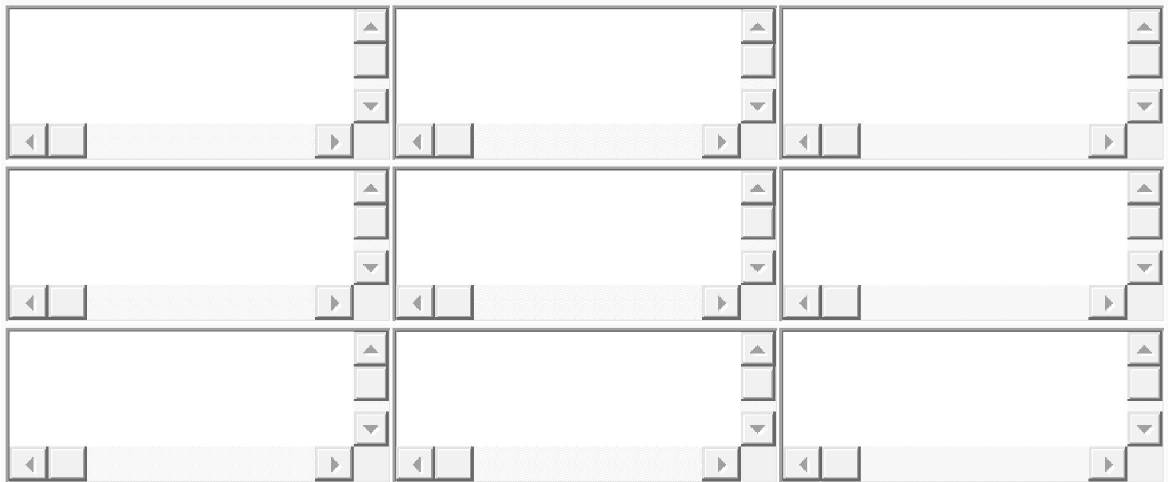
Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

```
-----------------------------------------------------------------------
---------------
| Id  | Operation                           | Name
|
-----------------------------------------------------------------------
---------------
|   0 | SELECT STATEMENT                    |
|
|   1 |  TEMP TABLE TRANSFORMATION          |
|
|   2 |   LOAD AS SELECT                    |
|
|*  3 |    TABLE ACCESS FULL                | CUSTOMERS
|
|   4 |   HASH GROUP BY                     |
|
|*  5 |    HASH JOIN                        |
|
|   6 |     TABLE ACCESS FULL               |
SYS_TEMP_0FD9D6613_C716F|
|*  7 |     HASH JOIN                       |
|
|*  8 |      TABLE ACCESS FULL              | TIMES
|
|   9 |      VIEW                           |
VW_ST_A3F94988          |
|  10 |       NESTED LOOPS                  |
|
|  11 |        PARTITION RANGE SUBQUERY     |
|
|  12 |         BITMAP CONVERSION TO ROWIDS|
|
|  13 |          BITMAP AND                 |
|
|  14 |           BITMAP MERGE              |
|
|  15 |            BITMAP KEY ITERATION     |
|
|  16 |             BUFFER SORT             |
|
|* 17 |              TABLE ACCESS FULL      | CHANNELS
|
```

```
|* 18 |              BITMAP INDEX RANGE SCAN|
SALES_CHANNEL_BIX       |
|  19 |            BITMAP MERGE             |
|
|  20 |             BITMAP KEY ITERATION    |
|
|  21 |              BUFFER SORT            |
|
|* 22 |               TABLE ACCESS FULL     | TIMES
|
|* 23 |              BITMAP INDEX RANGE SCAN|
SALES_TIME_BIX          |
|  24 |            BITMAP MERGE             |
|
|  25 |             BITMAP KEY ITERATION    |
|
|  26 |              BUFFER SORT            |
|
|  27 |               TABLE ACCESS FULL     |
SYS_TEMP_0FD9D6613_C716F|
|* 28 |              BITMAP INDEX RANGE SCAN|
SALES_CUST_BIX          |
|  29 |         TABLE ACCESS BY USER ROWID  | SALES
|
-------------------------------------------------------
---------------
```



The lines 1,2 and 3 of the plan materialize the *customers* subquery into the temporary table. In line 24, it scans the temporary table (instead of the subquery) to build the bitmap from the fact table. Line 26 is for scanning the temporary table for joining back instead of scanning *customers*. Note that the filter on *customers* is not needed to be applied on the temporary table since the filter is already applied while materializing the temporary

table.

Enabling the transformation

Star transformation is controlled by the *star_transformation_enabled* database initialization parameter. The parameter takes 3 values:

TRUE	The Oracle Optimizer performs transformation by identifying fact and constraint dimension tables automatically. This is done in a cost-based manner, i.e. the transformation is performed only if the cost of the transformed plan is lower than the non-transformed plan. Also the optimizer will attempt temporary table transformation automatically whenever materialization improves performance.
FALSE	The transformation is not tried.
TEMP_DISABLE	This value has similar behavior as TRUE except that temporary table transformation is not tried.

The default value of the parameter is FALSE. You have to change the parameter value and create indexes on the joining columns of the fact table to take advantage of this transformation.

Summary

Star transformation improves the performance of queries with a very big fact table joined to multiple dimension tables when the dimension tables have very selective predicates. The transformation avoids the full scan of the fact table. It fetches only relevant rows from the fact table that will eventually join to the constraint dimension rows. The transformation is performed based on cost - only when the cost of the transformed plan is lower than that of the non-transformed plan. If the dimension filters do not significantly reduce the amount of data to be retrieved from the fact table, then a full table scan is more efficient.

In this post we have tried to illustrate the basic ideas behind star transformation by showing simple example queries and plans. Oracle can do star transformation in more complex cases. For example, a query with multiple fact tables, snowflakes (dimension is a join of several normalized tables instead of denormalized single table), etc.

MySQL. Использование переменных в запросе

Довольно часто спрашивают, есть ли аналоги аналитических (оконных) функций в MySQL. Нет. Для их замены часто используют запросы с самосоединением, сложные подзапросы и прочее. Большинство таких решений оказываются неэффективными с точки зрения производительности.

Также в MySQL нет рекурсии. Однако с некоторой частью задач, которые обычно решаются аналитическими функциями или рекурсией, можно справиться и средствами MySQL.

Одним из этих средств является уникальный, нехарактерный для прочих СУБД механизм работы с переменными внутри запроса SQL. Мы можем объявить переменную внутри запроса, менять ей значение и подставлять в SELECT для вывода. Причем порядок обработки строк в запросе и, как следствие, порядок присвоения значений переменным можно задать в пользовательской сортировке!

Предупреждение. В статье подразумевается, что обработка выражений в предложении SELECT осуществляется слева направо, однако официального подтверждения такого порядка обработки в документации MySQL нет. Это необходимо иметь в виду при смене версии сервера.

Для гарантии последовательности вычисления можно использовать фиктивный оператор CASE или IF.

Аналог рекурсии

Рассмотрим простой пример, который генерирует последовательность Фибоначчи (в последовательности Фибоначчи каждый член равен сумме двух предыдущих, а первые 2 равны единице):

```
1.SELECT IF(X=1, Fn_1, Fn_2) F
2.FROM(
3.  SELECT @I := @I + @J Fn_1, @J := @I + @J Fn_2
4.  FROM
5.     (SELECT 0 dummy UNION ALL SELECT 0 UNION ALL SELECT
  0)a,
6.     (SELECT 0 dummy UNION ALL SELECT 0 UNION ALL SELECT
  0)b,
7.     (SELECT @I := 1, @J := 1)IJ
8.)T,
9.  /*Фиктивная таблица, для вывода последовательности в 1
  столбец*/
10.         (SELECT 1 X UNION ALL SELECT 2)X;
```

Данный запрос генерирует 18 чисел Фибоначчи, не считая первых двух:

F
2
3
5

8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765

Разберём теперь как оно работает.

В строчках 5) 6) генерируется 9 записей. Тут ничего необычного.

В строчке 7) мы объявляем две переменные @I, @J и присваиваем им 1.

В строке 3) происходит следующее: сначала переменной @I присваивается сумма двух переменных. Затем то же самое присваиваем переменной @J, причем с учетом того, что значение @I уже поменялось.

Другими словами, вычисления в SELECT выполняются слева направо – см. также замечание в начале статьи.

Причем изменение переменных осуществляется в каждой из наших 9 записей, т.е. при обработке каждой новой строки в переменных @I и @J будут содержаться значения, вычисленные при обработке предыдущей строки.

Чтобы решить эту же задачу средствами других СУБД, нам пришлось бы писать рекурсивный запрос!

Примечание:

Переменные нужно объявлять в отдельном подзапросе (строка 7), если бы мы объявили переменную в предложении SELECT, она, скорее всего, вычислилась бы только 1 раз (хотя конкретное поведение будет зависеть от версии сервера). Тип переменной определяется значением, которым она инициализирована. Этот тип может динамически меняться. Если переменной присвоить NULL, её типом будет BLOB.

Порядок обработки строк в SELECT, как было сказано выше, зависит от пользовательской сортировки. Простой пример нумерации строк в заданном порядке:

```
1.SELECT val, @I:=@I+1 Num
2.FROM
3.  (SELECT 30 val UNION ALL SELECT 20 UNION ALL SELECT 10
  UNION ALL SELECT 50)a,
4.  (SELECT @I := 0)I
5.ORDER BY val;
```

Val	Num
10	1
20	2
30	3
50	4

Аналоги аналитических функций

Переменные также можно использовать для замены аналитических функций. Далее несколько примеров. Для простоты будем считать, что все поля NOT NULL, а сортировка и партиционирование (PARTITION BY) происходят по одному полю. Использование NULL значений и более сложных сортировок сделает примеры более громоздкими, но суть не поменяет.

Для примеров создадим таблицу TestTable:

```
1.CREATE TABLE TestTable(
2.  group_id INT NOT NULL,
3.  order_id INT UNIQUE NOT NULL,
4.  value INT NOT NULL
5.);
```

где

group_id – идентификатор группы (аналог окна аналитической функции);

order_id – уникальное поле, по которому будет производиться сортировка;

value – некоторое числовое значение.

Заполним нашу таблицу тестовыми данными:

```
1.INSERT TestTable(order_id, group_id, value)
2.SELECT *
3.FROM(
4.  SELECT 1 order_id, 1 group_id, 1 value
5.  UNION ALL SELECT 2, 1, 2
6.  UNION ALL SELECT 3, 1, 2
7.  UNION ALL SELECT 4, 2, 1
8.  UNION ALL SELECT 5, 2, 2
```

```
9.   UNION ALL SELECT 6, 2, 3
10.        UNION ALL SELECT 7, 3, 1
11.        UNION ALL SELECT 8, 3, 2
12.        UNION ALL SELECT 9, 4, 1
13.        UNION ALL SELECT 11, 3, 2
14.      )T;
```

Примеры замены некоторых аналитических функций.

1) ROW_NUMBER() OVER(ORDER BY order_id)

```
1.SELECT T.*, @I:=@I+1 RowNum
2.FROM TestTable T,(SELECT @I:=0)I
3.ORDER BY order_id;
```

group_id	order_id	value	RowNum
1	1	1	1
1	2	2	2
1	3	2	3
2	4	1	4
2	5	2	5
2	6	3	6
3	7	1	7
3	8	2	8
4	9	1	9
3	11	2	10

2) ROW_NUMBER() OVER(PARTITION BY group_id ORDER BY order_id)

```
1.SELECT group_id, order_id, value, RowNum
2.FROM(
3.  SELECT T.*,
4.      IF(@last_group_id  =  group_id,  @I:=@I+1,  @I:=1)
  RowNum,
5.     @last_group_id := group_id
6.  FROM TestTable T,(SELECT @last_group_id:=NULL, @I:=0)I
7.  ORDER BY group_id, order_id
8.)T;
```

group_id	order_id	value	RowNum
1	1	1	1
1	2	2	2
1	3	2	3
2	4	1	1
2	5	2	2
2	6	3	3
3	7	1	1
3	8	2	2
3	11	2	3
4	9	1	1

3) SUM(value) OVER(PARTITION BY group_id ORDER BY order_id)

```
1.SELECT group_id, order_id, value, RunningTotal
2.FROM(
3.  SELECT T.*,
```

```
4.        IF(@last_group_id    =    group_id,    @I:=@I+value,
  @I:=value) RunningTotal,
5.      @last_group_id := group_id
6.   FROM TestTable T, (SELECT @last_group_id:=NULL, @I:=0)I
7.   ORDER BY group_id, order_id
8.)T;
```

group_id	order_id	value	RunningTotal
1	1	1	1
1	2	2	3
1	3	2	5
2	4	1	1
2	5	2	3
2	6	3	6
3	7	1	1
3	8	2	3
3	11	2	5
4	9	1	1

4) LAG(value) OVER(PARTITION BY group_id ORDER BY order_id)

```
1.SELECT group_id, order_id, value, LAG
2.FROM(
3.   SELECT T.*,
4.     IF(@last_group_id = group_id, @last_value, NULL) LAG,
5.     @last_group_id := group_id,
6.     @last_value := value
7.      FROM    TestTable    T,(SELECT    @last_value:=NULL,
  @last_group_id:=NULL)I
```

```
8.   ORDER BY group_id, order_id
9.)T;
```

group_id	order_id	value	LAG
1	1	1	NULL
1	2	2	1
1	3	2	2
2	4	1	NULL
2	5	2	1
2	6	3	2
3	7	1	NULL
3	8	2	1
3	11	2	2
4	9	1	NULL

Для LEAD всё то же самое, только нужно сменить сортировку на ORDER BY group_id, order_id DESC

Для функций COUNT, MIN, MAX всё несколько сложнее, поскольку, пока мы не проанализируем все строчки в группе(окне), мы не сможем узнать значение функции. MS SQL, например, для этих целей «спулит» окно (временно помещает строки окна в скрытую буферную таблицу для повторного к ним обращения), в MySQL такой возможности нет. Но мы можем для каждого окна вычислить значение функции в последней строке при заданной сортировке (т.е. после анализа всего окна), а затем, отсортировав строки в окне в обратном порядке, проставить вычисленное значение по всему окну.

Таким образом, нам понадобится две сортировки. Чтобы итоговая сортировка осталась той же, что и в примерах выше, отсортируем сначала по

полям group_id ASC, order_id DESC, затем по полям group_id ASC, order_id ASC.

5) COUNT(*) OVER(PARTITION BY group_id)

В первой сортировке мы просто нумеруем записи. Во второй всем строкам окна присваиваем максимальный номер, который и будет соответствовать количеству строк в окне.

```
1.SELECT group_id, order_id, value, Cnt
2.FROM(
3.  SELECT group_id, order_id, value,
4.    IF(@last_group_id = group_id, @MaxRowNum, @MaxRowNum
  := RowNumDesc) Cnt,
5.    @last_group_id := group_id
6.  FROM(
7.    SELECT T.*,
8.       IF(@last_group_id  =  group_id,  @I:=@I+1,  @I:=1)
  RowNumDesc,
9.      @last_group_id := group_id
10.          FROM TestTable T,(SELECT @last_group_id:=NULL,
  @I:=0)I
11.           ORDER  BY  group_id,  order_id  DESC  /*первая
  сортировка*/
12.                ) T, (SELECT         @last_group_id:=NULL,
  @MaxRowNum:=NULL)I
13.         ORDER BY group_id, order_id /*вторая сортировка*/
14.      )T;
```

group_id	order_id	value	Cnt
1	1	1	3
1	2	2	3
1	3	2	3
2	4	1	3
2	5	2	3

2	6	3	3
3	7	1	3
3	8	2	3
3	11	2	3
4	9	1	1

Функции MAX и MIN вычисляются по аналогии. Приведу только пример для MAX:

6) MAX(value) OVER(PARTITION BY group_id)

```
1.SELECT group_id, order_id, value, MaxVal
2.FROM(
3.  SELECT group_id, order_id, value,
4.     IF(@last_group_id = group_id, @MaxVal, @MaxVal := MaxVal) MaxVal,
5.    @last_group_id := group_id
6.  FROM(
7.    SELECT T.*,
8.       IF(@last_group_id = group_id, GREATEST(@MaxVal, value), @MaxVal:=value) MaxVal,
9.      @last_group_id := group_id
10.         FROM TestTable T,(SELECT @last_group_id:=NULL, @MaxVal:=NULL)I
11.         ORDER BY group_id, order_id DESC
12.       )T,(SELECT @last_group_id:=NULL, @MaxVal:=NULL)I
13.       ORDER BY group_id, order_id
14.     )T;
```

group_id	order_id	value	MaxVal
1	1	1	2
1	2	2	2

1	3	2	2
2	4	1	3
2	5	2	3
2	6	3	3
3	7	1	2
3	8	2	2
3	11	2	2
4	9	1	1

7) COUNT(DISTINCT value) OVER(PARTITION BY group_id)

Интересная вещь, которая отсутствует в MS SQL Server, но её можно вычислить с подзапросом, взяв MAX от RANK. Так же поступим и здесь. В первой сортировке вычислим RANK() OVER(PARTITION BY group_id ORDER BY value DESC), затем во второй сортировке проставим максимальное значение всем строкам в каждом окне:

```
1.SELECT group_id, order_id, value, Cnt
2.FROM(
3.  SELECT group_id, order_id, value,
4.    IF(@last_group_id = group_id, @Rank, @Rank := Rank)
  Cnt,
5.    @last_group_id := group_id
6.  FROM(
7.    SELECT T.*,
8.      IF(@last_group_id = group_id,
9.        IF(@last_value = value, @Rank, @Rank:=@Rank+1)
10.          , @Rank:=1) Rank,
11.          @last_group_id := group_id,
12.          @last_value := value
13.        FROM  TestTable  T,(SELECT  @last_value:=NULL,
  @last_group_id:=NULL, @Rank:=0)I
14.        ORDER BY group_id, value DESC, order_id DESC
```

```
15.        )T,(SELECT @last_group_id:=NULL, @Rank:=NULL)I
16.        ORDER BY group_id, value, order_id
17.      )T;
```

group_id	order_id	value	Cnt
1	1	1	2
1	2	2	2
1	3	2	2
2	4	1	3
2	5	2	3
2	6	3	3
3	7	1	2
3	8	2	2
3	11	2	2
4	9	1	1

Производительность

Для начала сравним по производительности нумерацию строк в запросе с помощью самосоединения и с помощью переменных:

1) Классический способ с самомоединением:

```
1. SELECT COUNT(*)N, T1.*
2. FROM TestTable T1
3.    JOIN TestTable T2 ON T1.order_id >= T2.order_id
4. GROUP BY T1.order_id;
```

Что на 10000 записей в таблице TestTable выдаёт:

Duration / Fetch

16.084 sec / 0.016 sec

2) С использованием переменных:

```
1. SELECT @N:=@N+1 N, T1.*
2. FROM TestTable T1, (SELECT @N := 0)M
3. ORDER BY T1.order_id;
```

Выдаёт:

Duration / Fetch

0.016 sec / 0.015 sec

Результат говорит сам за себя. Однако надо понимать, что вычисленные с помощью переменных значения не оптимально использовать в условиях фильтрации. Сортировка и вычисление будут происходить для ВСЕХ строк, несмотря на то, что в итоге нам нужна только малая их часть.

Рассмотрим более подробно на примере такой задачи:

Вывести по 2 первые строки из таблицы TestTable для каждого значения group_id, отсортированных по order_id.

Вот как эта задача решалась бы в СУБД с поддержкой аналитических функций:

```
1. SELECT group_id, order_id, value
2. FROM(
3.   SELECT *, ROW_NUMBER()OVER(PARTITION BY group_id ORDER
  BY order_id) RowNum
```

```
4.   FROM TestTable
5.)T
6.WHERE RowNum <= 2;
```

Поскольку СУБД «знает», как работает ROW_NUMBER, оптимизатору незачем нумеровать ВСЕ строки, чтобы выбрать первые две. И всё выполнится быстро (при наличии индекса по group_id, order_id, конечно).

В случае с MySQL решение с подобным алгоритмом будет выглядеть так:

```
1.SELECT group_id, order_id, value
2.FROM(
3.  SELECT T.*,
4.     IF(@last_group_id  =  group_id,  @I:=@I+1,  @I:=1)
  RowNum,
5.    @last_group_id := group_id
6.  FROM TestTable T,(SELECT @last_group_id:=NULL, @I:=0)I
7.  ORDER BY group_id, order_id
8.)T
9.WHERE RowNum <= 2;
```

Однако оптимизатор MySQL ничего не знает о том, по каким правилам мы вычисляем поле RowNum. Ему придётся пронумеровать ВСЕ строки, и только потом отобрать нужные.

Теперь представьте, что у нас 1 миллион записей и 20 уникальных значений group_id. Т.е. чтобы выбрать 40 строк, MySQL будет вычислять значение RowNum для миллиона строк! Красивого решения этой задачи одним запросом в MySQL нет. Но можно сначала получить список уникальных значений group_id, например, так:

```
1.SELECT DISTINCT group_id FROM TestTable;
```

Затем средствами любого другого языка программирования сгенерировать запрос вида:

```
1. SELECT * FROM TestTable WHERE group_id=1 ORDER BY order_id
   LIMIT 2
2. UNION ALL
3. SELECT * FROM TestTable WHERE group_id=2 ORDER BY order_id
   LIMIT 2
4. UNION ALL
5. …
6. SELECT * FROM TestTable WHERE group_id=20 ORDER BY
   order_id LIMIT 2;
```

20 лёгких запросов отработают намного быстрее, чем вычисление RowNum для миллиона строк.

Графовые базы данных

Графовая модель данных на логическом уровне представляет собой направленный граф, состоящий из узлов и ребер. Узлы соответствуют объектам базы данных, а ребра – связям между этими объектами. И узлы, и ребра могут обладать свойствами, кроме того, ребра имеют тип, определяющий характер связи. Графовая модель хорошо отражает семантику предметной области с многочисленными связями. Например, пользователи социальной сети могут быть связаны между собой родственными, дружественными, производственными и прочими отношениями.

Графовые СУБД используют графовую модель для описания и манипуляции данными в базе данных. Они относятся к направлению **NoSQL**, которое объединяет множество различных подходов к моделированию данных, отрицающих реляционную модель данных. Однако, в отличии от других моделей, графовые СУБД, как и базы данных SQL, поддерживают ACID-транзакции. Во многом это связано с тем, что данная модель не является агрегатной, и это препятствует её использованию в распределенной форме. Одной из

популярных графовых СУБД является Neo4j, имеющей оригинальный декларативный язык запросов **Cypher**.

Наверняка, прочитав предыдущую пару абзацев, читатель уже задал себе вопрос, почему в учебнике по SQL появилась глава, посвященная графовым базам данных? Дело в том, что, начиная с версии 2017, SQL Server стал поддерживать графовую модель данных!

Графовые базы данных SQL Server

Кардинальным отличием графовых баз данных в SQL Server от графовых баз данных направления NoSQL является то, что для моделирования графа в SQL Server используются таблицы. Это таблицы двух специальных видов. Один вид таблиц используется для создания узлов, и другой – для создания ребер (связей). Отметим, что табличное представление узлов и ребер дает возможность писать запросы к этим таблицам на языке SQL.

Таблица типа узла описывает некую сущность. Экземпляры этой сущности представлены строками таблицы и характеризуются одинаковым набором свойств (столбцами таблицы). Таблица типа ребра определяет характер и направление связи между узлами. Скажем, таблица ToBeFriends (дружить) могла бы описывать дружественную связь между экземплярами одной или разных сущностей.

Вероятно, пора переходить к примерам. Предлагаю взять учебную базу «Окраска», и отобразить реляционную структуру этой базы на структуру графа без потери информации. Это позволит нам писать запросы и сравнивать результаты на двух моделях, представляющих в разной форме одну и ту же информацию.

Итак, у нас есть два типа узлов, представляющих сущности квадрата и баллона, и одна связь между ними, которую можно выразить так: баллон окрашивает квадрат. Направление связи – от баллона к квадрату.

Создадим таблицы узлов:

```
1.--Квадраты
2.CREATE TABLE utqG (
3.    q_id INT PRIMARY KEY,
4.    q_name VARCHAR(35) NOT NULL,
5.) AS NODE;
6.--Баллончики
7.CREATE TABLE utvG (
8.    v_id INT PRIMARY KEY,
9.    v_name VARCHAR(35) NOT NULL,
10.        v_color char(1) NOT NULL
11.      ) AS NODE;
```

Как видите, таблицы создаются аналогично обычным реляционным за исключением указания типа – **AS NODE** (т.е. узел). Теперь посмотрим на структуру созданных таблиц:

```
1.SELECT table_name, column_name, data_type
2.FROM information_schema.COLUMNS
3.WHERE table_name='utqG';
```

table_name	column_name	data_type
utqG	graph_id_AA21DCF7CB44457BB308B21482806B87	bigint
utqG	$node_id_EEFDE0FB86F243E4A6667A5CE470F4F6	nvarchar
utqG	q_id	int
utqG	q_name	varchar

При создании таблицы узлов, помимо пользовательских, автоматически создаются еще два псевдостолбца – *graph_id* и *$node_id*. Давайте наполним таблицу данными и посмотрим, что находится в этих столбцах.

```
1. INSERT INTO utqG(q_id, q_name)
2. SELECT * FROM [sql-ex]..utq;
```

Здесь мы просто берем готовые данные, которые находятся в учебной базе данных с именем *sql-ex*.

```
1. SELECT TOP 2 * FROM utqG;
```

$node_id_EEFDE0FB86F243E4A6667A5CE470F4F6	q_id	q_name
{"type":"node","schema":"dbo","table":"utqG","id":0}	1	Square # 01
{"type":"node","schema":"dbo","table":"utqG","id":1}	2	Square # 02

Столбец *graph_id* отсутствует в выборке. Причиной является то, что этот столбец используется ядром СУБД и недоступен пользователю напрямую. Действительно, если выполнить запрос

```
1. SELECT graph_id FROM utqG;
```

то мы получим сообщение об ошибке:

Недопустимое имя столбца "graph_id".

Столбец *$node_id* является уникальным идентификатором узла, представленным в формате **JSON**. Шестнадцатеричный суффикс в имени столбца делает имя столбца глобально уникальным, однако для доступа к столбцу он не используется. Например,

```
1. SELECT TOP 2 $node_id, q_id FROM utqG;
```

$node_id_EEFDE0FB86F243E4A6667A5CE470F4F6	q_id
{"type":"node","schema":"dbo","table":"utqG","id":0}	1
{"type":"node","schema":"dbo","table":"utqG","id":1}	2

Таблица utvG выглядит аналогично таблице utqG, и поэтому мы не будем представлять тут её содержимое.

Теперь создадим таблицу ребра.

```
1. CREATE TABLE utbG (
2.     b_datetime datetime NOT NULL,
3.     b_vol tinyint NOT NULL,
4. ) AS EDGE;
```

Эта таблица содержит столбцы свойств – время окраски (*B_datetime*) и количество нанесенной из баллончика краски (*b_vol*). Она отличается от таблиц узлов типом – теперь это EDGE, а не NODE. Рассмотрим структуру этой таблицы:

```
1. SELECT column_name, data_type
2. FROM information_schema.COLUMNS WHERE table_name='utbG';
```

column_name	data_type
graph_id_9D6121CFD14948A5B03BBD6A4BDB4774	bigint
$edge_id_EAB5B85BC07649ED89435D6F2A2ACE83	nvarchar
from_obj_id_D0D5B23329B5409895672D5C32283E51	int

from_id_526FECFDEBB84E86B2F7440FF625BCF1	bigint
$from_id_BF1E69FB306E4225A58F4DAFAA8FBE94	nvarchar
to_obj_id_14C309807A224F7BA7BD8F88425A15AB	int
to_id_295F2F2BD42A425CAAD6ACC9F2174BC7	bigint
$to_id_79504454FC184B7F849B1AB28BCCC670	nvarchar
b_datetime	datetime
b_vol	tinyint

Как и для узлов, автоматически были созданы несколько псевдостолбцов, среди которых доступными пользователю являются:

- **$edge_id** – идентификатор ребра, формируется автоматически;
- **$from_id** – идентификатор узла, откуда исходит ребро;
- **$to_id** – идентификатор узла, куда входит ребро.

Заполним и эту таблицу данными. Если мы вспомним структуру исходной таблицы utb, то для каждой строки в столбец *$from_id* мы должны поместить идентификатор того узла из таблицы utvG, для которого *v_id* равен *b_v_id* таблицы utb. Тогда столбец *$to_id* должен содержать идентификатор того узла из таблицы utqG, для которого *q_id* равен, соответственно, *b_q_id* из той же строки таблицы utb. Читателю, наверное, будет проще понять оператор, который вставит описанные данные:

```
1.INSERT INTO utbG(b_datetime, b_vol, $from_id, $to_id)
2.SELECT b_datetime, b_vol,
3.(SELECT $node_id FROM utvG WHERE v_id = orig.b_v_id),
4.(SELECT $node_id FROM utqG WHERE q_id = orig.b_q_id)
5.FROM [sql-ex]..utb orig
```

Теперь мы можем посмотреть на данные в таблице ребра, которая устанавливает связь между двумя узлами – баллончиком с краской и окрашиваемым квадратом.

```
1. SELECT TOP 1 * FROM utbG;
```

Поскольку ширина результирующей таблицы значительно превышает ширину страницы, представим результат в формате «ключ: значение»

$edge_id_EAB5B85BC07649ED89435D6F2A2ACE83: {"type":"edge","schema":"dbo","table":"utbG","id":0}
$from_id_BF1E69FB306E4225A58F4DAFAA8FBE94: {"type":"node","schema":"dbo","table":"utvG","id":49}
$to_id_79504454FC184B7F849B1AB28BCCC670: {"type":"node","schema":"dbo","table":"utqG","id":21}
b_datetime: "2000-01-01 01:13:36.000"
b_vol: 50

Запросы к графовой базе данных

Восстановить вид исходной таблицы Utb можно с помощью обычного SQL-запроса:

```
1. SELECT B.b_datetime, Q.q_id b_q_id, V.v_id b_v_id,
   B.b_vol
2. FROM utbG B JOIN utqG Q ON B.$to_id = Q.$node_id
3. JOIN utvG V ON B.$from_id = V.$node_id;
```

Однако для облегчения навигации по графу в SQL Server добавлена функция **MATCH**, которая задает шаблон для поиска узлов в соответствии со связями. Синтаксис шаблона напоминает образцы, используемые в языке **Cypher**, графовой базы данных Neo4j.

В упрощенной форме шаблон имеет вид:

```
1.<узел, откуда исходит ребро> - (<ребро>) -> <узел, в который входит ребро>
```

Узел представляется именем таблицы или алиасом (псевдонимом). Алиас необходим, когда одна таблица используется в образце несколько раз, например, в случае самосоединений. Шаблоны в функции MATCH можно соединять при помощи оператора AND. Перейдем к примерам. Для начала перепишем предыдущий запрос в «графовом» стиле:

```
1.SELECT B.b_datetime, Q.q_id b_q_id, V.v_id b_v_id, B.b_vol
2.FROM utbG B, utqG Q, utvG V
3.WHERE match(V-(B)->Q);
```

Здесь мы используем алиасы только для сокращения записи. Мы можем поменять местами узлы, но направление связи должно сохраниться (от баллончика к квадрату):

```
1.SELECT B.b_datetime, Q.q_id b_q_id, V.v_id b_v_id, B.b_vol
2.FROM utbG B, utqG Q, utvG V
3.WHERE match(Q<-(B)-V);
```

В отличие от Cypher направление связи является обязательным, т.е. использование функции

```
1.match(V-(B)-Q);
```

будет вызывать ошибку. Cypher более гибок в этом отношении; направление можно не указывать, если оно может быть однозначно определено, например, если связь данного типа есть в одном направлении, но нет в другом.

Найти квадраты, которые окрашивались красной краской. Вывести идентификатор квадрата и объем красной краски.

Реляционная схема

```
1. SELECT b_q_id, SUM(b_vol) qty
2. FROM utB JOIN utV ON B_V_ID =V_ID
3. WHERE V_COLOR ='R'
4. GROUP BY b_q_id;
```

Графовая схема

```
1. SELECT q_id, SUM(b_vol) qty
2. FROM utbG B, utvG V, utqG Q
3. WHERE match(V-(B)->Q) AND V_COLOR ='R'
4. GROUP BY q_id;
```

Найти квадраты, которые окрашивались как красной, так и синей краской. Вывести: название квадрата.

Реляционная схема

```
1. SELECT q_name FROM utQ JOIN (
2. SELECT b_q_id FROM utB JOIN utV ON B_V_ID =V_ID
3. WHERE V_COLOR ='R'
4. INTERSECT
5. SELECT b_q_id FROM utB JOIN utV ON B_V_ID =V_ID
6. WHERE V_COLOR ='B'
7.) X ON Q_ID=b_q_id;
```

Графовая схема

```
1.SELECT DISTINCT q_name
2.FROM utbG B1,utbG B2, utvG VR, utvG VB, utqG Q
3.WHERE
4.match(VR-(B1)->Q<-(B2)-VB)  AND  VR.V_COLOR  ='R'  AND
  VB.V_COLOR ='B';
```

Обратите внимание, что для в общем случае разных узлов, используемых в шаблоне, требуется добавлять соответствующую таблицу в предложение FROM (как и таблицу связи). Если же подразумевается один и тот же узел, как в нашем случае с квадратом, то мы можем использовать этот узел для двусторонней связи. Предикат

```
1.match(VR-(B1)->Q<-(B2)-VB)
```

можно было также записать в таком виде

```
1.match(VR-(B1)->Q) AND match(VB-(B2)->Q)
```

или в таком

```
1.match(VR-(B1)->Q AND VB-(B2)->Q)
```

DISTINCT в этих запросах необходим, поскольку возможны дубликаты, если квадрат окрашивался несколькими красными (и/или синими) баллончиками.

Найти квадраты, которые окрашивались всеми тремя цветами.

Реляционная схема

Можно было бы добавить еще один запрос к предыдущему решению с помощью INTERSECT, однако давайте представим здесь более эффективное решение, опирающееся на тот факт, что количество уникальных цветов, которыми окрашивался квадрат, равно 3:

```
1.SELECT q_name FROM utQ JOIN (
2.SELECT b_q_id
```

```
3.FROM utB JOIN utV ON B_V_ID =V_ID
4.GROUP BY b_q_id
5.HAVING COUNT(DISTINCT v_color)=3
6.) X ON q_id=b_q_id;
```

Графовая схема

```
1.SELECT DISTINCT Q.q_name
2.FROM utbG B1,utbG B2,utbG B3, utvG VR, utvG VB, utvG VG,
  utqG Q
3.WHERE
4.match(VR-(B1)->Q<-(B2)-VB AND VG-(B3)->Q)
5.AND VG.V_COLOR ='G' AND VR.V_COLOR ='R' AND VB.V_COLOR
  ='B';
```

Найти баллончики, которыми окрашивали более одного квадрата.

Реляционная схема

```
1.
  SELECT v_name FROM
2.utB JOIN utV ON B_V_ID =V_ID
3.GROUP BY b_v_id,v_name
4.HAVING COUNT(DISTINCT b_q_id)>1;
```

Графовая схема

```
1.SELECT DISTINCT v_name
2.FROM utbG B1,utbG B2, utvG V, utqG Q,utqG Q2
3.WHERE
4.match(Q2<-(B2)-V-(B1)->Q) AND Q.q_id <> Q2.q_id;
```

Условие *match(Q2<-(B2)-V-(B1)->Q)* утверждает, что один баллон участвовал в двух окрасках, в то время как условие Q.q_id <> Q2.q_id говорит о том, что при этом окрашивались разные квадраты.

Предварительные выводы

На первый взгляд, запросы к реляционной и графовой схемам не так уж радикально отличаются. Более того, синтаксиса графовой базы данных можно не придерживаться, оставаясь в рамках языка SQL. Однако не стоит судить по рассмотренным примерам, поскольку преимущество графовых баз данных проявляется для тех предметных областей, которые характеризуются сложными связями между данными.

Приложения

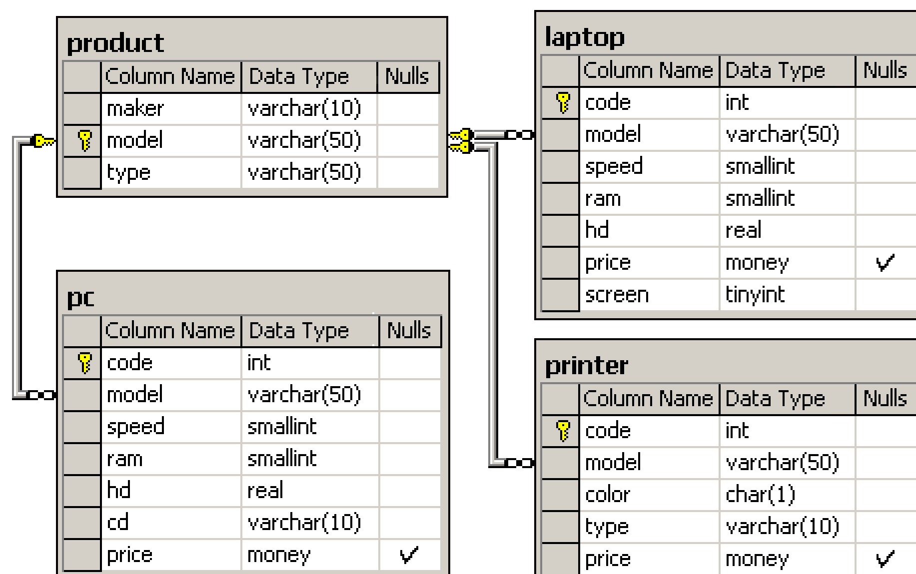
Здесь приводятся описания всех учебных баз данных, используемых для решения задач на сайте SQL-EX.RU, а также список всех задач обучающего этапа тестирования. Задачи, рассмотренные в книге, отмечены знаком «+».

Приложение 1. Описание учебных баз данных

Компьютерная фирма

Схема базы данных состоит из четырех отношений:

```
1.Product (maker, model, type)
2.PC (code, model, speed, ram, hd, cd, price)
3.Laptop (code, model, speed, ram, hd, screen, price)
4.Printer (code, model, color, type, price)
```



Отношение Product представляет производителя (maker), номер модели (model) и тип (PC — ПК, Laptop — портативный компьютер или Printer — принтер). Предполагается, что номера моделей уникальны для всех производителей и типов продуктов.

В отношении PC для каждого номера модели, обозначающего ПК, указаны скорость — speed (процессора в мегагерцах), общий объем RAM — ram (в мегабайтах), размер диска — hd (в гигабайтах), скорость считывающего устройства CD (например, 4x) и цена — price.

Отношение Laptop аналогично отношению PC за исключением того, что вместо скорости CD-привода содержится размер экрана — screen (в дюймах).

В отношении Printer для каждой модели принтера указывается, является ли он цветным — color ('y', если цветной), а также тип принтера — type (лазерный — Laser, струйный — Jet или матричный — Matrix) и цена — price.

База данных «Компьютерная фирма»

Схема БД состоит из четырех таблиц (рис.1.1):

- Product(maker, model, type)
- PC(code, model, speed, ram, hd, cd, price)

- Laptop(code, model, speed, ram, hd, screen, price)
- Printer(code, model, color, type, price)

Таблица Product представляет производителя (maker), номер модели (model) и тип (PC — ПК, Laptop — портативный компьютер или Printer — принтер). Предполагается, что в этой таблице номера моделей уникальны для всех производителей и типов продуктов. В таблице PC для каждого номера модели, обозначающего ПК, указаны скорость процессора — speed (МГерц), общий объем оперативной памяти - ram (Мбайт), размер диска — hd (в Гбайт), скорость считывающего устройства - cd (например, '4x') и цена — price. Таблица Laptop аналогична таблице PC за исключением того, что вместо скорости CD-привода содержит размер экрана — screen (в дюймах). В таблице Printer для каждой модели принтера указывается, является ли он цветным — color ('y', если цветной), тип принтера — type (лазерный — Laser, струйный — Jet или матричный — Matrix) и цена — price.

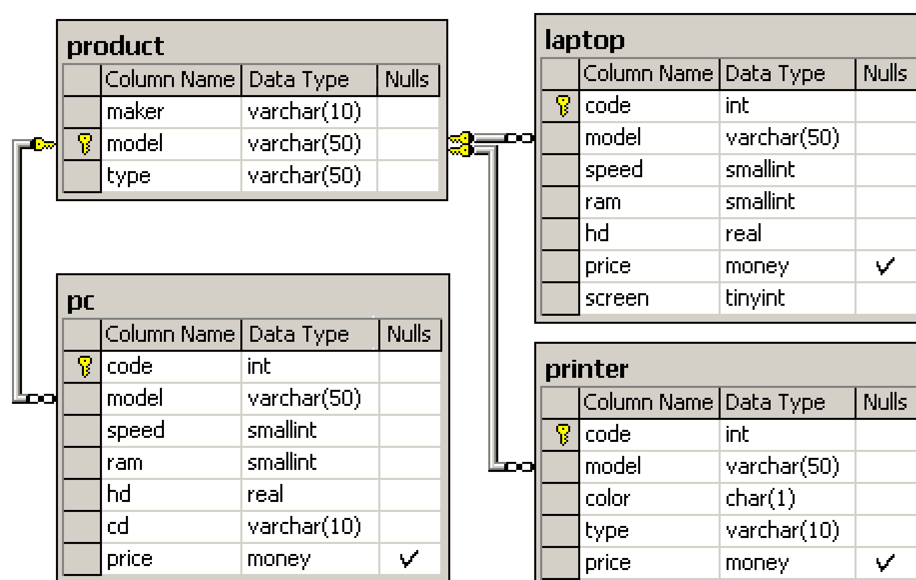


Рис. 1.1. Схема базы данных «Компьютерная фирма»

Дополнительную информацию можно извлечь из представленной на рис. 1.1 логической схемы данных. Таблицы по типам продукции (ПК, портативные компьютеры и принтеры) содержат внешний ключ (model) к таблице Product. Связь «один-ко-многим» означает, что в каждой из этих таблиц может отсутствовать модель, имеющаяся в таблице Product. С другой стороны, модель с одним и тем же номером может встречаться в такой таблице несколько раз, причем даже с полностью идентичными техническими характеристиками, так как первичным ключом здесь является столбец code. Последнее требует пояснения, так как разные люди вкладывают в понятие модели разный смысл. В рамках данной схемы считается, что модель — это единство производителя и технологии. Например, одинаковые модели могут комплектоваться технологически идентичными накопителями, но разной емкости, скажем, 60 и 80 Гбайт. В частности, это означает, что допустимо присутствие в таблице PC двух ПК с одинаковыми номерами модели, но по разной цене.

На языке предметной области данная схема может означать, что в таблице Product содержится информация обо всех известных поставщиках рассматриваемой продукции и моделях, которые они поставляют, а в остальных таблицах находятся имеющиеся в наличии (или продаже) модели. Поэтому вполне возможна ситуация, когда имеется поставщик (maker) с моделями, ни одной из которых нет в наличии.

Фирма вторсырья

Фирма имеет несколько пунктов приема вторсырья. Каждый пункт получает деньги для их выдачи сдатчикам вторсырья.

Income

	Column Name	Data Type	Nulls
🔑	code	int	
	point	tinyint	
	[date]	datetime	
	inc	smallmoney	

Outcome

	Column Name	Data Type	Nulls
🔑	code	int	
	point	tinyint	
	[date]	datetime	
	out	smallmoney	

Income_o

	Column Name	Data Type	Nulls
🔑	point	tinyint	
🔑	[date]	datetime	
	inc	smallmoney	

Outcome_o

	Column Name	Data Type	Nulls
🔑	point	tinyint	
🔑	[date]	datetime	
	out	smallmoney	

Сведения о получении денег на пункт приема записываются в таблицу:

```
1.Income_o (point, date,inc)
```

Первичным ключом является {point, date}, то есть прием денег (inc) производится не чаще одного раза в день. Сведения о выдаче денег за вторсырье записывается в таблицу:

```
1.Outcome_o (point, date, out)
```

В этой таблице аналогичный первичный ключ {point, date} гарантирует отчетность каждого пункта о выданных деньгах (out) не чаще одного раза в день.

В случае, когда приход и расход денег может фиксироваться несколько раз в день, используются таблицы (первичный ключ — code):

```
1.Income (code, point, date, inc)
2.Outcome (code, point, date, out)
```

База данных

«Фирма вторсырья»

Фирма занимается приемом вторсырья и имеет несколько пунктов приема. Каждый пункт получает деньги для их выдачи сдатчикам в обмен на сырье. Фактически, на схеме представлены две базы данных. В каждой задаче по этой схеме используется только одна пара таблиц (либо с суффиксом «_o», либо без него).

В таблицах Income_o и Outcome_o первичным ключом является пара атрибутов {point, date} — номер пункта приема и дата. Этот ключ должен моделировать ситуацию, когда сведения о получении денег на приемном пункте и их выдаче сдатчикам записываются в базу данных не чаще одного раза в день.

Income

Key	Column Name	Data Type	Nulls
🔑	code	int	
	point	tinyint	
	[date]	datetime	
	inc	smallmoney	

Outcome

Key	Column Name	Data Type	Nulls
🔑	code	int	
	point	tinyint	
	[date]	datetime	
	out	smallmoney	

Income_o

Key	Column Name	Data Type	Nulls
🔑	point	tinyint	
🔑	[date]	datetime	
	inc	smallmoney	

Outcome_o

Key	Column Name	Data Type	Nulls
🔑	point	tinyint	
🔑	[date]	datetime	
	out	smallmoney	

Рис. 2.1. Схема базы данных «Фирма вторсырья»

Примечание.

Значения данных в столбце date не содержат времени, например, 2001-03-22 00:00:00.000. К сожалению, использование для этого столбца типа данных datetime может вызвать непонимание, поскольку очевидно, что учет времени не позволит ограничить многократный ввод значений с одной и той же датой (и номером пункта), но отличающихся временем дня. Этот недостаток, связанный с отсутствием отдельных типов данных для даты и времени, уже преодолен в версии SQL Server 2008. При использовании же SQL Server 2000 обеспечить правильность ввода можно при

помощи, например, следующего ограничения (CK_Income_o):

```
1.ALTER TABLE Income_o ADD
2.CONSTRAINT PK_Income_o PRIMARY KEY
3. (
4. [point],
5. [date]
6. ),
7.CONSTRAINT CK_Income_o CHECK
8. (
9. DATEPART(hour,[date]) + DATEPART(minute,[date]) +
10.       DATEPART(second,[date]) +
  DATEPART(millisecond,[date]) = 0
11.       );
```

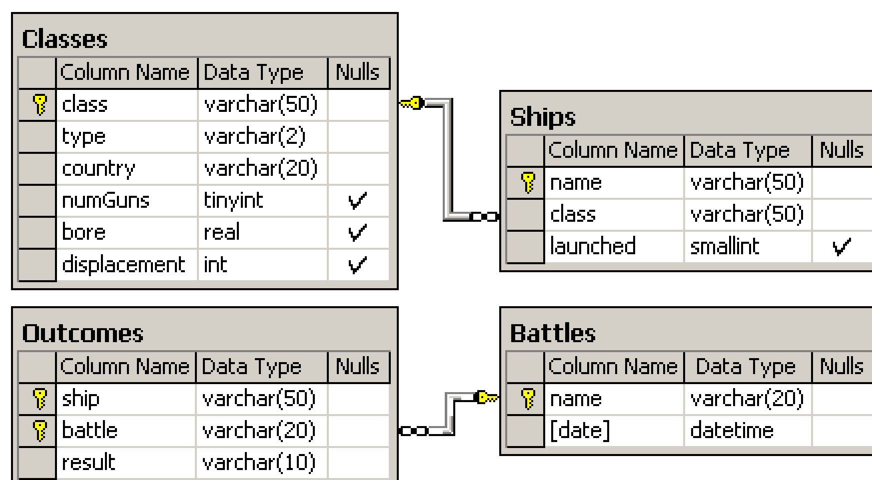
Это ограничение (сумма часов, минут, секунд и миллисекунд равна нулю) не позволит ввести какое-либо время, отличное от 00:00:00.000. При таком ограничении первичный ключ на данной таблице будет действительно гарантировать наличие лишь одной записи в день для каждой точки.

Таблица Income_o (point, date, inc) содержит информацию о поступлении денежных сумм (inc) на пункт приема (point). Аналогичная таблица — Outcome_o (point, date, out) — служит для контроля расхода денежных средств (out).

Вторая пара таблиц — Income (code, point, date, inc) и Outcome (code, point, date, out) — моделирует ситуацию, когда приход и расход денег может фиксироваться несколько раз в день. Следует отметить, что если записывать в последние таблицы только дату без времени (что и имеет место), то никакая естественная комбинация атрибутов не может служить первичным ключом, поскольку суммы денег также могут совпадать. Поэтому нужно либо учитывать время, либо добавить искусственный ключ. Мы использовали второй вариант, добавив целочисленный столбец code только для того, чтобы обеспечить уникальность записей в таблице.

Корабли

Рассматривается база данных кораблей, участвовавших во второй мировой войне.



Имеются следующие отношения:

```
1.Classes (class, type, country, numGuns, bore,
  displacement)
2.Ships (name, class, launched)
3.Battles (name, date)
4.Outcomes (ship, battle, result)
```

Корабли в «классах» построены по одному и тому же проекту, и классу присваивается либо имя первого корабля, построенного по данному проекту, либо названию класса дается имя проекта, которое не совпадает ни с одним из кораблей. Корабль, давший название классу, называется головным.

Отношение **Classes** содержит имя класса — class, тип — type (bb для боевого (линейного) корабля или bc для боевого крейсера), страну — country, в которой построен корабль, число главных орудий — numGuns, калибр орудий (диаметр ствола орудия в дюймах) — bore и водоизмещение (вес в тоннах) — displacement.

В отношении **Ships** записаны название корабля — name, имя его класса — class и год спуска на воду — launched.

В отношение **Battles** включены название — name и дата битвы — date, в которой участвовали корабли, а в отношении **Outcomes** — результат участия данного корабля в битве — result (потоплен — sunk, поврежден — damaged или невредим — ok).

Примечание:

1) В отношение Outcomes могут входить корабли, отсутствующие в отношении Ships. 2) Потопленный корабль в последующих битвах участия не принимает.

База данных «Корабли»

Рассматривается база данных кораблей, участвовавших в морских сражениях второй мировой войны. Имеются следующие отношения:

```
1.Classes (class, type, country, numGuns, bore,
  displacement)
2.Ships (name, class, launched)
3.Battles (name, date)
4.Outcomes (ship, battle, result)
```

Корабли в «классах» построены по одному и тому же проекту. Классу присваивается либо имя первого корабля, построенного по данному проекту, либо названию класса дается имя проекта, которое в этом случае не совпадает с именем ни одного из кораблей. Корабль, давший название классу, называется головным.

Атрибутами отношения Classes являются имя класса (class), тип (значение bb используется для обозначения боевого или линейного корабля, а bc для боевого крейсера), страну (country), которой принадлежат корабли данного класса, число главных орудий (numGuns), калибр орудий (bore — диаметр ствола орудия в дюймах) и водоизмещение в тоннах (displacement).

В отношение Ships записывается информация о кораблях: название корабля (name), имя его класса (class) и год спуска на воду (launched).

В отношение Battles включены название (name) и дата битвы (date), в которой участвовал корабль.

Отношение Outcomes используется для хранения информации о результатах участия кораблей в битвах, а именно, имя корабля (ship), название сражения

(battle) и чем завершилось сражение для данного корабля (потоплен — sunk, поврежден — damaged или невредим — ok).

Примечание:

В отношение Outcomes могут входить корабли, отсутствующие в отношении Ships.

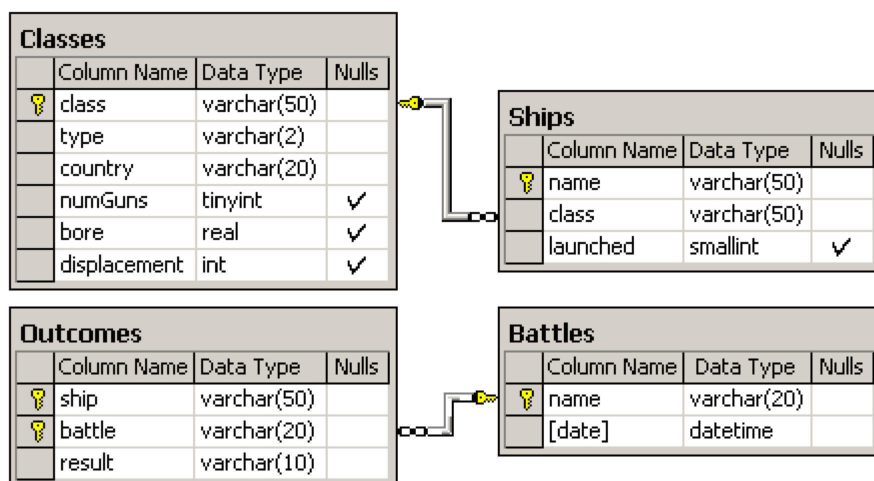


Рис. 3.1. Схема базы данных «Корабли»

Отметим несколько моментов, на которые следует обратить внимание при анализе схемы на рис. 3.1. Таблица Outcomes имеет составной первичный ключ {ship, battle}. Это ограничение не позволит ввести в базу данных дважды один и тот же корабль, принимавший участие в одном и том же сражении. Однако допустимо неоднократное присутствие одного и того же корабля в данной таблице, что означает участие корабля в нескольких битвах. Класс корабля определяется из таблицы Ships, которая имеет внешний ключ (class) к таблице Classes.

Особенностью данной схемы, которая усложняет логику запросов и служит причиной ошибок при решении задач, является то, что таблицы Outcomes и Ships никак не связаны, то есть в таблице результатов сражений могут находиться корабли, отсутствующие в таблице Ships. На основании этого, казалось бы, можно сделать вывод о том, что для таких кораблей их класс неизвестен, а, следовательно, неизвестны и все технические характеристики. Это не совсем так. Как следует из описания предметной области, имя головного корабля совпадает с именем класса, родоначальником которого он является. Поэтому если имя корабля из таблицы Outcomes совпадает с именем класса в таблице Classes, то однозначно можно сказать, что это головной корабль, и, следовательно, все его характеристики нам известны.

Каждый знает, как улучшить эту «плохую» схему: связать таблицы Ships и Outcomes по имени корабля, при этом столбец ship в Outcomes становится внешним ключом к таблице Ships. Безусловно, это так, однако не следует забывать, что в реальной ситуации не вся информация может быть доступна. Например, имеется архивная информация о кораблях, участвовавших в том или ином сражении, без указания классов этих кораблей. При наличии обсуждаемой связи сначала будет необходимо внести такой корабль в таблицу Ships, при этом столбец class должен допускать NULL-значения.

С другой стороны, что нам мешает ввести головной корабль, который попал в таблицу Outcomes, также и в таблицу Ships? В принципе ничего, так как год спуска на воду не является обязательной информацией. По этому поводу следует заметить, что администратор базы данных и разработчик приложения, как правило, разные люди. Не всегда разработчик приложения и его пользователи имеют права на модификацию данных.

Плохая структура еще не означает, что из нее нельзя извлечь достоверную информацию, чем собственно мы и занимаемся, решая предлагаемые задачи. Что касается учебных целей, то работа с такой структурой даст значительно больше в освоении языка, чем структура «хорошая», так как заставит писать более сложные запросы и научит учитывать дополнительные обстоятельства, накладываемые схемой. Этим видимо и руководствовались авторы этой схемы данных [2]. Кроме того, запросы, написанные для «плохой» схемы, будут давать правильные результаты и после улучшения структуры (хотя и станут менее эффективными), то есть тогда, когда вся информация станет доступной, и мы сможем установить связь между таблицами Ships и Outcomes.

Наконец, стоит обратить внимание на то, что столбец launched в таблице Ships допускает NULL-значения, то есть нам может быть неизвестен год спуска на воду того или иного корабля. То же самое мы можем сказать о кораблях из Outcomes, отсутствующих в Ships.

Что ж, перейдем к решению задач. Заметим лишь, что в настоящей главе мы уже не рассматриваем совсем простые задачи (хотя они имеются и для этой схемы), которых должно было хватить вам в первой главе.

Аэрофлот

Схема базы данных состоит из четырех отношений:

```
1.Company (ID_comp, name)
2.Trip (trip_no, id_comp, plane, town_from, town_to,
  time_out, time_in)
3.Passenger (ID_psg, name)
4.Pass_in_trip (trip_no, date, ID_psg, place)
```

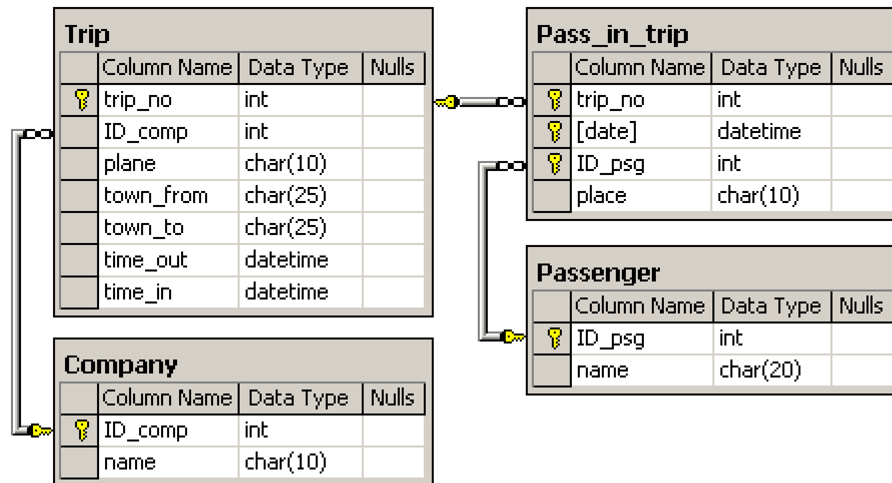


Таблица Company содержит идентификатор ID_comp и название компании — name, осуществляющей перевозку пассажиров.

Таблица Trip содержит информацию о рейсах: номер рейса — trip_no, идентификатор компании id_comp, тип самолета — plane, город отправления — town_from, город прибытия — town_to, время отправления — time_out и время прибытия — time_in.

Таблица Passenger содержит идентификатор — id_psg и имя пассажира — name.

Таблица Pass_in_trip содержит информацию о полетах: номер рейса — trip_no, дату вылета — date, идентификатор пассажира и место — place, на котором он сидел во время полета. При этом следует иметь в виду, что

- рейсы выполняются ежедневно, а длительность полета любого рейса менее суток;
- время и дата учитывается относительно одного часового пояса;
- среди пассажиров могут быть однофамильцы (одинаковые значения поля name, например, Bruce Willis);
- номер места в салоне — это число с буквой; число определяет номер ряда, буква (a — d) — место в ряду слева направо в алфавитном порядке;
- связи и ограничения показаны на схеме данных.

Окраска

Схема базы данных состоит из трех отношений (рис. П.5):

```
1.utQ (Q_ID, Q_NAME)
2.utV (V_ID, V_NAME, V_COLOR)
3.utB (B_Q_ID, B_V_ID, B_VOL, B_DATETIME)
```

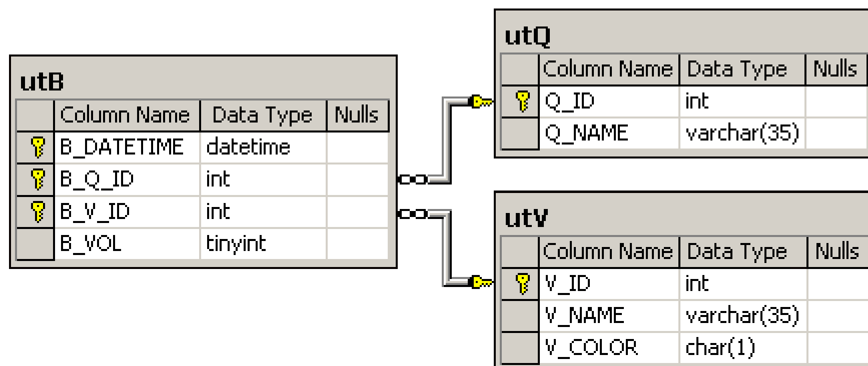


Таблица utQ содержит идентификатор — Q_ID и название квадрата — Q_NAME, цвет которого первоначально черный.

Таблица utV содержит идентификатор — V_ID, название — V_NAME и цвет — V_COLOR баллончика с краской.

Таблица utB содержит информацию об окраске квадрата баллончиком: B_Q_ID — идентификатор квадрата, B_V_ID — идентификатор баллончика, B_VOL — количество краски и B_DATETIME — время окраски.

При этом следует иметь в виду, что:

- баллончики с краской могут быть трех цветов — красный V_COLOR = 'R', зеленый V_COLOR = 'G', голубой V_COLOR = 'B' (латинские буквы);
- объем баллончика равен 255 и первоначально он полный;
- цвет квадрата определяется по правилу RGB, то есть R = 0, G = 0, B = 0 — черный, R = 255, G = 255, B = 255 — белый;
- запись в таблице закрасок utB уменьшает количество краски в баллончике на величину B_VOL и соответственно увеличивает количество краски в квадрате на эту же величину;
- значение 0 < B_VOL <= 255;

- количество краски одного цвета в квадрате не превышает 255, а количество краски в баллончике не может быть меньше или равно нулю.

Приложение 2.
Список задач

Здесь перечислены только задачи, рассматриваемые в книге. На сайте SQL-EX.RU, помимо них, есть много других задач и, кроме того, регулярно добавляются новые.

№	База	Уровень	Задача	Этап	ПиР
1	П1	1	Найдите номер модели, скорость и размер жесткого диска для всех ПК стоимостью менее 500 долларов. Вывести: model, speed и hd	Обуч.	
2	П1	1	Найдите производителей принтеров. Вывести: maker	Обуч.	+
3	П1	1	Найдите номер модели, объем памяти и размеры экранов портативных компьютеров, цена которых превышает 1000 дол.	Обуч.	
5	П1	1	Найдите номер модели, скорость и размер жесткого диска ПК, имеющих 12x или 24x CD-приводы и цену менее 600 долларов	Обуч.	
6	П1	2	Укажите производителя и скорость портативных компьютеров с жестким	Обуч.	+

			диском объемом не менее 10 Гбайт		
7	П1	2	Найдите номера моделей и цены всех продуктов (любого типа) выпущенных производителем B (латинская буква)	Обуч.	+
8	П1	2	Найдите производителя, продающего ПК, но не портативные компьютеры	Обуч.	+
10	П1	1	Найдите принтеры, имеющие самую высокую цену. Вывести: model, price	Обуч.	+
11	П1	1	Найдите среднюю скорость ПК	Обуч.	+
13	П1	1	Найдите среднюю скорость ПК, выпущенных производителем A	Обуч.	
15	П1	2	Найдите размеры жестких дисков, совпадающих у двух и более PC. Вывести: HD	Обуч.	+
16	П1	2	Найдите пары моделей PC, имеющих одинаковые скорость и RAM. В результате каждая пара указывается только один раз, то есть (i,j), но не (j,i), Порядок вывода: модель с большим номером, модель с меньшим номером, скорость и RAM	Обуч.	+
17	П1	2	Найдите портативные компьютеры, скорость которых меньше скорости любого из ПК. Вывести: type, model, speed	Обуч.	+
18	П1	2	Найдите производителей самых дешевых цветных	Обуч.	+

			принтеров. Вывести: maker, price		
20	П1	2	Найдите производителей, выпускающих по меньшей мере три различных модели ПК. Вывести: Maker, число моделей	Обуч.	

№	База	Уровень	Задача	Этап	ПиР
23	П1	3	Найдите производителей, которые производили бы как ПК со скоростью не менее 750 МГц, так и ПК-блокноты со скоростью не менее 750 МГц. Вывести: Maker	Обуч.	+
24	П1	3	Перечислите номера моделей любых типов, имеющих самую высокую цену по всей имеющейся в базе данных продукции	Обуч.	
25	П1	3	Найдите производителей принтеров, которые производят ПК с наименьшим объемом RAM и с самым быстрым процессором среди всех ПК, имеющих наименьший объем RAM. Вывести: Maker	Обуч.	
26	П1	3	Найдите среднюю цену ПК и Портативных компьютеров, выпущенных производителем A (латинская буква). Вывести: одна общая средняя цена	Обуч.	+
27	П1	3	Найдите средний размер диска ПК каждого из тех производителей, которые	Обуч.	+

			выпускают и принтеры. Вывести: maker, средний размер HD		
71	П1	1	Найти тех производителей ПК, все модели ПК которых имеются в таблице PC	Обуч.	
30	П2	3	В предположении, что приход и расход денег на каждом пункте приема фиксируется произвольное число раз (в обе таблицы добавлен первичный ключ code), написать запрос с выходными данными (point, date, out, inc), в котором каждому пункту за каждую дату соответствует одна строка	Обуч.	+
59	П2	3	Посчитать остаток денежных средств на каждом пункте приема для базы данных с отчетностью не чаще одного раза в день. Вывод: пункт, остаток.	Обуч.	+
60	П2	1	Посчитать остаток денежных средств на начало дня 15.04.2001 на каждом пункте приема для базы данных с отчетностью не чаще одного раза в день. Вывод: пункт, остаток.	Обуч.	+
32	П3	3	Одной из характеристик корабля является половина куба калибра его главных орудий (mw). С точностью до 2 десятичных знаков определите среднее значение mw для кораблей каждой страны, у которой есть корабли в базе данных.	Обуч.	

№	База	Уровень	Задача	Этап	ПиР
37	ПЗ	2	Найдите классы, в которые входит только один корабль из базы данных (учесть также корабли в Outcomes)	Обуч.	+
38	ПЗ	2	Найдите страны, имевшие когда-либо классы обычных боевых кораблей ('bb') и имевшие когда-либо классы крейсеров ('bc').	Обуч.	
39	ПЗ	3	Найдите корабли, «сохранившиеся для будущих сражений»; то есть выведенные из строя в одной битве (damaged), они участвовали в другой	Обуч.	+
46	ПЗ	3	Укажите названия, водоизмещение и число орудий кораблей, участвовавших в сражении при Гвадалканале (Guadalcanal)	Обуч.	+
51	ПЗ	3	Найдите названия кораблей, имеющих наибольшее число орудий среди всех кораблей такого же водоизмещения (учесть корабли из таблицы Outcomes)	Обуч.	+
53	ПЗ	1	Определите среднее число орудий для классов линейных кораблей. Получить результат с точностью до двух десятичных знаков	Обуч.	+

54	ПЗ	2	С точностью до двух десятичных знаков определите среднее число орудий всех линейных кораблей (учесть корабли из таблицы Outcomes)	Обуч.	+
55	ПЗ	1	Для каждого класса определите год, когда был спущен на воду первый корабль этого класса. Если год спуска на воду головного корабля неизвестен, определите минимальный год спуска на воду кораблей этого класса. Вывести: класс, год	Обуч.	+
56	ПЗ	3	Для каждого класса определите число кораблей этого класса, потопленных в сражении. Вывести: класс и число потопленных кораблей	Обуч.	+
57	ПЗ	3	Для классов, имеющих потери в виде потопленных кораблей и не менее трех кораблей в базе данных, вывести имя класса и число потопленных кораблей	Обуч.	+

70	П3	3	Укажите сражения, в которых участвовало, по меньшей мере, три корабля одной и той же страны.	Обуч.	+
77	П4	2	Определить дни, когда было выполнено максимальное число рейсов из Ростова ('Rostov'). Вывод: число рейсов, дата.	Обуч.	-
93	П4	2	Для каждой компании, перевозившей пассажиров, подсчитать время, которое провели в полете самолеты с пассажирами. Вывод: название компании, время в минутах.	Обуч.	-
11	П4	2	Среди пассажиров, которые пользовались услугами не менее двух авиакомпаний, найти тех, кто совершил одинаковое количество полётов самолетами каждой из этих авиакомпаний. Вывести имена таких пассажиров.	Рейтинг	-
(-2)	П3	2	Для каждой страны определить год, когда на воду было	Рейтинг	+

			спущено максимальное количество ее кораблей. В случае, если окажется несколько таких лет, взять минимальный из них. Вывод: страна, количество кораблей, год		
17	П3	1	Найдите названия всех тех кораблей из базы данных, о которых можно определенно сказать, что они были спущены на воду до 1941 г.	Рейтинг	+

Приложение 3. Хроники Торуса

На сайте sql-ex.ru предлагается несколько задач, связанных с мифической планетой **Торус**. В рамках этого приложения дается введение в топологию планеты и объясняются необходимые термины.

Тор – это геометрическая фигура, представляющая собой поверхность вращения в форме бублика (см. http://en.wikipedia.org/wiki/Torus).

К сожалению, из-за прекращения поддержки Adobe Flash player наша чудная мультимедиа-презентация (навигатор по Торусу) больше не поможет вам точнее понять взаимное расположение стран TIJ, чтобы успешнее решать соответствущие задачи на sql-ex.ru. :-)

Кнопки клавиатуры (qwe, asd, ->, <-) служили для организации вращения тора, а кнопки со стрелками вверх и вниз использовались для зуммирования. Клик на страну на торе вызывал перерисовку прямоугольной карты Меркадота.

Планета Торус

Планета **Торус** имеет тороидальную поверхность. Если эту поверхность развернуть (спроецировать) на плоскость в координатах Меркадота, то страны на этой карте будут выглядеть как ячейки "таблицы" с числом столбцов равным 7. Согласно 28-ричной системе исчисления, принятой на планете, страны именуются следующим образом:

T00	T01	...	T06
T10	T11	...	T16
...			
TA0	TA1	...	TA6
...			
TR0	TR1	...	TR6

Меркадот – знаменитый путешественник и картограф, живший в стране T00 и разработавший карты планеты, наименее искажающие углы и расстояния. Все страны на Торусе примерно равноправны с точки зрения топологии, поэтому Меркадот разработал единую систему карт для планеты Торус. В этой системе схема карты для страны TR6, например, выглядит следующим образом:

TR6	TR0	...	TR5
T06	T00	...	T05
...			
TQ6	TQ0	...	TQ5

В силу особенностей тороидальной поверхности оказывается, что страна T00, как и всякая другая страна на планете, граничит с 8-ю другими странами, а вовсе не с тремя.

На Торусе для обозначения соседних стран используются единообразные аббревиатуры. Так, например, в стране T00 о стране TR0 всегда говорили как о северном (NN) соседе, о TR6 как о северо-западном (NW) соседе и так далее (следуя против хода часовой стрелки). Подобным образом (NN, NW, WW, SW, SS, SE, EE, NE) обозначают своих соседей и другие страны. Возможно, это связано с направлением магнитных полей планеты.

В результате развития цивилизации в торусианской галактике было обнаружено немало тороидальных планет, названия которых имеют следующие обозначения в астрономическом каталоге: Torus MxN, M - число областей по вертикали (широте), а N - число областей по горизонтали (долготе), т.е. общее число областей на такой планете равно MxN. При этом области не всегда представляют собой страны, как это имеет место на планете Торус, которая в данной классификации обозначается как Torus 28x7.

До последнего времени самой большой (по числу областей) из обнаруженных планет являлся Torus 36x36. Для описания ее топологии торусианцы придумали 36-ричную математическую систему исчисления BASE 36. Согласно этой системе, области на Торус 36x36 имеют аббревиатуры от T00 до TZZ.

С тех пор как среди участников соревнований на сайте sql-ex.ru появились торусианцы, стала доступной новая информация об их галактике **Лорус** (**Lorus**). Например, оказалось, что Torus 36x36 - не самая большая планета этой системы. В целях культурного обмена, немало фактологических данных из исторических хроник Torus было занесено в основные и проверочные базы сайта (Painting, Ships и прочих).

Из условий задач на sql-ex становится известным, что единицей длины на Торусе является торометр, а единой валютой – торобакс. Основой жизни там является вода и ее разновидность – лед. В условиях задач приводятся основные сведения о физико-химических характеристиках, экономических и политических системах планет Torus MxN.

Рассмотрим задачу про Торус, в которой мы вообще не будем привязываться ни к одной из “обычных” баз, и предложим следующее “универсальное” условие.

Соседние страны на Торусе 3x7

Для каждой из стран планеты Торус 3x7:

```
T00   T01   ...   T06
T10   T11   ...   T16
T20   T21   ...   T26
```

найти список всех соседних стран, с которыми она имеет границу ненулевой длины. Решение привести в виде таблицы из пяти колонок: [State] [NN] [WW] [SS] [EE].

Подсказка. Вот как выглядит ответ для обычного (нетороидального) случая.

State	NN	WW	SS	EE
T00	NULL	NULL	T10	T01
T01	NULL	T00	T11	T02
T02	NULL	T01	T12	T03
T03	NULL	T02	T13	T04
T04	NULL	T03	T14	T05
T05	NULL	T04	T15	T06
T06	NULL	T05	T16	NULL

T10	T00	NULL	T20	T11
T11	T01	T10	T21	T12
T12	T02	T11	T22	T13
T13	T03	T12	T23	T14
T14	T04	T13	T24	T15
T15	T05	T14	T25	T16
T16	T06	T15	T26	NULL
T20	T10	NULL	NULL	T21
T21	T11	T20	NULL	T22
T22	T12	T21	NULL	T23
T23	T13	T22	NULL	T24
T24	T14	T23	NULL	T25
T25	T15	T24	NULL	T26
T26	T16	T25	NULL	NULL

Решение для тороидального случая.

```
1.SELECT
2. 'T'+CAST(i  AS varchar)+CAST(j  AS varchar) [State]
3.,'T'+CAST(k1 AS varchar)+CAST(j1 AS varchar) [NN]
4.,'T'+CAST(i2 AS varchar)+CAST(k2 AS varchar) [WW]
5.,'T'+CAST(k4 AS varchar)+CAST(j4 AS varchar) [SS]
6.,'T'+CAST(i3 AS varchar)+CAST(k3 AS varchar) [EE]
7.FROM (SELECT j=0 UNION ALL SELECT 1 UNION ALL SELECT 2
  UNION ALL SELECT 3 UNION ALL
8.         SELECT 4 UNION ALL SELECT 5 UNION ALL SELECT 6)J
```

```
9.CROSS JOIN (SELECT i=0 UNION ALL SELECT 1 UNION ALL SELECT
  2)I
10.      CROSS       APPLY(SELECT       i1=i-1,j1=j,i2=i,j2=j-
  1,i3=i,j3=j+1,i4=i+1,j4=j)D
11.      CROSS APPLY(SELECT
12.       k1=CASE WHEN i1<0 THEN 2 ELSE i1 END
13.      ,k2=CASE WHEN j2<0 THEN 6 ELSE j2 END
14.      ,k3=CASE WHEN j3>6 THEN 0 ELSE j3 END
15.      ,k4=CASE WHEN i4>2 THEN 0 ELSE i4 END
16.      )E;
```

Правильный ответ:

State	NN	WW	SS	EE
T00	T20	T06	T10	T01
T01	T21	T00	T11	T02
T02	T22	T01	T12	T03
T03	T23	T02	T13	T04
T04	T24	T03	T14	T05
T05	T25	T04	T15	T06
T06	T26	T05	T16	T00
T10	T00	T16	T20	T11
T11	T01	T10	T21	T12
T12	T02	T11	T22	T13
T13	T03	T12	T23	T14
T14	T04	T13	T24	T15

T15	T05	T14	T25	T16
T16	T06	T15	T26	T10
T20	T10	T26	T00	T21
T21	T11	T20	T01	T22
T22	T12	T21	T02	T23
T23	T13	T22	T03	T24
T24	T14	T23	T04	T25
T25	T15	T24	T05	T26
T26	T16	T25	T06	T20

Заключение

Автор продолжает работать над книгой, добавляя новые материалы и редактируя старые. В частности, добавлены главы, посвященные языку определения данных (**DDL** — Data Definition Language) и новым синтаксическим конструкциям запросов, которые появились в последних версиях стандарта и поддерживаются реализациями.

Планируется написать об основах нормализации таблиц и оптимизации запросов. Автор с благодарностью примет ваши конструктивные замечания и предложения, которые вы можете отправлять по адресу электронной почты s.moiseenko[@]sql-ex[DOT]ru.

Список цитируемых источников

1. Дейт К.Дж. Введение в системы баз данных, 6-е издание. - К.; М.; СПб.: Издательский дом "Вильямс", 2000

2. Ульман Дж.Д., Уидом Дж. Введение в системы баз данных. - М.: Издательство "Лори", 2000

3. Muthusamy Anantha Kumar. SQL Server and Collation, 2004

4. Мартин Грабер. Справочное руководство по SQL. - М.: Издательство "Лори", 1997

5. Kalen Delaney. Inside Microsoft SQL Server 2005: The Storage Engine (Microsoft Press, 2006) ISBN 978-0735621053

6. Codd E.F. The Relational Model for Database Management Version 2. - Reading, Mass.: Addison-Wesley, 1989

7. Джо Селко. Программирование на SQL для профессионалов. - 2-е издание - М.: Издательство "Лори", 2004

8. Itzik Ben-Gan. Inside Microsoft SQL Server 2008: T-SQL Querying: Microsoft Press, 2009