Учебник по oracle sql

Содержание

1. Введение в SQL . . . 4
   1.1. Диалекты SQL . . . 4
   1.2. Особенности SQL . . . 4
   1.3. Зачем изучать SQL . . . 4
2. Выполнение SQL. Облачные сервисы . . . 5
   2.1. LiveSQL . . . 5
   2.2. SQL Fiddle . . . 6
3. Инструменты для работы с БД Oracle . . . 7
   3.1. Средства разработки . . . 8
   3.2. Проектирование БД . . . 8
4. Таблицы . . . 8
   4.1. Создание таблицы . . . 8
   4.2. Значения по умолчанию . . . 9
   4.3. Понятие NULL. Not-null колонки . . . 10
   4.4. Комментарии к таблице, колонкам . . . 11
5. Основные типы данных . . . 11
   5.1. Varchar2 . . . 11
   5.2. Number . . . 12
   5.3. Date . . . 12
   5.4. Boolean . . . 13
6. Написание SQL- кода . . . 13
   6.1. Комментарии . . . 13
7. Сортировка результатов. Order by . . . 14
   7.1. Подготовка тестовых данных . . . 14
   7.2. Сортировка по возрастанию. Asc . . . 15
   7.3. Сортировка по убыванию. Desc . . . 16
   7.4. Порядок сортировки по-умолчанию . . . 16
   7.5. Сортировка по порядковому номеру . . . 17
   7.6. Nulls last. Nulls first . . . 18
8. Оператор WHERE. Операторы сравнения . . . 19
   8.1. Операторы сравнения . . . 19
9. Проверка нескольких условий. AND, OR . . . 21
10. Проверка значения на NULL . . . 24
11. IN, NOT IN . . . 26
   11.1. Вхождение в набор данных. IN . . . 26
   11.2. Отсутствие в наборе данных. NOT IN . . . 27

12. Вхождение в диапазон. BETWEEN. NOT BETWEEN . . . 29
13. Соединения таблиц . . . 30
13.1. Создание тестовых таблиц . . . 30
13.2. Join . . . 32
13.3. Left join . . . 32
13.4. Соединение таблиц без join . . . 34
14. Древовидные структуры данных. Рекурсивные запросы . . . 34
14.1. Реализация древовидных структур в РСУБД . . . 35
14.2. Connect by . . . 35
14.3. Псевдостолбец level . . . 36
14.4. Псевдостолбец CONNECT_BY_ISLEAF . . . 37
14.5. Сортировка в рекурсивных запросах . . . 38
14.6. Нарушение древовидной структуры при выборке . . . 39
15. Операторы для работы с множествами . . . 40
15.1. Объединение запросов . . . 40
15.2. Разница запросов . . . 43
15.3. Пересечение запросов . . . 47
15.4. Следить за порядком колонок . . . 48
15.5. Сортировка . . . 49
15.6. Приоритет выполнения . . . 50
16. Подзапросы в Oracle . . . 51
16.1. Подготовка тестовых данных . . . 51
16.2. Подзапросы в where- части запроса . . . 52
16.3. Подзапросы в select-части . . . 54
16.4. Подзапросы во FROM части . . . 54
16.5. Коррелированные подзапросы . . . 55
16.6. Подзапросы в IN, NOT IN . . . 56
17. Псевдостолбцы в Oracle . . . 57
17.1. Подготовка данных . . . 57
17.2. ROWID . . . 60
17.3. level . . . 60
18. Функции для работы со строками . . . 60
18.1. upper, lower . . . 61
18.2. Конкатенация строк . . . 61
18.3. Поиск подстроки . . . 62
18.4. Подобие строк. Like . . . 64
18.5. INITCAP . . . 67
18.6. Замена подстроки . . . 68
18.7. Trim. Удаление пробелов в конце/начале строки . . . 68
18.8. LPAD, RPAD . . . 69
19. Функции для работы с NULL . . . 69

19.1. Nvl . . . 70
19.2. Nvl2 . . . 71
19.3. Coalesce . . . 71
20. Агрегирующие функции . . . 72
20.1. Подготовка данных . . . 72
20.2. Having . . . 74
21. Работа с датами в Oracle . . . 75
21.1. Date . . . 75
21.2. Приведение строки к дате . . . 75
21.3. Функция SYSDATE . . . 76
21.4. Приведение даты к строке . . . 76
21.5. trunc . . . 76
21.6. add_months . . . 76
21.7. Разница между датами . . . 77
21.8. TIMESTAMP . . . 78
21.9. EXTRACT . . . 78
22. Оператор INSERT . . . 79
22.1. Вставка с указанием колонок . . . 80
22.2. Вставка без указания колонок . . . 80
22.3. INSERT INTO … SELECT . . . 81
23. Оператор UPDATE . . . 82
24. Оператор DELETE . . . 84
24.1. Удаление данных из связанных таблиц . . . 84
25. Команда MERGE . . . 85
25.1. Подготовка данных . . . 85
25.2. Использование MERGE . . . 86
26. Первичные ключи . . . 90
26.1. Добавление первичного ключа в таблицу . . . 91
26.2. Составные первичные ключи . . . 92
27. Внешние ключи . . . 93
27.1. Создание внешних ключей . . . 94
27.2. Составные внешние ключи . . . 96
28. Уникальные ключи . . . 97
28.1. Создание уникальных ключей . . . 97
29. Представления . . . 100
29.1. Создание представлений . . . 100
29.2. Изменение данных представления . . . 103
30. Изменение структуры таблицы . . . 109
30.1. Подготовка данных . . . 109
30.2. Добавление колонки в таблицу . . . 109
30.3. Удаление колонки из таблицы . . . 111

30.4. Переименование колонки . . . 112
30.5. Изменение типа данных колонки . . . 112
30.6. Изменение атрибута NOT NULL в колонке . . . 114
30.7. Переименование таблицы . . . 114

1. Введение в SQL

SQL, или **Structure Query Language**(Структурированный язык запросов) является основным инструментом для взаимодействия с реляционными базами данных.

С помощью sql можно:

- Получать данные из базы данных
- Сохранять данные в базу данных
- Производить манипуляции с объектами базы данных

1.1. Диалекты SQL

Релящионных систем управления базой данных(СУБД) существует достаточно много. И как правило, в каждой СУБД есть свои отличительные особенности в SQL, которые заключаются в наличии или отсутствии в нем определенных функций, различиях синтаксиса самого SQL, а также по функциональным возможностям этого языка.

В данном учебнике мы будем рассматривать СУБД Oracle.

1.2. Особенности SQL

Пара слов о том, что необычного в SQL.

В отличие многих других языков программирования, например таких как Java, Pascal или JavaScript, программирование на которых заключается в том, чтобы описать, как нужно что-то сделать, в SQL описывается, **что** нужно сделать(т.е. какой результат мы хотим получить). SQL - ближайший к данным язык программирования. Он больше всего приближен к "чистым" данным системы. Под "чистыми" данными подразумевается то, что ниже тех абстракций, с которыми работает sql, уже не будет.

1.3. Зачем изучать SQL

Как уже говорилось, sql является основным средством общения с реляционными базами данных.

Когда какая-либо программа хочет получить, сохранить, или изменить данные в БД, то она это делает посредством SQL. Какой-нибудь список классов, с которыми работает объектно-ориентированный язык, должен получить данные, которые будут храниться в этих классах. Это все делается с помощью SQL.

Даже если в программе нигде явно не пишутся SQL-запросы, а используется с виду обычный программный код(например на языке Java), то это вовсе не значит, что в данном случае общение с БД происходит каким-то другим способом. Скорее всего, в программе используется специальная библиотека, которая превратит код на языке Java в соответствующий код на языке SQL и отправит его на выполнение БД. Подобных библиотек существует великое множество почти для всех популярных языков программирования.

2. Выполнение SQL. Облачные сервисы

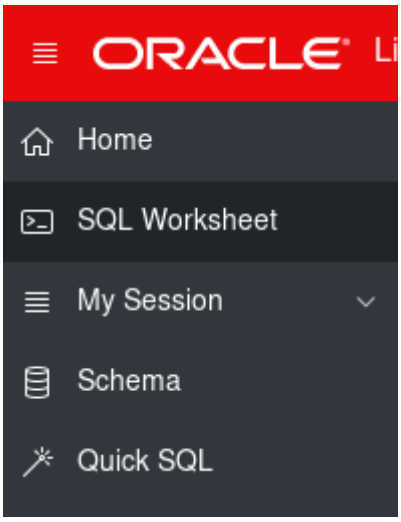
Для того, чтобы начать работу с БД(причем любой), она должна быть где-либо установлена, и к ней должен быть доступ на подключение и выполнение запросов.

2.1. LiveSQL

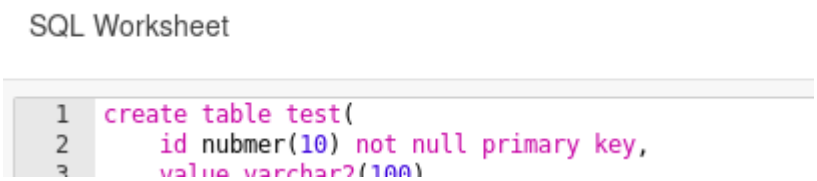
Далее для выполнения sql-запросов будет использоваться сервис Live SQL. Он позволяет выполнять SQL в облаке, что непременно большой плюс - там гораздо быстрее зарегистрироваться, чем скачивать, устанавливать и настраивать себе БД Oracle.

Работать с livesql очень просто; опишем стандартные шаги, необходимые для запуска своих sql-запросов.

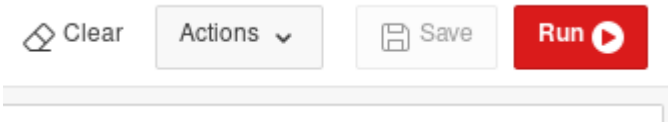
Входим под своей учеткой, после чего в левом боковом меню выбираем `SQL WorkSheet`:



В открывшемся окне вводим наши SQL-запросы:



Чтобы выполнить запрос, написанный в SQL Worksheet, нажимаем нк кнопку `Run`, которая находится сверху над полем для ввода текста запроса:



Впринципе, работа с LiveSQL не должна вызывать вопросов, но на всякий случай вот видео с youtube(на английском) с подробным описанием работы в нем:

▶ https://www.youtube.com/watch?v=4oxsxJQQC-s *(YouTube video)*

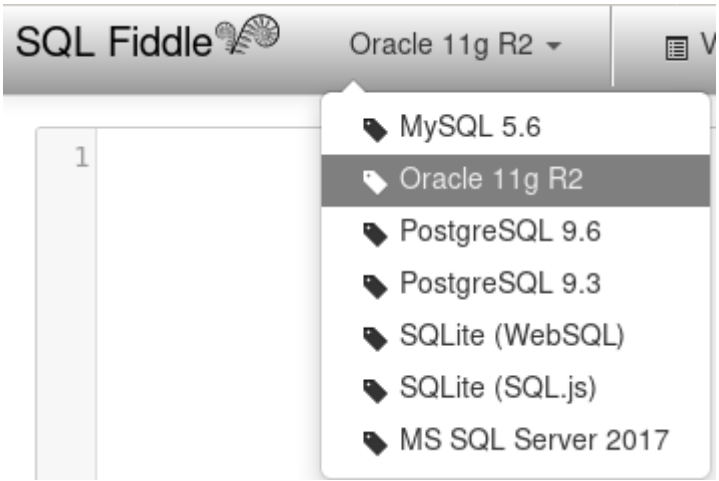
2.2. SQL Fiddle

SQL Fiddle - еще один популярный сервис для работы с SQL. Поддерживает разные базы данных. Для работы SQLFiddle даже не требует регистрации.

Далее будет описано, как работать с данным сервисом.

Сначала заходим на SQL Fiddle.

Т.к. сервис поддерживает работу с несколькими БД, нужно выбрать ту, с которой будем работать - это Oracle:



Перед началом работы SQL Fiddle требует создания схемы. Это значит, что таблицы, с которыми нужно работать, должны быть созданы на этом этапе. Вводим текст ddl-скрипта (скрипта, который создает таблицы и др. объекты БД), после чего нажимаем на кнопку `Build Schema`:

```
Create table test(
  msg varchar2(10) not null
 );
```

После того, как схема будет построена, можно выполнять SQL-запросы. Они вводятся в правой панели(она называется **Query Panel**). Чтобы выполнить запрос, нажимаем на кнопку `Run Sql`:

```
select *
from dual
```

Результаты выполнения запросов отображаются под панелями создания схемы и ввода sql:

DUMMY
X

3. Инструменты для работы с БД Oracle

Далее будут приведены ссылки на полезные инструменты, которые могут пригодиться для работы с Oracle.

3.1. Средства разработки

- PL/SQL Developer - известная среда разработки для Oracle, платная, есть пробный период.
- SQL Developer - бесплатная среда разработки от Oracle.
- Toad for Oracle
- DBForge studio for Oracle
- JetBrains Datagrip - отлично подходит, если необходимо работать одновременно с разными БД. Если рассматривать функционал, доступный с БД Oracle, то немного отстает от всех вышеперечисленных.

3.2. Проектирование БД

- SQL Data Modeler - бесплатный, предоставляется корпорацией Oracle. Обладает обширным функционалом, заточенным именно на работу с БД Oracle.
- ERWin DataModeler - платный. Есть триал период. Хорошо подходит для моделирования структуры данных без привязки к БД.

4. Таблицы

Данные в реляционных базах данных хранятся в таблицах. Таблицы - это ключевой объект, с которыми придется работать в SQL.

Таблицы в БД совсем не отличаются от тех таблиц, с которыми все уже знакомы со школы - они состоят из **колонок** и **строк**.

Каждая колонка в таблице имеет своё **имя** и свой **тип**, т.е. тип данных, которые будут в ней содержаться. Помимо типа данных для колонки можно указать максимальный размер данных, которые могут содержаться в этой таблице.

Например, мы можем указать, что для колонки возраст тип данных - это целое число, и это число должно состоять **максимум** из 3-х цифр. Т.о. максимальное число, которое может содержаться в этой колонке = 999. А с помощью дополнительных конструкций можно задать и правила проверки корректности для значения в колонке,- например, мы можем указать, что для колонки возраст в таблице минимальное значение = 18.

4.1. Создание таблицы

```
create table hello(
    text_to_hello varchar2(100)
);
```

После выполнения данной sql-команды в базе данных будет создана таблица под названием `hello`. Эта таблица будет содержать всего одну колонку под названием `text_to_hello`. В этой колонке мы можем хранить только строковые значения(т.е. любой текст, который можно

ввести с клавиатуры) длинной до 100 байт.

Обратите внимание на размер допустимого текста в колонке `text_to_hello`. 100 байт - это не одно и то же, что и 100 символов! Для того, чтобы сказать базе данных Oracle, что длина строки может быть 100 **символов**, нужно было определить столбец следующим образом:

```
text_to_hello varchar2(100 char)
```

4.1.1. Создание таблицы с несколькими полями

В таблице может много столбцов. Напимер, можно создать таблицу с тремя, пятью или даже 100 колонками. В версиях oracle с 8i по 11g максимальное количество колонок в одной таблице достигает 1000.

Для того, чтобы создать таблицу с несколькими колонками, нужно перечислить все колонки через запятую.

Например, создадим таблицу `cars`, в которой будем хранить марку автомобиля и страну-производитель:

```
create table cars(
    model varchar2(50 char),
    country varchar2(70 char)
);
```

Эта таблица может содержать, например, такие данные:

model	country
toyota	japan
ВАЗ	Россия
Tesla	USA

Следует обратить внимание на последние 2 строки в таблице `cars` - они не полные. Первая из них содержит данные только в колонке `model`, вторая - не содержит данных ни в одной из колонок. Эта таблица может даже состоять из миллиона строк, подобных последней - и каждая строка не будет содержать в себе абслютно никаких данных.

4.2. Значения по умолчанию

При создании таблицы можно указать, какое значение будет принимать колонка по умолчанию:

```
create table cars(
    model varchar2(50 char),
    country varchar2(50 char),
    wheel_count number(2) default 4
)
```

В этом примере создается таблица cars, в которой помимо модели и страны-производителя хранится еще и количество колес, которое имеет автомобиль. И поле `wheel_count` по-умолчанию будет принимать значение, равное 4.

Что значит по-умолчанию? Это значит, что если при вставке данных в эту таблицу не указать значение для колонки `wheel_count`, то оно будет равно числу 4.

4.3. Понятие NULL. Not-null колонки

Ячейки в таблицах могут быть пустыми, т.е. не содержать значения. Для обозначения отсутствия значения в ячейке используется ключевое слово `NULL`. Null могут содержать ячейки с любым типом данных.

Рассмотрим таблицу cars из предыдущего примера. В каждой из трех ее колонок может храниться Null(даже в колонке `wheel_count`, если указать значение Null явно при вставке).

Но представляют ли информационную ценность строки в таблице, где абслютно нет значений? Конечно нет. Если рассматривать таблицу `cars` как источник информации об автомобилях, то нам хотелось бы получать хоть какую-то полезную информацию. Наиболее важной здесь будет колонка model - без нее информация о стране-производителе и количестве колес будет бесполезной.

Для того, чтобы запретить Null-значения в колонке при создании таблицы, используется конструкция `not null`:

```
create table cars(
    model varchar2(50 char) not null,
    country varchar2(50 char),
    wheel_count number(2) default 4
)
```

Теперь БД **гарантирует**, что колонка `model` не будет пустой, по крайней мере до тех пор, пока флаг `not null` включен для этой колонки.

Также можно указать, что колонка `wheel_count` тоже не должна содержать `Null`:

```
create table cars(
    model varchar2(50 char) not null,
    country varchar2(50 char),
    wheel_count number(2) default 4 not null
);
```

4.4. Комментарии к таблице, колонкам

Для создаваемых таблиц и их колонок можно указывать комментарии. Это значительно облегчит понимание того, для чего и как они используются.

Например, укажем комментарии для таблицы `cars` и ее колонок:

```
comment on table cars is 'Список автомобилей';

comment on column cars.model is 'Модель авто, согласно тех. паспорту';
comment on column cars.country is 'Страна-производитель';
comment on column cars.wheel_count is 'Количество колес';
```

Для того, чтобы удалить комментарий, нужно просто задать в качестве его значения пустую строку:

```
comment on table cars is '';
```

5. Основные типы данных

Таблицы могут содержать не только строки. Рассмотрим основные типы данных в БД Oracle.

5.1. Varchar2

Строковый тип. При создании таблицы всегда нужно указывать размер строки.

Размер может указываться в байтах либо в символах.

По-умолчанию максимальный размер строки равен 4000 байт, либо 4000 символов.

Этот размер может быть изменен дополнительной настройкой БД.

```
country(100);-- строка из 100 байт
country(100 char); -- строка из 100 символов
```

5.2. Number

Числовой тип данных. Используется для хранения как целых чисел, так и дробных чисел.

Тип Number может хранить положительные или отрицательные числа, размер которых ограничен 38 цифрами.

Размер числового типа можно ограничивать:

```
age number(3); -- максимальное число = 999; минимальное= -999
price number(5,3);  -- максимальное число = 99.999; минимальное= -99,999
must_print number(1); -- максимальное число = 9; минимальное= -9
rounded_price number(5, -2); -- Число, округленное до 2-символа влево, начиная от
разделителя дроби
```

Рассмотрим последний пример: `rounded_price number(5, -2)`.

Значение -2 здесь означает, что любое дробное число, которое будет записываться в эту колонку, будет округлено, включая две предшествующих резделителю дроби цифры. Ниже показаны примеры входных чисел и числа, в которые они будут преобразованы при сохранении их в колонке `rounded_price`:

Входное число	Выходное число
3245.3	3200
12.345	0

5.3. Date

Тип Date предназначен для хранения даты и времени. Данный тип данных хранить в себе следующую информацию:

- Год
- Месяц
- День(Число)
- Часы
- Минуты
- Секунды

Не всегда бизнес-логика приложения требует хранения даты вплоть до секунды или до дня - иногда нас может интересовать лишь конкретный месяц в году, или только год. В таких случаях незначимая информация как правило устанавливается в некое начальное значение, например:

Что нам нужно хранить	Что мы сохраняем в БД

Дату вплоть до числа	20.05.2019
Определенный месяц в году	01.10.2019
Время, до минут	01.01.1900 20:48

Во втором случае нас интересует только месяц и год, но мы не можем игнорировать число, поэтому *мы сами* решили использовать в качестве дня первый день месяца. Здесь могло быть и 3, и 30, и 20 число месяца. Просто при работе с такими колонками следует знать, для чего они используются и использовать только ту часть даты, которая должна использоваться согласно бизнес-логике.

В третьем случае дата как таковая нас вообще не интересует - нам важно знать только время, поэтому год, месяц и число можно выбрать любые. Конечно, чтобы использовать только время, нужно будет производить определенные манипуляции со значением такого столбца(например отделение значения часов и/или минут, приведение числа, месяца и года к определенным значениям, и т.п.). Также, если известно, что придется работать только определенной частью даты, можно использовать тип number. Как пример - колонка `release_year`, которая хранит в себе год выпуска определенной модели авто. Здесь месяц, число и время скорее всего не понадобятся совсем.

5.4. Boolean

Логического типа данных в БД Oracle нет. Но вместо него можно использовать уже знакомые типы `number` или `varchar2`:

```
create table questions(
    is_right_n number(1), ①
    is_right_c varchar2(1) ②
);
```

① значение, равное 1 трактуем как истину, иначе - ложь(или наоборот)

② значение, равное символу 'Y' - истинно, 'N' - ложно

6. Написание SQL- кода

Код SQL, как и любой другой, можно сохранять в файлы. Расширение этих файлов на самом деле не имеет значения, но принято сохранять sql-скрипты с расширением .sql.

Некоторые IDE могут сохранять SQL-код и с другими расширениями файлов(например PL/SQL Developer - для т.н. тестовых скриптов он использует расширение *.tst.

6.1. Комментарии

В SQL можно и нужно добавлять комментарии. Это участки текста, предназначенные для чтения другими людьми, и которые не обрабатываются базой данных.

Комментарий может быть однострочным:

```
-- Получить базовую информацию о записях в блоге
select a.title,
       a.name,
       a.create_date
from posts a
```

Многострочные комментарии также поддерживаются:

```
/* Отобразить записи в блоге
   для пользователя johndoe
   в порядке их публикации
*/
select a.title,
       a.post
from posts a
where a.username = 'JOHNDOE'
order by a.publish_date
```

Многострочные комментарии начинаются с символов `/` и заканчиваются символами `/`. Вообще говоря, такой коментарий может быть и однострочным:

```
/* Однострочный комментарий */
select a.title
from posts a
```

7. Сортировка результатов. Order by

При выборке данных из БД мы можем сортировать извлекаемые данные в нужном нам порядке. Использование сортировки поможет сделать получаемые данные более удобочитаемыми и воспринимаемыми для анализа человеком.

7.1. Подготовка тестовых данных

Создадим таблицу, которая будет содержать список блюд ресторана:

```
create table dishes(
    name varchar2(100) not null,
    price number(5,2) not null,
    rating number(5)
);

comment on column dishes.name is 'Наименование блюда';
comment on column dishes.price is 'Стоимость за одну порцию';
comment on column dishes.rating is 'Популярность блюда';

insert into dishes(name, price, rating) values ('Макароны с сыром', 20.56, 320);
insert into dishes(name, price, rating) values ('Борщ', 10, 130);
insert into dishes(name, price, rating) values ('Чай с лимоном', 1.34, 270);
insert into dishes(name, price, rating) values ('Чай с молоком', 1.20, 280);
insert into dishes(name, price, rating) values ('Свиная отбивная', 30.50, 320);
insert into dishes(name, price, rating) values ('Овощной салат', 5.70, null);
```

Овощной салат - новинка в меню, и его еще не успели оценить посетители; Именно поэтому в колонке `rating` содержится `null`. Конечно, здесь возможны варианты - например, можно было хранить значение 0 для обозначения отсутствия оценок блюда посетителями, но для демонстрационных целей мы здесь будем хранить `null`.

7.2. Сортировка по возрастанию. Asc

Для того, чтобы получить данные в определенном порядке, используется конструкция `order by`. Для того, чтобы сортировка выполнялась по возрастанию, к конструкции `order by` добавляется атрибут `asc`.

Получим все блюда в из меню и отсортируем их по стоимости начиная с дешевых и заканчивая самыми дорогими:

```
select *
from dishes
order by price asc
```

NAME	PRICE	RATING
Чай с молоком	1.2	280
Чай с лимоном	1.34	270
Овощной салат	5.7	-
Борщ	10	130
Макароны с сыром	20.56	320
Свиная отбивная	30.5	320

Сортировка может выполняться одновременно по нескольким полям. Давайте добавим сортировку по рейтингу блюд начиная от самых непопулярных и заканчивая самыми

популярными блюдами. При этом, кроме рейтинга мы будем сортировать блюда по стоимости - от дешевых к дорогим:

```
select *
from dishes
order by rating asc, price asc
```

NAME	PRICE	RATING
Борщ	10	130
Чай с лимоном	1.34	270
Чай с молоком	1.2	280
Макароны с сыром	20.56	320
Свиная отбивная	30.5	320
Овощной салат	5.7	-

Как видно, блюда, которые имеют одинаковый рейтинг, расположились в порядке возрастания их цен.

7.3. Сортировка по убыванию. Desc

Для сортировки по-убыванию используется атрибут `desc`.

Например, для получения списка блюд начиная от самых популярных и заканчивая самыми непопулярными, можно написать следующий запрос:

```
select *
from dishes
order by rating desc
```

NAME	PRICE	RATING
Овощной салат	5.7	-
Макароны с сыром	20.56	320
Свиная отбивная	30.5	320
Чай с молоком	1.2	280
Чай с лимоном	1.34	270
Борщ	10	130

7.4. Порядок сортировки по-умолчанию

Если в конструкции order by не указывать порядок сортировки, то oracle будет производить сортировку по возрастанию.

Т.е. следующий запрос:

```
select *
from dishes
order by price asc, rating asc
```

Аналогичен следующему:

```
select *
from dishes
order by price, rating
```

7.5. Сортировка по порядковому номеру

Вместо указания колонки, по которой должна производиться сортировка, можно указать ее порядковый номер в выборке. Следующие 2 запроса идентичны:

```
select price, rating
from dishes
order by price, rating
select price, rating
from dishes
order by 1, 2
```

Однако, такого подхода следует избегать, и вот почему.

Предположим, мы написали следующий запрос:

```
select name, price
from dishes
order by 2 desc
```

Этот запрос выводит список блюд начиная от самых дорогих и заканчивая самыми дешевыми:

NAME	PRICE
Свиная отбивная	30.5
Макароны с сыром	20.56
Борщ	10
Овощной салат	5.7
Чай с лимоном	1.34
Чай с молоком	1.2

Проходит несколько месяцев, и мы решаем извлекать кроме наименования блюда и его цены еще и рейтинг. Пишется следующий запрос:

```
select name, rating, price
from dishes
order by 2 desc
```

NAME	RATING	PRICE
Овощной салат	-	5.7
Макароны с сыром	320	20.56
Свиная отбивная	320	30.5
Чай с молоком	280	1.2
Чай с лимоном	270	1.34
Борщ	130	10

Но эти данные идут не в том порядке, который нам нужен! Они отсортированы по рейтингу, а не по стоимости. Это произошло потому, что колонка с ценой теперь третья по счету, а не вторая, и при добавлении в выборку еще одной колонки нужно было проверить `order by` - блок и изменить порядковый номер для сортировки.

7.6. Nulls last. Nulls first

Сортировка производится по определенным значениям. Но что делать, если значение в колонке отсутствует, т.е. в нем содержится `null`?

Здравый смысл подсказывает, что сортировка по `null`-значениям невозможна.

Но мы можем указать, где должны располагаться `null` -значения при сортировке в начале или конце. Достигается это путем использования конструкций `nulls last` и `nulls first`. Использование первой разместит все `null` - значения в конце, а второй - в начале.

```
select *
from dishes
order by rating nulls first
```

NAME	PRICE	RATING
Овощной салат	5.7	-
Борщ	10	130
Чай с лимоном	1.34	270
Чай с молоком	1.2	280
Свиная отбивная	30.5	320
Макароны с сыром	20.56	320

```
select *
from dishes
order by rating nulls last
```

NAME	PRICE	RATING
Борщ	10	130
Чай с лимоном	1.34	270
Чай с молоком	1.2	280
Свиная отбивная	30.5	320
Макароны с сыром	20.56	320
Овощной салат	5.7	-

8. Оператор WHERE. Операторы сравнения

Использование оператора `where` позволяет добавить фильтр на те данные, которые будет обрабатывать sql, будь то выборка, вставка, обновление или удаление.

Подготовка тестовых данных Для демонстрации будем использовать те же данные, что и в примере с `order by`:

```
create table dishes(
name varchar2(100) not null,
price number(5,2) not null,
rating number(5)
);

comment on column dishes.name is 'Наименование блюда';
comment on column dishes.price is 'Стоимость за одну порцию';
comment on column dishes.rating is 'Популярность блюда';

insert into dishes(name, price, rating) values ('Макароны с сыром', 20.56, 320);
insert into dishes(name, price, rating) values ('Борщ', 10, 130);
insert into dishes(name, price, rating) values ('Чай с лимоном', 1.34, 270);
insert into dishes(name, price, rating) values ('Чай с молоком', 1.20, 280);
insert into dishes(name, price, rating) values ('Свиная отбивная', 30.50, 320);
insert into dishes(name, price, rating) values ('Овощной салат', 5.70, null);
```

8.1. Операторы сравнения

В `where` можно использовать следующие реляционные операторы:

Оператор	Что обозначает

<	меньше
>	больше
≤	меньше либо равно
≥	больше либо равно
!=	не равно

Рассмотрим применение данных операторов на примерах.

8.1.1. Оператор "Меньше"(<)

```
select d.*
from dishes d
where d.rating < 320
```

Данный запрос вернет список всех блюд, рейтинг которых меньше, чем 320:

NAME	PRICE	RATING
Борщ	10	130
Чай с лимоном	1.34	270
Чай с молоком	1.2	280

8.1.2. Оператор "Больше"(>)

```
select d.*
from dishes d
where d.rating > 270
```

Данный запрос вернет список блюд с рейтингом, большим, чем 270:

NAME	PRICE	RATING
Макароны с сыром	20.56	320
Чай с молоком	1.2	280
Свиная отбивная	30.5	320

8.1.3. Оператор "Больше либо равно"(≥)

```
select d.*
from dishes d
where d.rating > 270
```

Данный запрос вернет список блюд с рейтингом, большим либо равным 270:

NAME	PRICE	RATING
Макароны с сыром	20.56	320
Чай с лимоном	1.34	270
Чай с молоком	1.2	280
Свиная отбивная	30.5	320

8.1.4. Оператор "Меньше либо равно"(≤)

```
select d.*
from dishes d
where d.rating  320
```

Данный запрос возвращает все блюда, рейтинг которых меньше либо равен 320:

NAME	PRICE	RATING
Макароны с сыром	20.56	320
Борщ	10	130
Чай с лимоном	1.34	270
Чай с молоком	1.2	280
Свиная отбивная	30.5	320

9. Проверка нескольких условий. AND, OR

При выборке данных мы можем указывать несколько условий одновременно. Для объединения условий можно использовать операторы `and`(логическое *И*) и `or`(логическое *ИЛИ*). Разберем каждый из них на примерах.

Пример 1: получим список блюд с рейтингом, меньшим чем 320, но со стоимостью большей, чем 2:

```
select *
from dishes
where rating < 320
and price > 2
```

NAME	PRICE	RATING
Борщ	10	130

Пример 2: получим список блюд, рейтинг которых варьируется от 280 до 320 включительно, и цена которых меньше 30:

```
select d.*
from dishes d
where rating □ 280
and rating □ 320
and price < 30
```

NAME	PRICE	RATING
Макароны с сыром	20.56	320
Чай с молоком	1.2	280

При использовании ключевого слова AND следует всегда помнить о том, что если хотя бы одно из условий будет ложным, то результат всего выражения также будет ложным:

```
select d.*
from dishes d
where rating □ 280
and rating □ 320
and price < 30
and 1 = 0
```

```
no data found
```

В приведенном выше примере выражение 1 = 0 является ложным, а значит и всё условие также становится ложным, что приводит к отсутствию данных в выборке.

При использовании ключевого слова AND следует всегда помнить о том, что если хотя бы одно из условий будет ложным, то результат всего выражения также будет ложным.

Пример □3: Получить список блюд, рейтинг которых либо 320, либо 280:

```
select d.*
from dishes d
where rating =280
or rating = 320
```

NAME	PRICE	RATING
Макароны с сыром	20.56	320
Чай с молоком	1.2	280
Свиная отбивная	30.5	320

При использовании ключевого слова OR все выражение будет считаться истинным, если **хотя бы одно** условие из перечисленных будет истинным.

Например, следующий запрос вернет все строки из таблицы `dishes`:

```
select d.*
from dishes d
where rating = -1
or 1 < 2
```

NAME	PRICE	RATING
Макароны с сыром	20.56	320
Борщ	10	130
Чай с лимоном	1.34	270
Чай с молоком	1.2	280
Свиная отбивная	30.5	320
Овощной салат	5.7	-

Несмотря на то, что у нас нет ни одного блюда с рейтингом, равным -1, запрос вернул все строки, так как для каждой строки будет истинынм условие `1 < 2`.

Используя комбинирование `AND` и `OR`, можно составлять более сложные условия.

Пример 4: Получить список блюд, рейтинг которых равен 320 и стоимость больше 30, *либо* рейтинг которых меньше, чем 270:

```
select d.*
from dishes d
where (d.rating = 320 and d.price > 30)
or d.rating < 270
```

NAME	PRICE	RATING
Борщ	10	130
Свиная отбивная	30.5	320

В приведенном выше примере использовались скобки для группировки условий. Скобки используются для того, чтобы определить приоритет вычисления выражения. Здесь все, как если бы мы считали обычное математическое выражение - то, что в скобках, вычисляется отдельно.

Свиная отбивная попала в выборку потому, что она удовлетворяет условию `d.rating = 320 and d.price > 30` (Ее стоимость 30.5, а рейтинг = 320). Борщ попал в выборку потому, что он удовлетворяет условию `d.rating < 270` (его рейтинг равен 130). Так как между двумя условиями(выражение в скобках рассматриваем как одно условие) стоит `OR`, то в выборку попадает любая строка, которая удовлетворяет хотя бы одному из этих условий.

Для того, чтобы лучше понять, какую роль здесь выполняют скобки, выполним тот же запрос, но только расставим в нем скобки по-другому:

```
select d.*
from dishes d
where d.rating = 320 and
(d.price > 30 or d.rating < 270)
```

NAME	PRICE	RATING
Свиная отбивная	30.5	320

Сейчас в выборку попала только свиная отбивная. Давайте разберемся, почему. Учитывая скобки, каждая строка в выборке должна удовлетворять следующим условиям:

- Рейтинг должен быть равен 320
- Стоимость должна быть больше 30, *либо* рейтинг должен быть меньше 270

При этом, эти два условия должны быть истинными одновременно, т.к. между ними указано ключевое слово `AND`.

Итак, блюд с рейтингом, равным 320, всего два - "Макароны с сыром" и "Свиная отбивная". Т.е. по первому условию в выборку попадают всего 2 блюда. Теперь по второму условию. У Макарон с сыром стоимость = 20.56, рейтинг = 320. Посмотрим, будет ли истинным второе условие для них; для этого просто мысленно подставим значения в условие: `20.56 > 30` или `320 < 270`. Для того, чтобы данное условие было истинным, достаточно, чтобы хотя бы одно из его частей было истинным, т.к. используется `OR`. Но, как видно, ни одно из них не является истинным. Это значит, что все выражение в скобках является ложным, а значит и данная строка не попадет в выборку.

10. Проверка значения на NULL

Если обратить внимание на результаты запросов, выше, то можно заметить, что строка, содержащая `NULL` в колонке `rating` не была возвращена ни одним из них.

Как уже говорилось ранее, `NULL` - это отсутствие значения. Соответственно, он и не может быть больше, меньше, либо даже равняться какому-либо значению, даже себе.

Например, следующий запрос не вернет ни одной строки, хотя мы вроде как и указываем в запросе необходимый критерий - равенство `NULL`:

```
select d.*
from dishes d
where d.rating = NULL
```

```
no data found
```

Теперь попробуем получить все блюда, у которых рейтинг укзан, т.е. те строки из таблицы, где значение rating не равно `NULL`:

```
select d.*
from dishes d
where d.rating <> NULL
```

Получим аналогичный результат - ни одной строки не будет получено:

```
no data found
```

Встает вопрос - как определить, что колонка содержит NULL?

Для этого используются операторы `IS NULL` и `IS NOT NULL`.

`IS NULL` проверяет, является ли значение равным NULL, в то время как `IS NOT NULL` проверяет, является ли значение любым, но не NULL.

Работу данных операторов лучше рассмотреть на примерах.

Получим блюда, которые еще не получили никакой оценки, т.е. те блюда, которые в колонке `rating` содержат NULL:

```
select d.*
from dishes d
where d.rating IS NULL
```

NAME	PRICE	RATING
Овощной салат	5.7	-

А теперь получим все блюда, которые уже получили оценку:

```
select d.*
from dishes d
where d.rating IS NOT NULL
```

Как видно, `IS NOT NULL` возвращает все строки, кроме тех, которые содержат `NULL`:

NAME	PRICE	RATING
Макароны с сыром	20.56	320
Борщ	10	130
Чай с лимоном	1.34	270
Чай с молоком	1.2	280
Свиная отбивная	30.5	320

В Oracle пустая строка эквивалентна `NULL`

11. IN, NOT IN

11.1. Вхождение в набор данных. IN

Условие IN позволяет ответить на следующий вопрос: "Входит (IN) ли значение в заданный набор данных?".

Следующий пример вернет все блюда, рейтинг которых равен 320 либо 270:

```
select d.*
from dishes d
where d.rating IN (320, 270)
```

NAME	PRICE	RATING
Макароны с сыром	20.56	320
Чай с лимоном	1.34	270
Свиная отбивная	30.5	320

Использовать можно любые типы, не только числа:

```
select d.*
from dishes d
where d.name IN ('Макароны с сыром', 'Овощной салат', 'Борщ')
```

NAME	PRICE	RATING
Макароны с сыром	20.56	320
Борщ	10	130
Овощной салат	5.7	-

Следуеть помнить, что при сравнении строк учитывается регистр, т.е. следующий запрос:

```
select d.*
from dishes d
where d.name IN ('Макароны с сыром', 'ОВОЩНОЙ салат', 'БОРЩ')
```

Не вернет строки с овощным салатом и борщом:

NAME	PRICE	RATING
Макароны с сыром	20.56	320

Можно попробовать поправить ситуацию и воспользоваться уже знакомой функцией UPPER. Напомним, что эта функция приводит строку к верхнему регистру:

```
select d.*
from dishes d
where UPPER(d.name) IN ('Макароны с сыром', 'ОВОЩНОЙ салат', 'БОРЩ')
```

NAME	PRICE	RATING
Борщ	10	130

Итак, следующим запросом мы фактически сказали БД: "Покажи нам все строки из таблицы dishes, в которых наименование, написанное большими буквами, будет равно либо **'Макароны с сыром'**, либо **'ОВОЩНОЙ салат'**, либо **'БОРЩ'**".

Почему в выборку не попали макароны с сыром и овощной салат? Ответ прост - строка >"МАКАРОНЫ С СЫРОМ" не идентична строке "Макароны с сыром", как и строка **"ОВОЩНОЙ САЛАТ"** не идентична строке **"ОВОЩНОЙ салат"**.

Как же можно получить все три интересующих нас блюда, не переживая за то, что регистры строк(а здесь достаточно несовпадения и хотя бы в одном символе) в таблице dishes не совпадут с регистрами строк, которые мы перечисляем в выражении `IN`?

Ответ прост - привести к верхнему/нижнему регистру как строки в таблице, так и строки в выражении `IN`.

Следующий запрос выдаст список всех интересующих нас блюд:

```
select d.*
from dishes d
where UPPER(d.name) IN (upper('Макароны с сыром'), upper('ОВОЩНОЙ салат'), upper('
БОРЩ'))
```

NAME	PRICE	RATING
Макароны с сыром	20.56	320
Борщ	10	130
Овощной салат	5.7	-

11.2. Отсутствие в наборе данных. NOT IN

Условие `NOT IN` выполняет функцию, противоположную выражению `IN`: убедиться, что значение не входит в указанный набор данных.

Напимер, нам требуется получить список блюд, за исключением чая с молоком и овощного салата:

```
select *
from dishes
where name not in ('Овощной салат', 'Чай с молоком')
```

Получим следующий результат:

NAME	PRICE	RATING
Макароны с сыром	20.56	320
Борщ	10	130
Чай с лимоном	1.34	270
Свиная отбивная	30.5	320

Для понимания того, как работает конструкция NOT IN, лучше рассматривать приведенный пример как следующий, *эквивалентный* запрос:

```
select *
from dishes
where name <> 'Овощной салат'
and name <> 'Чай с молоком'
```

При использовании `NOT IN`, проверяемое значение будет поочередно сравнено с каждым из значений, перечисленных в скобках после `NOT IN`, и если хотя бы одно сравнение не будет истинным, то все условие будет считаться ложным.

Если в списке значений `NOT IN` будет присутствовать хотя бы одно `NULL`-значение, то условие будет ложным для всех обрабатываемых строк, даже тех, где проверяемое значение является `NULL`.

Для большего понимания рассмотрим это на примере.

Предположим, мы хотим получить список блюд, рейтинг которых не 320 и не NULL. Для этого мы написали следующий запрос:

```
select *
from dishes
where rating not in (320, null)
```

```
no data found
```

Результат получился немного не таким, как хотелось бы. Для того, чтобы понять, почему не было получено никаких данных, следует понимать, как рассматривается данный запрос:

```
select *
from dishes
where rating <> 320
and rating <> null
```

Теперь все должно быть более понятным. Причина кроется в выражении `rating <> null`. Как уже было рассмотрено, сравнение с `NULL` всегда дает ложный результат, а так как используется логическое И(and), то и результат всего выражения WHERE будет ложным.

Поэтому, используя `NOT IN`, всегда следует убедиться в отсутствии null-значений.

12. Вхождение в диапазон. BETWEEN. NOT BETWEEN

`BETWEEN` используется для того, чтобы проверить значение на вхождение в диапазон. Проверять вхождение в диапазон значений можно *строки, числа и даты.*

Пример 1: Получить список блюд, рейтинг которых колеблется от 270 до 320 включительно:

```
select d.*
from dishes d
where rating between 270 and 320
```

NAME	PRICE	RATING
Макароны с сыром	20.56	320
Чай с лимоном	1.34	270
Чай с молоком	1.2	280
Свиная отбивная	30.5	320

Следует помнить, что граничные значения диапазона всегда включаются при проверке, т.е. этот запрос идентичен следующему:

```
select d.*
from dishes d
where d.rating  270
and d.rating  320
```

Пример 2: Получить список блюд, рейтинг которых колеблется от 270 до 320, и стоимость которых от 1 до 6:

```
select d.*
from dishes d
where d.rating between 270 and 320
and d.price between 1 and 6
```

NAME	PRICE	RATING
Чай с лимоном	1.34	270
Чай с молоком	1.2	280

Пример 3: Получить список блюд с рейтингом, значения которого не входят в диапазон чисел от 270 до 320:

```
select d.*
from dishes d
where d.rating not between 270 and 320
```

NAME	PRICE	RATING
Борщ	10	130

Здесь для того, чтобы исключить значения из диапазона, перед `between` было добавлено ключевое слово `NOT`.

13. Соединения таблиц

Работать с одной таблицей в БД приходится редко. Как правило, данные распределены по нескольким таблицам, которые связаны между собой.

13.1. Создание тестовых таблиц

Для демонстрации соединений понадобится несколько таблиц.

```
create table app_users(
    login varchar2(50 char) primary key,
    registration_date date default sysdate not null,
    email varchar2(200 char) not null
);

comment on table app_users is 'Пользователи';

create table app_roles(
    role_id number(10) primary key,
    role_name varchar2(50) not null
);

comment on table app_roles is 'Роли в системе';

create table user_roles(
    login varchar2(50 char) not null,
    role_id number(10) not null,
    constraint user_roles_login_fk foreign key(login)
    references app_users(login),
    constraint user_roles_role_id_fk foreign key(role_id)
    references app_roles(role_id)
);

insert into app_users values('johndoe', sysdate, 'johndoe@johndoemail.com');
insert into app_users values('alex', sysdate, 'alexman@mail.com');
insert into app_users values('kate', sysdate, 'kate@somemaill.com');
insert into app_users values('mike', sysdate, 'mike@mikemailll.com');
insert into app_users values('dmitry', sysdate, 'dmitry@somemaill.com');
insert into app_users values('mr_dude', sysdate, 'mr_dude@email.dude');

insert into app_roles values(1, 'admin');
insert into app_roles values(2, 'boss');
insert into app_roles values(3, 'employee');
insert into app_roles values(4, 'support');

insert into user_roles values('johndoe', 1);
insert into user_roles values('johndoe', 2);
insert into user_roles values('johndoe', 3);
insert into user_roles values('alex', 3);
insert into user_roles values('kate', 3);
insert into user_roles values('mike', 2);
insert into user_roles values('dmitry', 3);
```

Как видно, информация о пользователях хранится в нескольких таблицах. Для того, чтобы получить данные "вместе", придется использовать соединения.

13.2. Join

Получим список пользователей вместе с ролями, которыми они обладают в системе:

```
select au.login, au.email, ar.role_name
from app_users au
JOIN user_roles ur on au.login = ur.login
JOIN app_roles ar on ar.role_id = ur.role_id
```

Получим следующий результат:

LOGIN	EMAIL	ROLE_NAME
johndoe	johndoe@johndoemail.com	admin
johndoe	johndoe@johndoemail.com	boss
johndoe	johndoe@johndoemail.com	employee
alex	alexman@mail.com	employee
kate	kate@somemaill.com	employee
mike	mike@mikemailll.com	boss
dmitry	dmitry@somemaill.com	employee

Приведенный запрос можно читать по-порядку:

1. Берем все записи из таблицы `user_roles`
2. Теперь "приклеиваем" справа к нашему набору данных строки из таблицы `app_roles`, у которых в колонке `role_id` содержатся такие же значения, как и в колонке `role_id` таблицы `user_roles`. При этом строки, у которых эти значения не совпадают, убираются из результирующего набора
3. К получившемуся на шаге 2 набору данных "приклеиваем" справа строки из таблицы app_users, у которых значение в колонке login совпадает со значением колонки `login` в таблице `user_roles`. Опять же, строки, у которых эти значение не совпадают, удаляются из результирующего набора данных.
4. Из получившегося набора данных, выбираем только колонки `login`, `email`, `role_name`. После "склейки" данных наш набор содержит все колонки, которые содержатся в используемых таблицах, так что мы могли показать значения вообще любых колонок из любой из этих трех таблиц(либо вообще все).

13.3. Left join

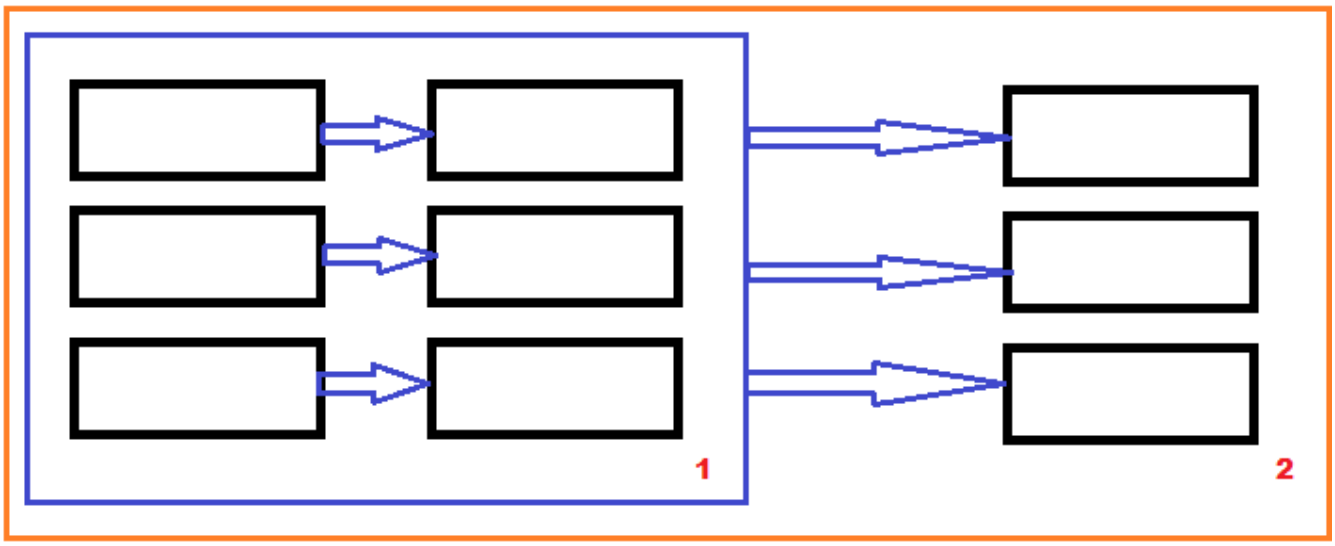
Предыдущий запрос выводил только тех пользователей, у которых действительно были назначены некие роли в приложении. Теперь покажем всех пользователей и их роли. Для этого будет использоваться `LEFT JOIN`. Он отличается от обычного `JOIN` тем, что он не убирает строки из уже имеющегося набора данных когда "приклеивает" справа новые данные.

```
select au.login, au.email, ar.role_name
from app_users au
LEFT JOIN user_roles ur on au.login = ur.login
LEFT JOIN app_roles ar on ar.role_id = ur.role_id
```

LOGIN	EMAIL	ROLE_NAME
johndoe	johndoe@johndoemail.com	admin
johndoe	johndoe@johndoemail.com	boss
mike	mike@mikemailll.com	boss
johndoe	johndoe@johndoemail.com	employee
alex	alexman@mail.com	employee
kate	kate@somemaill.com	employee
dmitry	dmitry@somemaill.com	employee
mr_dude	mr_dude@email.dude	-

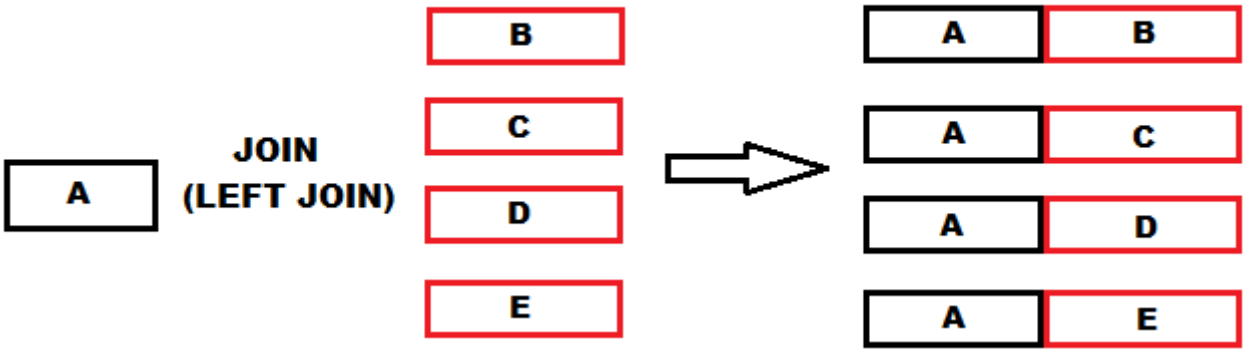
Как видно, теперь к результирующей выборке добавился пользователь `mr_dude`, которому не были назначены права.

Схематично процесс "приклеивания" показан на рисунке:



Исходная таблица и первый `JOIN`(или `LEFT JOIN`) дают некий набор данных, который обозначен цифрой **1**. Все, далее стоит этот набор данных рассматривать как одну таблицу, к которой еще раз "приклеиваются" данные с помощью еще одного соединения.

Еще одна схема соединений:



Она показывает, что если одной записи в левой части нашего "текущего" набора данных соответствует несколько строк в "добавляемой" таблице, то количество строк после соединения увеличится - для одна строка из левой части набора данных будет соединена **с каждой** строкой из правой части данных.

13.4. Соединение таблиц без join

Пример из части, где описывалось соединение `join`, может быть записан и без использования этого самого `join`.

```
select au.login, au.email, ar.role_name
from app_users au
JOIN user_roles ur on au.login = ur.login
JOIN app_roles ar on ar.role_id = ur.role_id
```

```
select au.login, au.email, ar.role_name
from app_users au,
user_roles ur,
app_roles ar
where au.login = ur.login
and   ar.role_id = ur.role_id
```

Эти два запроса идентичны.

Вообще, Oracle позволяет записать и left/right join - соединения подобным образом, указывая правила соединения в части `where` запроса. Данный синтаксис использовался до версии БД = 9i и здесь рассматриваться не будет.

14. Древовидные структуры данных. Рекурсивные запросы

Достаточно часто приходится иметь дело с древовидными структурами данных. Классическим примером является структура подразделений организации, где один отдел является частью другого, и при этом также состоит из нескольких подразделений. Также

можно в виде дерева описать отношения между сотрудниками - кто кому приходится начальником; некий список документов, где один документ появляется на основании другого, а тот в свою очередь был создан на основании третьего, и т.п.

14.1. Реализация древовидных структур в РСУБД

Для того, чтобы можно было листья дерева собрать воедино, нужно знать, как они соотносятся друг с другом. Как правило, все данные, которые нужно хранить в виде дерева, хранятся в одной таблице. Для того, чтобы по определенной строке определить ее родителя, в таблицу добавляется колонка, которая ссылается на родителя в этой же таблице. У корневого узла в дереве колонка с id родительского узла остается пустой.

Для разбора создадим таблицу, которая будет содержать список подразделений.

```
create table departments(
id number primary key,
dept_name varchar2(100),
parent_id number,
constraint departments_parent_id_fk foreign key(parent_id)
references departments(id));

comment on table departments is 'Подразделения';

comment on column departments.parent_id is 'Ссылка на родительский узел';

insert into departments values(1, 'ЗАО ИнвестКорп', null);
insert into departments values(2, 'Бухгалтерия', 1);
insert into departments values(3, 'Отдел продаж', 1);
insert into departments values(4, 'IT-отдел', 1);
insert into departments values(5, 'Дирекция', 1);
insert into departments values(6, 'Бухгалтерия по участку 1', 2);
insert into departments values(7, 'Бухгалтерия по участку 2', 2);
insert into departments values(8, 'Отдел QA', 4);
insert into departments values(9, 'Отдел разработки', 4);
```

14.2. Connect by

Oracle имеет свой собственный синтаксис для написания рекурсивных запросов. Сначала пример:

```
select d.*
from departments d
start with d.id = 1
connect by prior id = d.parent_id
```

ID	DEPT_NAME	PARENT_ID
1	ЗАО ИнвестКорп	-
2	Бухгалтерия	1
6	Бухгалтерия по участку 1	2
7	Бухгалтерия по участку 2	2
3	Отдел продаж	1
4	IT-отдел	1
8	Отдел QA	4
9	Отдел разработки	4
5	Дирекция	1

Данный запрос проходит по дереву вниз начиная с узла, имеющего `id = 1`.

`connect by` задает правило, по которому дерево будет обходиться. В данном примере мы указываем, что у строк, которые должны будут выбираться на следующем шаге, значение столбца `parent_id` должно быть таким же, как значение столбца `id` на текущем.

В конструкции `start with` не обязательно указывать некие значения для `id` строк. Там можно указывать любое выражение. Те строки, для которых оно будет истинным, и будут являть собой стартовые узлы в выборке.

14.3. Псевдостолбец level

При использовании рекурсивных запросов, написанных с использованием `connect by`, становится доступен такой псевдостолбец, как `level`. Этот псевдостолбец возвращает 1 для корневых узлов в дереве, 2 для их дочерних узлов и т.д.

```
select dp.*, level
from departments dp
start with dp.parent_id is null
connect by prior id = dp.parent_id
```

ID	DEPT_NAME	PARENT_ID	LEVEL
1	ЗАО ИнвестКорп	-	1
2	Бухгалтерия	1	2
6	Бухгалтерия по участку 1	2	3
7	Бухгалтерия по участку 2	2	3
3	Отдел продаж	1	2
4	IT-отдел	1	2
8	Отдел QA	4	3
9	Отдел разработки	4	3
5	Дирекция	1	2

В приведенном выше примере мы начинаем строить наше дерево с корневых узлов, не зная их конкретных `id`. Но мы знаем, что у корневых узлов нет родителей, что и указали в конструкции `start with` - `parent_id is null`. В этом случае корневые узлы дерева, которое вернет запрос, будут совпадать с корневыми узлами дерева, которое хранится в БД.

Можно, например, используя `level`, вывести дерево в более красивом виде:

```
select lpad(dp.dept_name, length(dp.dept_name) + (level * 4) - 4, ' ') dept_name,
level
from departments dp
start with dp.parent_id is null
connect by prior id = dp.parent_id
```

DEPT_NAME	LEVEL
ЗАО ИнвестКорп	1
Бухгалтерия	2
Бухгалтерия по участку 1	3
Бухгалтерия по участку 2	3
Отдел продаж	2
IT-отдел	2
Отдел QA	3
Отдел разработки	3
Дирекция	2

Здесь используется функция lpad, которая дополняет передаваемую строку(наименование подразделения) до определенной длины(длина наименования + уровень вложенности * 4) пробелами слева. Кстати, функция `rpad` работает так же, только дополняет символы справа.

14.4. Псевдостолбец CONNECT_BY_ISLEAF

Данный псевдостолбец вернет 1 в том случае, когда у узла в дереве больше нет потомков, и 0 в противном случае.

```
select dp.dept_name, CONNECT_BY_ISLEAF
from departments dp
start with dp.parent_id is null
connect by prior id = dp.parent_id
```

DEPT_NAME	CONNECT_BY_ISLEAF
ЗАО ИнвестКорп	0
Бухгалтерия	0
Бухгалтерия по участку 1	1
Бухгалтерия по участку 2	1
Отдел продаж	1
IT-отдел	0
Отдел QA	1
Отдел разработки	1
Дирекция	1

14.5. Сортировка в рекурсивных запросах

В запросах с использованием `CONNECT BY` нельзя использовать `ORDER BY` и `GROUP BY`, т.к. они нарушат древовидную структуру.

Это можно увидеть на примере:

```
select dp.dept_name, level
from departments dp
start with dp.parent_id is null
connect by prior id = dp.parent_id
order by dp.dept_name asc
```

DEPT_NAME	LEVEL
IT-отдел	2
Бухгалтерия	2
Бухгалтерия по участку 1	3
Бухгалтерия по участку 2	3
Дирекция	2
ЗАО ИнвестКорп	1
Отдел QA	3
Отдел продаж	2
Отдел разработки	3

Как видно, корневой узел теперь шестой в выборке, а на первом месте подразделение, которое находится на втором уровне вложенности в дереве.

Для того, чтобы отсортировать данные, не нарушая их древовидной структуры, используется конструкция `ORDER SIBLINGS BY`. В этом случае сортировка будет применяться отдельно для каждой группы потомков в дереве:

DEPT_NAME	LEVEL
ЗАО ИнвестКорп	1
IT-отдел	2
Отдел QA	3
Отдел разработки	3
Бухгалтерия	2
Бухгалтерия по участку 1	3
Бухгалтерия по участку 2	3
Дирекция	2
Отдел продаж	2

Теперь узлы, находящиеся на одном уровне, сортируются в алфавитном порядке, при этом структура дерева не нарушена.

14.6. Нарушение древовидной структуры при выборке

Предположим, что мы хотим получить структуру подразделений начиная с тех, чьи названия содержат в себе слово *отдел*:

```
select *
from departments
start with upper(dept_name) like upper('%Отдел%')
connect by prior id = parent_id
```

ID	DEPT_NAME	PARENT_ID
3	Отдел продаж	1
4	IT-отдел	1
8	Отдел QA	4
9	Отдел разработки	4
8	Отдел QA	4
9	Отдел разработки	4

Некоторые строки дублируются, хотя в таблице имена подразделений не повторяются.

Теперь выполним тот же запрос, только добавим к списку колонок псевдостолбец `level`:

```
select id, dept_name, parent_id, level
from departments
start with upper(dept_name) like upper('%Отдел%')
connect by prior id = parent_id
```

ID	DEPT_NAME	PARENT_ID	LEVEL
3	Отдел продаж	1	1
4	IT-отдел	1	1
8	Отдел QA	4	2
9	Отдел разработки	4	2
8	Отдел QA	4	1
9	Отдел разработки	4	1

Теперь понятно, что строки дублируются из-за того, что они находятся на разных уровнях в дереве.

Разберем, почету так происходит, пройдя путь построения дерева:

1. В качестве корней дерева добавляются узлы, которые удовлетворяют условию, находящемуся в `START WITH`. Это *Отдел продаж*, *IT-отдел*, *Отдел QA* и *Отдел разработки*. Все они находятся на первом уровне вложенности в дереве.
2. Рекурсивно ищем потомков для всех выбранных на первом шаге узлов. Из всех них вложенность есть только у отдела IT - и внутри него как раз находятся *Отдел QA* и *Отдел разработки*, поэтому они добавляются со вторым уровнем вложенности.

Следует различать фактическое дерево, которое хранится в таблице, и то, которое получается при выборке, так как они могут не совпадать. На практике такая необходимость почти не встречается, и если при выборке данные не отражают той структуры, которая хранится в БД, то скорее всего запрос написан с ошибкой.

15. Операторы для работы с множествами

15.1. Объединение запросов

Предположим, что у нас есть 2 таблицы - таблица учителей `teachers` и таблица учеников `students`:

```
create table teachers(
    id number primary key,
    first_name varchar2(50) not null,
    last_name varchar2(100)
);

create table students(
    id number primary key,
    first_name varchar2(50) not null,
    last_name varchar2(100),
    group_id number
);

    insert into teachers values (1, 'Галина', 'Иванова');
    insert into teachers values (2, 'Нина', 'Сидорова');
    insert into teachers values (3, 'Евгения', 'Петрова');

    insert into students values (1, 'Александр', 'Обломов', 1);
    insert into students values (2, 'Николай', 'Рудин', 2);
    insert into students values (3, 'Евгения', 'Петрова', 1);
```

Перед нами стоит задача - нужно отобразить единым списком учителей и учеников.

Мы можем написать запрос для получения списка учителей:

```
select first_name, last_name
from teachers
```

Точно также можно получить список всех учеников:

```
select first_name, last_name
from students
```

Для того, чтобы эти данные "склеить", используется оператор `UNION`:

```
select first_name, last_name
from teachers

union

select first_name, last_name
from students
```

FIRST_NAME	LAST_NAME
Александр	Обломов
Галина	Иванова
Евгения	Петрова
Николай	Рудин
Нина	Сидорова

Если внимательно посмотреть на получившийся результат, то можно заметить, что данных в "склеенной" выборке стало меньше.

Все дело в том, то оператор UNION удаляет дубликаты из итоговой выборки. А так как у нас есть учитель "Евгения Петрова" и ученик "Евгения Петрова", то при объединении оставляется только одна строка.

Для того, чтобы объединить данные из нескольких запросов без удаления дубликатов, используется оператор `UNION ALL`:

```
select first_name, last_name
from teachers

union all

select first_name, last_name
from students
```

Если вы знаете, что в объединяемых данных не будет повторяющихся строк, используйте `UNION ALL`. В таком случае БД не будет тратить время на то, чтобы убрать дубликаты из итоговой выборки.

Для того, чтобы `UNION` работал, должны соблюдаться некоторые условия:

- Количество полей в каждой выборке должно быть одинаковым
- Поля должны иметь одинаковый тип

То есть, следующий запрос вернет ошибку, т.к. в первой части объединения запрос возвращает число первой колонкой, а второй - строку:

```
select id, first_name
from teachers

union

select first_name, last_name
from students
```

Результат - ошибка `ORA-01790: expression must have same datatype as corresponding expression`.

Кстати, псевдонимы столбцов не обязательно должны совпадать у всех частей соединения:

```
select first_name teacher_first_name, last_name teacher_last_name
from teachers

union

select first_name, last_name
from students
```

TEACHER_FIRST_NAME	TEACHER_LAST_NAME
Александр	Обломов
Галина	Иванова
Евгения	Петрова
Николай	Рудин
Нина	Сидорова

Следует обратить внимание на то, что в результирующей выборке псевдонимы для колонок взялись такие же, как и в запросе из первой части объединения. Если поменять эти части местами, то псеводнимы также изменятся:

```
select first_name, last_name
from students

union

select first_name teacher_first_name, last_name teacher_last_name
from teachers
```

FIRST_NAME	LAST_NAME
Александр	Обломов
Галина	Иванова
Евгения	Петрова
Николай	Рудин
Нина	Сидорова

15.2. Разница запросов

Подготовим тестовые данные:

```
create table cars(
    car_id number not null,
    car_model varchar2(100) not null,
    release_year number
);

create table car_offers(
    car_model varchar2(100) not null,
    release_year number
);

insert into cars
values(1, 'Volkswagen passat', 1998);

insert into cars
values(2, 'Volkswagen passat', 1998);

insert into cars
values(3, 'Mersedes SL', 2010);

insert into cars
values(4, 'Lexus S300', 2005);

insert into cars
values(5, 'Mersedes SL', 2008);

insert into car_offers
values('Lexus S300', 2010);

insert into car_offers
values('Tesla', 2017);

insert into car_offers
values('Volkswagen passat', 1998);

insert into car_offers
values('Volkswagen passat', 2003);
```

Посмотрим на данные в таблицах:

CAR_ID	CAR_MODEL	RELEASE_YEAR
1	Volkswagen passat	1998
2	Volkswagen passat	1998
3	Mersedes SL	2010
4	Lexus S300	2005
5	Mersedes SL	2008

Рисунок 1. Таблица cars

CAR_MODEL	RELEASE_YEAR
Lexus S300	2010
Tesla	2017
Volkswagen passat	1998
Volkswagen passat	2003

Рисунок 2. Таблица car_offers

Получим список предлагаемых нам моделей автомобилей, которых нет среди нашего автопарка. Для этого будем использовать оператор `MINUS`, который возвращает уникальные строки из первого запроса, которых нет во втором запросе:

```
select car_model
from car_offers

MINUS

select car_model
from cars
```

CAR_MODEL
Tesla

Если искать только отсутствующие у нас марки авто, то найдется лишь одна модель, которой нет у нас - `Tesla`.

Теперь получим предложения автомобилей, у которых *либо год, либо модель* не совпадают с теми авто, что есть у нас:

```
select car_model, release_year
from car_offers

MINUS

select car_model, release_year
from cars
```

CAR_MODEL	RELEASE_YEAR
Lexus S300	2010
Tesla	2017
Volkswagen passat	2003

```
Типы данных в колонках и их количество в
каждом из запросов
должны совпадать.
```

Если мы в первом запросе поменяем местами колонки, то запрос не выполнится и мы получим ошибку `ORA-01790: expression must have same datatype as corresponding expression`:

```
-- Ошибка, типы данных возвращаемых колонок  в
-- обоих запросах должны совпадать
select release_year, car_model
from car_offers

MINUS

select car_model, release_year
from cars
```

Если запросы возвращают неодинаковое количество колонок, при выполнении запроса получим ошибку `ORA-01789: query block has incorrect number of result columns`:

```
-- Ошибка, запросы должны возвращать
-- одинаковое количество колонок
select release_year
from car_offers

MINUS

select car_model, release_year
from cars
```

MINUS возвращает **уникальные** строки, которые отсутствуют во втором запросе.

Разберем это на примере. Для начала удалим из таблицы `car_offers` модели Volkswagen passat:

```
delete
from car_offers
where car_model = 'Volkswagen passat'
```

Теперь данные в таблице `car_offers` выглядят вот так:

CAR_MODEL	RELEASE_YEAR
Lexus S300	2010
Tesla	2017

Теперь получим список моделей авто, которые есть у нас, но отсутствуют в списке предложений:

```
select car_model, release_year
from cars

MINUS

select car_model, release_year
from car_offers
```

CAR_MODEL	RELEASE_YEAR
Lexus S300	2005
Mersedes SL	2008
Mersedes SL	2010
Volkswagen passat	1998

В результате мы видим всего одну строку с моделью Volkswagen passat 1998 года, несмотря на то, что в таблице `cars` таких записей две. Как было сказано, это произошло потому, что оператор `MINUS` удаляет дубликаты и возвращает только уникальные строки.

15.3. Пересечение запросов

В качестве тестовых данных будем использовать таблицы из примера про разность запросов.

Для получения пересечения данных между двумя запросами используется оператор `INTERSECT`. Он возвращает уникальные строки, которые присутствуют как в первом, так и во втором запросе.

Ограничения при использовании `INTERSECT` такие же, как и при использовании `UNION` и `MINUS`:

- Оба запроса должны возвращать одинаковое количество колонок
- Типы данных в колонках должны совпадать.

Получим список моделей автомобилей, которые есть и в автопарке, и в списке предлагаемых для покупки моделей:

```
select car_model, release_year
from cars

INTERSECT

select car_model, release_year
from car_offers
```

CAR_MODEL	RELEASE_YEAR
Volkswagen passat	1998

Как и в случае с `MINUS`, `INTERSECT` убрал дубликаты и оставил только одну модель авто, которая встречается и в таблице `cars`(2 раза), и в таблице `car_offers`(1 раз).

15.4. Следить за порядком колонок

При использовании операторов UNION, MINUS и INTERSECT нужно внимательно следить за порядком колонок в каждом из запросов, ведь несоблюдение порядка следования приведет к некорректным результатам.

Как было рассмотрено, Oracle будет проверять, чтобы тип колонок в каждом из запросов совпадал, но проверять, правильно ли расположены колонки одного типа, он не будет (потому что не сможет).

```
select car_model model, car_id release_year
from cars

minus

select car_model, release_year
from car_offers
```

MODEL	RELEASE_YEAR
Lexus S300	4
Mersedes SL	3
Mersedes SL	5
Volkswagen passat	1
Volkswagen passat	2

В запросе выше, в первой его части, вместо колонки `release_year` по ошибке была указана колонка `car_id`. Так как обе имеют числовой тип, ошибки не было, но данные на выходе получились ошибочными.

Следует внимательно следить за порядком колонок в каждом из запросов при использовании операторов для работы с множествами.

15.5. Сортировка

`ORDER BY` добавляется в конце запроса, и применяется к уже получившемуся в результате выполнения оператора множества набору данных.

Следующий пример получит список авто из имеющихся у нас, но отсутствующих в списке предлагаемых моделей, и отсортирует итоговую выборку по возрастанию года выпуска:

```
select car_model, release_year
from cars


minus

select car_model, release_year a2
from car_offers
order by release_year
```

Использовать сортировку в первом запросе нельзя, получим ошибку `ORA-00933: SQL command not properly ended`:

```
-- Ошибка!
select car_model, release_year
from cars
order by release_year

minus

select car_model, release_year
from car_offers
```

Как и в случае с обычными запросами, сортировать можно по порядковому номеру колонки итоговой выборки:

```
select car_model, release_year
from cars

minus

select car_model, release_year a2
from car_offers
-- Сортировка по модели авто
order by 1 desc
```

15.6. Приоритет выполнения

Между собой операторы множества имеют одинаковый приоритет. Если в запросе используется несколько таких операторов, то они выполняются последовательно.

```
select car_model, release_year
from cars

minus

select car_model, release_year
from car_offers

union all

select car_model, release_year
from cars
```

CAR_MODEL	RELEASE_YEAR
Lexus S300	2005
Mersedes SL	2008
Mersedes SL	2010
Volkswagen passat	1998
Volkswagen passat	1998
Mersedes SL	2010
Lexus S300	2005
Mersedes SL	2008

В данном примере сначала был выполнен оператор `MINUS`, и уже после к полученному результату был применен оператор `UNION ALL`.

Чтобы изменить порядок выполнения операторов, используются скобки:

```
select car_model, release_year
from cars

minus

-- minus будет применен
-- к результату выполнения
-- UNION ALL
(select car_model, release_year
from car_offers

UNION ALL

select car_model, release_year
from cars)
```

Здесь оператор `MINUS` будет применяться к набору данных, который получится в результате выполнения `UNION ALL`.

16. Подзапросы в Oracle

Подзапросы представляют собой обычные SQL-запросы, которые являются частью другого SQL-запроса.

Подзапросы - важная часть в изучении SQL. Некоторые данные просто не могут быть получены, если их не использовать. Далее будут рассмотрены примеры использования подзапросов в Oracle.

16.1. Подготовка тестовых данных

```
create table books(
    book_id number primary key,
    book_name varchar2(200) not null,
    author varchar2(50 char) not null,
    release_year number not null
);

create table book_orders(
    book_id number not null,
    quantity number(2) not null,
    order_date date not null
);


comment on table books is 'Книги';
comment on table book_orders is 'Статистика продаж за день';


insert into books
values(1, 'Властелин колец', 'Толкин', 1954);

insert into books
values(2, 'Гордость и предубеждение', 'Джейн Остин', 1813);

insert into books
values(3, 'Тёмные начала', 'Филип Пулман', 1995);

insert into books
values(4, 'Автостопом по галактике', 'Дуглас Адамс', 1979);

insert into book_orders
values(1, 1, to_date('31.12.2005', 'dd.mm.yyyy'));

insert into book_orders
values(1, 4, to_date('30.12.2005', 'dd.mm.yyyy'));

insert into book_orders
values(2, 2, to_date('10.05.2005', 'dd.mm.yyyy'));

insert into book_orders
values(2, 1, to_date('12.05.2005', 'dd.mm.yyyy'));

insert into book_orders
values(3, 2, to_date('05.11.2005', 'dd.mm.yyyy'));
```

16.2. Подзапросы в where- части запроса

Получим информацию о продажах книги "Властелин колец":

```
select bo.*
from book_orders bo
where bo.book_id = (select book_id from books where book_name = 'Властелин колец');
```

BOOK_ID	QUANTITY	ORDER_DATE
1	1	31-DEC-05
1	4	30-DEC-05

Здесь использовался подзапрос, чтобы определить id книги с названием 'Властелин колец'.

Если выполнить подзапрос отдельно:

```
select book_id
from books
where book_name = 'Властелин колец'
```

То мы получим одну строку, которая будет содержать значение `book_id`, равое 1. Поэтому самый первый запрос эквивалентен следующему:

```
select bo.*
from book_orders bo
where bo.book_id = 1
```

Следует обратить внимание на то, что в данном случае подзапрос должен возвращать только одну строку, состоящую из одной колонки. Следующие запросы работать не будут:

```
select bo.*
from book_orders bo
where bo.book_id = (select book_id, book_name from books where book_name = 'Властелин
колец')
```

Данный запрос выдаст ошибку `ORA-00913: too many values`, т.к. подзапрос возвращает одну строку с двумя колонками.

```
select bo.*
from book_orders bo
where bo.book_id = (select book_id from books)
```

А здесь будет ошибка `ORA-01427: single-row subquery returns more than one row`, что переводится как "однострочный подзапрос возвращает более одной строки". Из-за этого результат выполнения данного подзапроса нельзя подставить в условие сравнения, т.к. сравнение должно работать с одиночными значениями.

16.3. Подзапросы в select-части

Подзапросы, которые возвращают одиночные значения, можно использовать прямо в части `SELECT` в качестве колонок. Результат выполнения подзапроса будет добавляться к каждой строке, как обычная колонка:

```
select b.*,
       (select count(*) from book_orders) ord_cnt
from books b
```

BOOK_ID	BOOK_NAME	AUTHOR	RELEASE_YEAR	ORD_CNT
1	Властелин колец	Толкин	1954	5
2	Гордость и предубеждение	Джейн Остин	1813	5
3	Тёмные начала	Филип Пулман	1995	5
4	Автостопом по галактике	Дуглас Адамс	1979	5

Здесь мы добавили колонку `ord_cnt`, которая содержит количество всех имеющихся заказов по всем книгам.

Здесь также нельзя, чтобы запрос возвращал несколько колонок или несколько строк. Зато запрос может ничего не возвращать, тогда значение в колонке будет `NULL`:

BOOK_ID	BOOK_NAME	AUTHOR	RELEASE_YEAR	BOOK_ID_SUBQ
1	Властелин колец	Толкин	1954	-
2	Гордость и предубеждение	Джейн Остин	1813	-
3	Тёмные начала	Филип Пулман	1995	-
4	Автостопом по галактике	Дуглас Адамс	1979	-

Т.к. утверждение `2 > 10` ложно, подзапрос не вернет ни одной записи, поэтому значение в соответствующей колонке будет `NULL`.

16.4. Подзапросы во FROM части

Подзапросы можно использовать во FROM части запроса, и обращаться к данным, которые они возвращают, как с полноценной таблицей(в пределах запроса; каким-либо образом удалить или изменить данные в подрапросе не получится).

```
select b_orders.*
from (
select b.book_id, b.book_name, bo.quantity, bo.order_date
from books b
join book_orders bo on bo.book_id = b.book_id) b_orders
where b_orders.quantity > 1
```

BOOK_ID	BOOK_NAME	QUANTITY	ORDER_DATE
1	Властелин колец	4	30-DEC-05
2	Гордость и предубеждение	2	10-MAY-05
3	Тёмные начала	2	05-NOV-05

Здесь мы написали отдельный запрос, дали ему псевдоним `b_orders`, поместили его во `FROM` часть, как будто это обычная таблица, и дальше работаем с псевдонимом данного подзапроса.

В подзапросе использовались соединения

Сам подзапрос можно выполнить отдельно:

```
select b.book_id, b.book_name, bo.quantity, bo.order_date
from books b
join book_orders bo on bo.book_id = b.book_id
```

BOOK_ID	BOOK_NAME	QUANTITY	ORDER_DATE
1	Властелин колец	1	31-DEC-05
1	Властелин колец	4	30-DEC-05
2	Гордость и предубеждение	2	10-MAY-05
2	Гордость и предубеждение	1	12-MAY-05
3	Тёмные начала	2	05-NOV-05

Как можно заметить, там есть строки, в которых количество(столбец `quantity`) равен 1.

Но в первом примере этих строк нет, т.к. мы прописали условие `where b_orders.quantity > 1`.

Подзапросов во `FROM` части может быть несколько, т.е. мы можем соединять их, как обычные таблицы(опять, про соединения таблиц можно почитать вот здесь.

В отличие от подзапросов, которые используются в select-части, данные подзапросы могут возвращать более одной строки (более того, как правило, они и возвращают много строк, иначе зачем их использовать?).

16.5. Коррелированные подзапросы

Коррелированный подзапрос - это такой подзапрос, который использует для своей работы данные из внешнего по отношению к нему запроса. Например:

```
select b.*,
       (select count(*) from book_orders where book_id = b.book_id) ord_cnt
from books b
```

BOOK_ID	BOOK_NAME	AUTHOR	RELEASE_YEAR	ORD_CNT
1	Властелин колец	Толкин	1954	2
2	Гордость и предубеждение	Джейн Остин	1813	2
3	Тёмные начала	Филип Пулман	1995	1
4	Автостопом по галактике	Дуглас Адамс	1979	0

Здесь подзапрос подсчитывает количество дней, в которые производились продажи определенной книги. Т.е. подзапрос считает количество строк в таблице `book_orders` по значению колонки `book_id`, которую он берет из внешнего запроса. В условии прописывается `where book_id = b.book_id`, что означает: "Возьми для каждой строки из основного запроса значение колонки `book_id`, и посчитай количество строк в таблице `book_orders` с таким же `book_id`."

16.6. Подзапросы в IN, NOT IN

Ранее уже рассматривались примеры и особенности использования `IN` и `NOT IN` в SQL. В качестве перечисляемых значений в этих операторах были значения, которые прописывал сам программист. На практике чаще всего в качестве источника для значений в этих операторах используются подзапросы:

```
select b.*
from books b
where b.book_id in (
    select book_id
    from book_orders bo
    where bo.quantity < 2)
```

BOOK_ID	BOOK_NAME	AUTHOR	RELEASE_YEAR
2	Гордость и предубеждение	Джейн Остин	1813
3	Тёмные начала	Филип Пулман	1995

Данный запрос выводит список книг, у которых были продажи менее, чем по 2 штуки в день.

Список книг для оператора `IN` формируется в результате выполнения подзапроса, а не ручного кодирования значений программистом.

Подзапросы в `IN` и `NOT IN` должны возвращать строки с одной колонкой. Следующий запрос выдаст ошибку `ORA-00913: too many values`, т.к. подзапрос получает список строк с двумя колонками:

```
select b.*
from books b
where b.book_id in (
    select book_id, quantity
    from book_orders bo
    where bo.quantity < 2)
```

При этом не следует забывать об особенности использования `NOT IN`: Если в списке значений для проверки есть хотя бы одно `NULL`-значение, то результат выражения будет ложным, и запрос не вернет никаких данных:

```
select b.*
from books b
where b.book_id  not in (
    select book_id
    from book_orders bo
    where bo.quantity < 2

    union

    select null
    from dual)
```

```
no data found
```

Здесь при помощи объединения] запросов в выборку подзапроса была добавлена строка с одним `NULL`-значением, и как следствие, запрос не вернул никаких данных.

17. Псевдостолбцы в Oracle

К псевдостолбцам можно относиться как к обычным колонкам в таблице, за тем лишь исключением, что данные, которые они представляют, в таблице не хранятся.

Некоторые псевдостолбцы доступны только в определенном контексте, например, лишь при использовании рекурсивных запросов.

Мы рассмотрим не все псевдостолбцы, доступны в Oracle, а лишь самые основные и часто используемые. Полный их список и описание можно почитать в документации.

17.1. Подготовка данных

Мы будем использовать таблицу `dishes`, которая создается в части про операторы сравнения.

17.1.1. ROWNUM

Данный псевдостолбец возвращает порядковый номер, под которым Oracle выбирает строку из таблицы. Для первой строки значение `ROWNUM` будет равно 1, для второй - 2, и т.д.

Один из классических примеров использования `ROWNUM` - ограничение количества получаемых строк из таблицы:

```
select d.*
from dishes d
where rownum < 3
```

NAME	PRICE	RATING
Макароны с сыром	20.56	320
Борщ	10	130

Если в запросе используется сортировка, то она может изменить порядок строк. Т.е. строка из таблицы могла получаться первой, и ей мог быть присвоен rownum = 1, но *после* того, как все строки были получены, они были отсортированы в другом порядке:

```
select d.*, rownum
from dishes d
where rownum < 6
order by price asc
```

NAME	PRICE	RATING	ROWNUM
Чай с молоком	1.2	280	4
Чай с лимоном	1.34	270	3
Борщ	10	130	2
Макароны с сыром	20.56	320	1
Свиная отбивная	30.5	320	5

Что же делать, если мы хотим пронумеровать наши строки начиная от 1 таким образом, чтобы у самого дешевого блюда был номер 1, у более дорогого - 2 и т.п.?

Для этого можно использовать подзапросы:

```
select dishes_ordered.*, rownum
from (
  select d.*
  from dishes d
  order by price asc
) dishes_ordered
```

NAME	PRICE	RATING	ROWNUM
Чай с молоком	1.2	280	1
Чай с лимоном	1.34	270	2
Овощной салат	5.7	-	3
Борщ	10	130	4
Макароны с сыром	20.56	320	5
Свиная отбивная	30.5	320	6

Теперь все будет работать, т.к. сортировка данных была произведена в подзапросе еще **до того, как данные будут получаться внешним запросом**, а значит и до того, как каждой строке будет присваиваться значение `ROWNUM`.

Следует отметить, что использование оператора `>` с `ROWNUM` не имеет смысла. Рассмотрим это на примере:

```
select d.*
from dishes d
where rownum > 3
```

Этот запрос ничего не выведет, несмотря на то, что строк в таблице больше трех. Все потому, что `rownum` хранит в себе номер строки, под которым Oracle получает ее из таблицы или соединения. В примере выше у первой строки(какой бы она не была, она все равно будет первой) значение `rownum` будет равно 1. Это значит, что условие `rownum > 3` будет ложным, и строка не будет добавлена в выборку. Следующая строка опять будет иметь `rownum = 1`, что опять приведет значение условия в False, и так будет для всех строк из таблицы `dishes`.

17.1.2. Top-N query

Получим топ-3 блюда по рейтингу с помощью `rownum`:

```
select rt.name,
       rt.price,
       rt.rating,
       ROWNUM
from (
  select d.*
  from dishes d
  order by d.rating desc nulls last
) rt
where ROWNUM <= 3
```

NAME	PRICE	RATING	ROWNUM
Макароны с сыром	20.56	320	1
Свиная отбивная	30.5	320	2
Чай с молоком	1.2	280	3

Подобного рода запросы относятся к так называемым *top-N queries*, т.е. они получают часть данных, основываясь на каком-либо критерии сортировки (в данном случае это рейтинг блюд).

17.2. ROWID

`ROWID` содержит в себе адрес строки в таблице. На практике он используется не часто, но иногда его значение может понадобиться сторонним библиотекам.

Сам `rowid` уникально идентифицирует определенную строку в таблице, но это не означает, что `rowid` уникален в пределах всей базы данных.

Значение `rowid` нельзя использовать для того, чтобы ссылаться на определенную строку в таблице, т.к. оно может измениться.

Для примера просто получим все строки из таблицы `dishes` с их `rowid`:

```
select rowid, d.name
from dishes d
```

ROWID	NAME
ACCyF+ADDAAABIkAAA	Макароны с сыром
ACCyF+ADDAAABIkAAB	Борщ
ACCyF+ADDAAABIkAAC	Чай с лимоном
ACCyF+ADDAAABIkAAD	Чай с молоком
ACCyF+ADDAAABIkAAE	Свиная отбивная
ACCyF+ADDAAABIkAAF	Овощной салат

17.3. level

Данный псевдостолбец доступен только в рекурсивных запросах. Подробнее про него можно почитать в части про рекурсивные запросы.

18. Функции для работы со строками

Сначала создадим тестовую таблицу.

```
create table articles(
    title varchar2(50) not null,
    author varchar2(50) not null,
    msg varchar2(300) not null,
    publish_date date not null
);

comment on table articles is 'Твиты';
comment on column articles.title is 'Заголовок';
comment on column articles.author is 'Автор';
comment on column articles.msg is 'Сообщение';

insert into articles values ('Новый фотоаппарат!', 'johndoe',
    'Сегодня купил себе новый фотоаппарат. Надеюсь, у меня будут получаться отличные
фотографии!', sysdate);
insert into articles values ('Насобирал денег', 'johndoe',
    'Целый год я шел к этой цели, и вот наконец-то у меня все получилось, и заветная
сумма собрана!', sysdate - 1);
insert into articles values ('Задался целью', 'johndoe',
    'Итак, я задался целью купить себе фотоаппарат. Для начала нужно насобирать денег
на него.', sysdate - 2);
insert into articles values ('Сходил в ресторан!', 'user003',
    'Пришел из ресторана. Еда была просто восхитительна!', sysdate - 3);
insert into articles values ('Съездили в отпуск!', 'artem69',
    'Наконец-то выбрались с женой и детьми в отпуск, было замечательно!', sysdate - 4
);
```

Таблица articles представляет собой место хранения сообщений пользователей, что-то вроде twitter.

18.1. upper, lower

Данные функции уже описывались раньше.

- `UPPER`: приводит строку к верхнему регистру
- `LOWER`: приводит строку к нижнему регистру

Рекомендуется использовать одну из этих строк, если нужно сравнить две строки между собой без учета регистра символов.

18.2. Конкатенация строк

Конкатенация - это обычная "склейка" строк. Т.е., если у нас есть 2 строки - `Новый`, `фотоаппарат`, то результатом конкатенации будет строка `Новый фотоаппарат`.

Для склейки строк в Oracle используется оператор `||`.

```
select 'Автор:' || art.author frmt_author,
       'Заголовок:"' || art.title  || '"' frmt_title
from articles art
```

FRMT_AUTHOR	FRMT_TITLE
Автор:johndoe	Заголовок:"Новый фотоаппарат!"
Автор:johndoe	Заголовок:"Насобирал денег"
Автор:johndoe	Заголовок:"Задался целью"
Автор:user003	Заголовок:"Сходил в ресторан!"
Автор:artem69	Заголовок:"Съездили в отпуск!"

18.3. Поиск подстроки

Для того, чтобы найти вхождение одной строки в другую, используется функция INSTR. Она возвращает позицию вхождения одной строки в другую. Если вхождения не обнаружено, то в качестве результата будет возвращет 0.

Следующий запрос возвращает позицию, начиная с которой в заголовках записей пользователей встречается символ восклицательного знака:

```
select a.title,
       instr(a.title, '!') pos
from articles a
```

TITLE	POS
Новый фотоаппарат!	18
Насобирал денег	0
Задался целью	0
Сходил в ресторан!	18
Съездили в отпуск!	18

Как видно, для тех заголовков, которые не содержат восклицательный знак, функция INSTR вернула 0.

В функции INSTR можно задавать позицию, начиная с которой следует производить поиск вхождения:

```
select a.title,
       instr(a.title, 'o', 3) pos
from articles a
```

TITLE	POS
Новый фотоаппарат!	8
Насобирал денег	4
Задался целью	0
Сходил в ресторан!	3
Съездили в отпуск!	12

Данный запрос вернет позицию буквы о в заголовках записей, но поиск будет производить лишь начиная с 3-го символа заголовка.

Так, в строке **Новый фотоаппарат** мы получили результат 8, хотя буква о есть и раньше - на второй позиции.

В качестве стартовой позиции поиска можно указывать отрицательное число. В этом случае функция отсчитает от конца строки указанное количество символов и будет производить поиск начиная от этой позиции и заканчивая началом строки:

```
select a.title,
       instr(a.title, 'a', -4) pos
from articles a
```

TITLE	POS
Новый фотоаппарат!	14
Насобирал денег	8
Задался целью	4
Сходил в ресторан!	0
Съездили в отпуск!	0

Также можно указать, какое по счету совпадение нужно искать(4-ый параметр в функции INSTR):

```
select a.title,
       instr(a.title, 'o', 1, 2) pos
from articles a
```

TITLE	POS
Новый фотоаппарат!	8
Насобирал денег	0
Задался целью	0
Сходил в ресторан!	14
Съездили в отпуск!	0

18.4. Подобие строк. Like

Предположим, нам понадоболось посмотреть, какие чаи есть у нас в меню. В данном примере единственный способ, которым мы можем определить, что блюдо является чаем - это проверить, содержится ли слово чай в наименовании.

Но оператор сравнения здесь не подойдет, так как он вернет лишь те строки, которые будут полностью совпадать со строкой Чай.

Перед рассмотрением примера добавим в таблицу меню немного чайных блюд:

```
insert into dishes(name, price, rating) values ('Зеленый чай', 1, 100);
insert into dishes(name, price, rating) values ('Чай%', 2, 100);
insert into dishes(name, price, rating) values ('Чай+', 1, 200);
insert into dishes(name, price, rating) values ('Чай!', 1, 666);
```

Гениальные маркетологи решили, что будут добавлять по одному символу в конце слова чай для обозначения его крепости - **чай%** - совсем слабенький, **чай+** взбодрит с утра, а с **чаем!** можно забыть про сон на ближайшие сутки. Не будем задумываться, почему именно так, а просто примем это как есть.

Итак, первый пример использования `LIKE`:

```
select d.*
from dishes d
where d.name like 'Чай%'
```

NAME	PRICE	RATING
Чай с лимоном	1.34	270
Чай с молоком	1.2	280
Чай%	2	100
Чай+	1	200
Чай!	1	666

Как видно, были получены все блюда, наименования которых начиналось с последовательности символов, составляющей слово Чай. Символ `%` в условии `LIKE` соответствует любой последовательности символов. Т.е. предыдущий запрос можно было читать так: "Получить все блюда, первые символы наименований которых составляют слово Чай, а после этих символов следует последовательность из любых символов в любом количестве, мне не важно". Кстати, в результат не попал зеленый чай - первые 3 символа наименования у него равны **"Зел"**, но никак не **"Чай"**.

Если не указывать символ `%`, то запрос не вернет никаких данных:

```
select d.*
from dishes d
where d.name like 'Чай'
```

```
no data found
```

При задании шаблонов в LIKE можно использовать следующие символы соответствия:

- %(знак процента). Ему соответствует 0 или больше символов в значении.
- _(нижнее подчеркивание). Ему соответствует ровно один символ в значении.

Получим все чаи, названия которых придумали маркетологи(а это любой 1 символ после слова **"чай"**):

```
select d.*
from dishes d
where d.name like ('Чай_')
```

NAME	PRICE	RATING
Чай%	2	100
Чай+	1	200
Чай!	1	666

Также, как и при обычном сравнении, учитывается регистр строк. Следующий запрос не вернет никаких данных, т.к. нет блюд, начинающихся со строки **"чай"**, есть только блюда, начинающиеся на **"Чай"**(первая буква заглавная):

```
select d.*
from dishes d
where d.name like ('чай%')
```

```
no data found
```

Получим только зеленый чай:

```
select d.*
from dishes d
where d.name like ('%чай')
```

NAME	PRICE	RATING
Зеленый чай	1	100

Здесь символ процента был перемещен перед словом **"чай"**, что означает: "Любая последовательность символов(или их отсутствие), заканчивающаяся словом чай".

А для того, чтобы получить список всех блюд, в наименовании которых содержится слово **"чай"**, можно написать следующий запрос:

```
select d.*
from dishes d
where upper(d.name) like upper('%чай%')
```

NAME	PRICE	RATING
Чай с лимоном	1.34	270
Чай с молоком	1.2	280
Зеленый чай	1	100
Чай%	2	100
Чай+	1	200
Чай!	1	666

18.4.1. Выражение ESCAPE в LIKE

Перед рассмотрением выражения опять добавим немного данных в таблицу `dishes`:

```
insert into dishes values ('Кофе(0.4% кофеина)', 30, 20);
insert into dishes values ('Кофе(0.3% кофеина)', 30, 20);
insert into dishes values ('Кофе(0.1% кофеина)', 30, 20);
insert into dishes values ('Кофе(без кофеина)', 30, 20);
```

Перед нами стоит задача: получить список кофейных блюд, содержащих кофеин.

Можно выделить некоторый список признаков, по которым мы сможем определить, что кофе с кофеином:

- Наименование начинается со слова "Кофе"
- Если кофе с кофеином, то в скобках указывается его процентное содержание в виде **"n% кофеина"**, где n - некоторое число.

На основании этих заключений можно написать следующий запрос:

```
select d.*
from dishes d
where d.name like ('Кофе%кофеина')
```

```
no data found
```

В чем проблема, должно быть понятно - в том, что символ “%” в условии `LIKE` обозначает совпадение с 0 или больше любых символов.

Для того, чтобы учитывать непосредственно символ “%” в строке, условие `LIKE` немного видоизменяется:

```
select d.*
from dishes d
where d.name like ('Кофе%\% кофеина%') escape '\'
```

NAME	PRICE	RATING
Кофе(0.4% кофеина)	30	20
Кофе(0.3% кофеина)	30	20
Кофе(0.1% кофеина)	30	20

Здесь после ключевого слова escape мы указываем символ, который будет экранирующим, т.е. если перед символами`%` будет стоять символ `\`, то он будет рассматриваться как совпадение с одним символом `%`, а не как совпадение 0 и больше любых символов.

18.5. INITCAP

Функция `INITCAP` делает первую букву каждого слова заглавной, оставляя остальную часть слова в нижнем регистре.

```
select initcap(art.author)
from articles art
```

AUTHOR_INITCAP
Johndoe
Johndoe
Johndoe
User003
Artem69

```
select initcap(art.msg) msg_initcap
from articles art
```

MSG_INITCAP
Сегодня Купил Себе Новый Фотоаппарат. Надеюсь, У Меня
Целый Год Я Шел К Этой Цели, И Вот Наконец-То У Меня Е
Итак, Я Задался Целью Купить Себе Фотоаппарат. Для На
Пришел Из Ресторана. Еда Была Просто Восхитительна!

Если строка состоит из нескольких слов, то в каждом из этих слов первая буква будет заглавной, а остальные - прописными.

18.6. Замена подстроки

Для замены подстроки в строке используется функция `REPLACE`. Данная функция принимает 3 параметра, из них последний - не обязательный:

```
replace(исходная_строка, что_меняем, на_что_меняем)
```

В случае, если не указать, на какую строку производить замену, то совпадения будут просто уделены из исходной строки.

Например, получим все "твиты" пользователя johndoe, но в заголовке поста заменим слово "фотоаппарат" заменим на слово "мыльница":

```
select replace(a.title, 'фотоаппарат', 'мыльница') new_title,
       a.msg
from articles a
where a.author = 'johndoe'
```

NEW_TITLE	
Новый мыльница!	Сегодня купил
Насобирал денег	Целый год я ш
Задался целью	Итак, я задал

18.7. Trim. Удаление пробелов в конце/начале строки

Есть 3 основных функции для удаления "лишних" пробелов из строки:

- `trim` - удалить пробелы вначале и в конце строки
- `ltrim` - удалить пробелы вначале строки (слева)
- `rtrim` - удалить пробелы в конце строки (справа)

```
select trim('    John Doe      ') from dual;
select rtrim('    John Doe     ') from dual;
select ltrim('    John Doe    ') from dual;
-- То же самое, что и trim
select ltrim(rtrim('     John Doe    ')) from dual;
```

18.8. LPAD, RPAD

Эти функции используются, чтобы дополнить строку какими-либо символами до определенной длины.

`LPAD` (left padding) используется для дополнения строки символами слева, а `RPAD` (right padding) - для дополнения справа.

```
select lpad('1', 5, '0') n1,
       lpad('10', 5, '0') n2,
       lpad('some_str', 10) n2_1,
       rpad('38', 5, '0') n3,
       rpad('3', 5, '0') n4
from dual
```

N1	N2	N2_1	N3	N4
00001	00010	some_str	38000	30000

Первый параметр в этой функции - строка, которую нужно дополнить, второй - длина строки, которую мы хотим получить, а третий - символы, которыми будем дополнять строку. Третий параметр не обязателен, и если его не указывать, то строка будет дополняться пробелами, как в колонке `n2_1`.

19. Функции для работы с NULL

Так как `NULL` - особое значение, то он удостоился отдельных функций в Oracle, которые умеют работать с ним "из коробки".

Работать будем со следующей таблицей:

```
create table profiles(
    login varchar2(30) primary key,
    last_updated date,
    status varchar2(50)
);

comment on table profiles is 'Профили форума';
comment on column profiles.last_updated is 'Дата последнего обновления';
comment on column profiles.status is 'Статус';

insert into profiles(login, last_updated, status)
values ('johndoe', to_date('01.01.2009 23:40', 'dd.mm.yyyy hh24:mi'), '');

insert into profiles(login, last_updated, status)
values ('admin', to_date('01.01.2019 21:03', 'dd.mm.yyyy hh24:mi'), 'Я админ. Все
вопросы ко мне');

insert into profiles(login, last_updated, status)
values ('alisa', null, 'Окажу помощь в проектировании домов');

insert into profiles(login, last_updated, status)
values ('nelsol', null, null);
```

19.1. Nvl

```
select nvl(2, 10) nvl_1,
    nvl(null, 20) nvl_2
from dual
```

Данная функция принимает 2 параметра. Если первый параметр равен `NULL`, то будет возвращен второй параметр. В противном случае функция вернет первый параметр.

```
select pf.login,
    pf.last_updated,
    nvl(pf.status, '<нет данных>') status
from profiles pf
```

LOGIN	LAST_UPDATED	STATUS
johndoe	01-JAN-09	<нет данных>
admin	01-JAN-19	Я админ. Все вопросы ко мне
nelsol	-	<нет данных>

Здесь мы получаем данные из таблицы профилей, и в том случае, если статус пуст, выводим строку *"<нет данных>"*.

19.2. Nvl2

Функция `nvl2` работает немного сложнее. Она принимает 3 параметра. В том случае, если первый параметр не `NULL`, она вернет второй параметр. В противном случае она вернет третий параметр:

```
select pf.login,
    pf.last_updated,
    nvl2(pf.status, 'статус указан', 'статус не указан') status
from profiles pf
```

LOGIN	LAST_UPDATED	STATUS
johndoe	01-JAN-09	статус не указан
admin	01-JAN-19	статус указан
nelsol	-	статус не указан

19.3. Coalesce

Данная функция принимает на вход любое количество параметров и возвращает первое, из них, которое не является `NULL`:

```
select login,
       coalesce(status, 'статус пуст') first_not_null,
       coalesce(status, null, null, 'статус пуст') first_not_null_1,
       coalesce('статус всегда заполнен', status) first_not_null_2
from profiles
```

LOGIN	FIRST_NOT_NULL	FIRST_NOT_NULL_1	FIRST_NOT_NULL_2
johndoe	статус пуст	статус пуст	статус всегда заполнен
admin	Я админ. Все вопросы ко мне	Я админ. Все вопросы ко мне	статус всегда заполнен
nelsol	статус пуст	статус пуст	статус всегда заполнен

В том случае, если в функцию `COALESCE` передаются параметры разных типов, то все параметры будут приведены к типу первого `NOT NULL` аргумента.

В том случае, если этого сделать не получится, будет выброшена ошибка:

```
select coalesce(null, 'String', 'String_2') not_null_str
from dual
```

NOT_NULL_STR
String

```
select coalesce(null, 'String', 23.4) not_null_str
from
```

```
ORA-00932: inconsistent datatypes: expected CHAR got NUMBER
```

При этом, если число привести к строке самим, все будет работать, как ожидается:

```
select coalesce(null, 'String', to_char(23.4)) not_null_str
from dual
```

NOT_NULL_STR
String

20. Агрегирующие функции

Агрегирующие функции - это такие функции, которые выполняются не для каждой строки отдельно, а для определенных групп данных.

20.1. Подготовка данных

```
create table employees(
    id number not null,
    first_name varchar2(50 char) not null,
    last_name varchar2(100 char),
    bd date not null,
    job varchar2(100)
);

insert into employees
values(1, 'Василий', 'Петров', to_date('07.10.1990', 'dd.mm.yyyy'), 'Машинист');

insert into employees
values(2, 'Александр', 'Сидоров', to_date('18.07.1980', 'dd.mm.yyyy'), 'Бухгалтер');

insert into employees
values(3, 'Евгения', 'Цветочкина', to_date('18.07.1978', 'dd.mm.yyyy'), 'Бухгалтер');

insert into employees
values(4, 'Владимир', 'Столяров', to_date('18.07.1977', 'dd.mm.yyyy'), 'Слесарь');
```

Например, следующий запрос найдет минимальную дату рождения среди всех сотрудников:

```
select min(bd)
from employees
```

MIN(BD)
18-JUL-77

```
select min(bd) minbd, max(bd) maxbd
from employees
```

MINBD	MAXBD
18-JUL-77	07-OCT-90

Здесь также были добавлены псевдонимы `minbd` и `maxbd` для колонок.

Агрегирующие фунции могут быть использованы в выражениях:

```
select min(bd) + 1 minbd, add_months(max(bd), 2) maxbd
from employees
```

MINBD	MAXBD
19-JUL-77	07-DEC-90

Но получение одной-единственной даты мало что дает, хотелось бы видеть больше данных, соответствующих минимальной или максимальной дате в наборе данных.

```
select min(bd), max(bd), first_name
from employees
group by first_name
```

MIN(BD)	MAX(BD)	FIRST_NAME
18-JUL-78	18-JUL-78	Евгения
07-OCT-90	07-OCT-90	Василий
18-JUL-80	18-JUL-80	Александр
18-JUL-77	18-JUL-77	Владимир

Если посмотреть на результат запроса, то все равно трудновато понять, что дают в этом примере добавление имени и группировка записей по нему. Для лучшего понимания добавим в таблицу еще пару записей:

```
insert into employees
values(5, 'Евгения', 'Кукушкина', to_date('18.07.1989', 'dd.mm.yyyy'), 'Арт-директор'
);

insert into employees
values(6, 'Владимир', 'Кукушкин', to_date('22.05.1959', 'dd.mm.yyyy'), 'Начальник
департамента охраны');
```

Теперь выполним запрос еще раз:

```
select min(bd), max(bd), first_name
from employees
group by first_name
```

MIN(BD)	MAX(BD)	FIRST_NAME
18-JUL-78	18-JUL-89	Евгения
07-OCT-90	07-OCT-90	Василий
18-JUL-80	18-JUL-80	Александр
22-MAY-59	18-JUL-77	Владимир

Теперь можно заметить несколько особенностей:

- Количество строк не изменилось
- В строке с именем "Евгения" изменилась максимальная дата рождения
- В строке с именем "Владимир" изменилась минимальная дата рождения

Теперь видно, что агрегирующие функции могут применяться не ко всему набору данных, а к определенным частям этого набора. В данном случае группы были разбиты по именам, т.е. 2 записи с именем "Евгения", 2 записи с именем "Владимир", а остальные записи представляют собой отдельные группы из одного элемента.

При этом следует обратить внимание, что несмотря на то, что остальные колонки в строках с именем "Евгения" или "Владимир" отличаются между собой, они все равно попадают в одну группу, т.к. группировка производится *только* по имени.

20.2. Having

Выведем список имен, которые встречаются более одного раза:

```
select first_name, count(*)
from employees
group by first_name
having count(*) > 1
```

FIRST_NAME	COUNT(*)
Евгения	2
Владимир	2

`Having` работает аналогично условию `where`, но только для значений агрегатных функций.

21. Работа с датами в Oracle

В БД Oracle для работы с датами предназначены 2 типа - `DATE` и `TIMESTAMP`.

Отдельно можно упомянуть `INTERVAL` - интервальный тип, который хранит диапазон между двумя датами.

21.1. Date

Тип `DATE` используется чаще всего, когда необходимо работать с датами в БД Oracle. Он позволяет хранить даты с точностью *до секунд*.

Некоторые БД, например MySQL, также имеют тип `DATE`, но там может храниться дата лишь с точностью до дня.

21.2. Приведение строки к дате

Одна из часто встречающихся ситуаций - необходимость представить строку в виде типа данных `DATE`. Делается это при помощи функции `to_date`. Данная функция принимает 2 параметра - строку, содержащую в себе собственно дату, и строку, которая указывает, как нужно интерпретировать первый параметр, т.е. где в этой дате год, где месяц, число и т.п.

```
select to_date('2020-03-01', 'yyyy-mm-dd') d1, ①
to_date('2020-03-01', 'yyyy-dd-mm') d2 ②
from dual
```

① 1 марта 2020 года

② 3 января 2020 года

На самом деле, функция `to_date` может работать и без строки с форматом даты, а также с еще одним дополнительным параметром, который будет указывать формат языка, но мы будем рассматривать вариант с двумя параметрами. Более детально ознакомиться с функцией `to_date` можно вот здесь.

Как видно, строка, определяющая формат даты, имеет очень большое значение. В примере выше, мы получили две разные даты, изменив лишь их формат в функции `to_date`.

21.3. Функция SYSDATE

Данная функция возвращает текущую дату. В зависимости от того, когда следующий запрос выполнится, значение `SYSDATE` будет всегда разным.

```
select sysdate -- вернет текущую дату
from dual
```

21.4. Приведение даты к строке

Чтобы отобразить дату в нужном нам формате, используется функция `to_char`.

```
select to_char(sysdate, 'yyyy-mm-dd') d1,
       to_char(sysdate, 'dd.mm.yyyy') d2,
       to_char(sysdate, 'dd.mm.yyyy hh24:mi') d3,
       to_char(sysdate, 'hh24:ss yyyy.mm.dd') d4
from dual
```

D1	D2	D3	D4
2020-04-12	12.04.2020	12.04.2020 00:04	00:35 2020.04.12

21.5. trunc

Функция `trunc` округляет дату до определенной точности. Под точностью в округлении даты следует понимать ту ее часть(день, месяц, год, час, минута), которая не будет приведена к единице, а будет такой же, как и в исходной дате.

```
select trunc(sysdate, 'hh24'),
       trunc(sysdate, 'dd'), ②
       trunc(sysdate), ③
       trunc(sysdate, 'mm'),
       trunc(sysdate, 'yyyy')
from dual
```

Если не указывать формат округления, то trunc округлит до дней, т.е. колонки `2` и `3` будут содержать одинаковое значение.

21.6. add_months

Функция `add_months` добавляет указанное количество месяцев к дате. Для того, чтобы отнять месяцы от даты, нужно передать в качестве второго параметра отрицательное число:

```
select add_months(sysdate, 1) d1,
       add_months(sysdate, 6) d2, ①
       add_months(sysdate, -6) d3 ②
from dual
```

① полгода после текущей даты

② полгода до текущей даты

D1	D2	D3
12-MAY-20	12-OCT-20	12-OCT-19

21.7. Разница между датами

Если просто отнять от одной даты другую, то мы получим разницу между ними в днях. Также, к датам можно прибавлять и отнимать обычные числа, и Oracle будет оперировать ими как днями:

```
select to_date('2020-03-05', 'yyyy-mm-dd') - to_date('2020-03-01', 'yyyy-mm-dd') a,
       to_date('2020-03-05 01:00', 'yyyy-mm-dd hh24:mi') - to_date('2020-03-05',
'yyyy-mm-dd')  b,
       sysdate + 1 tomorrow, -- на 1 день больше
       sysdate - 1 yesterday-- на 1 день меньше
from dual
```

A	B	TOMORROW	YESTERDAY
4	.041666666666666666666666666666666666667	12-APR-20	10-APR-20

21.7.1. Months_between

Функция `months_between` возвращает разницу между датами в месяцах:

```
select months_between(
    to_date('2020-04-01', 'yyyy-mm-dd'),
    to_date('2020-02-01', 'yyyy-mm-dd')) months_diff_1,

    months_between(
    to_date('2020-04-01', 'yyyy-mm-dd'),
    to_date('2020-02-10', 'yyyy-mm-dd')) months_diff_2
from dual
```

MONTHS_DIFF_1	MONTHS_DIFF_2
2	1.709677419354838709677419354838709677 42

21.8. TIMESTAMP

Тип TIMESTAMP является расширением типа DATE. Он также, как и тип DATE, позволяет хранить год, месяц, день, часы, минуты и секунды. Но пимимо всего этого в TIMESTAMP можно хранить **доли секунды**.

TIMESTAMP - максимально точный тип данных для хранения даты, точнее в ORACLE уже нет.

При описании колонки с типом TIMESTAMP можно указать точность, с которой будут храниться доли секунды. Это может быть число от 0 до 9. По умолчанию это значение равно 6.

Пример создания таблицы с колонкой типа TIMESTAMP:

```
create table user_log(
  username varchar2(50 char) not null,
  login_time timestamp(8) not null,
  logout_time timestamp -- эквивалентно TIMESTAMP(6)
);
```

Колонка logout_time может хранить доли секунды с точностью до 6 знаков после запятой, а колонка login_time - с точностью до 8 знаков.

21.8.1. SYSTIMESTAMP

Данная функция работает так же, как и SYSDATE, только она возвращает текущую дату в формате TIMESTAMP:

```
select systimestamp
from dual
```

SYSTIMESTAMP
13-APR-20 07.13.09.688480 AM +00:00

21.9. EXTRACT

Функция extract позволяет извлечь из даты определенные составные части, например получить только год, или только месяц и т.п.

```
select extract (year from to_date('01.01.2020', 'dd.mm.yyyy')) year,
       extract (month from to_date('01.01.2020', 'dd.mm.yyyy')) month,
       extract (day from to_date('01.01.2020', 'dd.mm.yyyy')) day
from dual
```

YEAR	MONTH	DAY
2020	1	1

Извлекаемые части имеют числовой тип данных, т.е. колонки `year`, `month` и `day` всего лишь числа.

Несмотря на то, что тип `DATE` хранит также время вплоть до секунд, получить часы, минуты или секунды нельзя:

```
select extract (hour from to_date('01.01.2020 21:40:13', 'dd.mm.yyyy hh24:mi:ss'))
from dual
```

В ответ мы получим ошибку `ORA-30076: invalid extract field for extract source`.

Но если использовать тип `TIMESTAMP`, то помимо года, месяца и дня с помощью функции `EXTRACT` можно по отдельности получить значение часов, минут и секунд:

```
select extract(hour from systimestamp) hour,
       extract(minute from systimestamp) minute,
       extract(second from systimestamp) second
from dual
```

HOUR	MINUTE	SECOND
23	26	23.86374

21.9.1. Приведение строки к timestamp

Для приведения строки к типу `timestamp` используется фукнция `TO_TIMESTAMP`:

```
select TO_TIMESTAMP('2020-01-01 14:43:00.99', 'yyyy-mm-dd hh24:mi:ss.ff') d1,
       TO_TIMESTAMP('2020-01-01 14:43:00.997836765', 'yyyy-mm-dd hh24:mi:ss.ff9') d2
from dual
```

В запросе выше следует обратить внимание на то, как указывается точность долей секунды. `ff3` будет сохранять точность до тысячных долей секунды, `ff9` - до максимальных 9-и разрядов.

Форматы строк для приведения к датам очень разнообразны. Здесь приведены варианты, которые чаще всего понадобятся на практике. Ознакомиться со всеми форматами строк можно в документации.

22. Оператор INSERT

Как уже говорилось ранее, `INSERT` предназначен для вставки данных в таблицу. Существует

несколько вариантов его использования.

22.1. Вставка с указанием колонок

В таком варианте после указания таблицы в скобках перечисляются колонки, в которые будут записываться указываемые данные.

```
insert into employees(id, name, age)
values(1, 'John', 35)
```

Данный способ является предпочтительным, т.к. он более информативен - сразу видно, **что** за данные вставляются в таблицу.

Также, при использовании такого способа, можно изменять порядок перечисления данных для строки:

```
insert into employees(age, name, id)
values(30, 'Dave', 2)
```

22.2. Вставка без указания колонок

При таком варианте список столбцов таблицы не перечисляется, а сразу указываются значения, которые вставляются в таблицу:

```
insert into employees
values(1, 'John', 35)
```

Такой способ подразумевает, что мы будем перечислять значения для всех колонок в таблице, причем в том порядке, в котором они следуют в таблице.

Такой способ лучше не использовать, т.к. он:

- Неинформативен. Невозможно сказать, что значит 1, а что 35 без просмотра структуры таблицы
- Нестабилен к изменениям. Добавление/удаление колонки из таблицы потребует добавления/удаления значения из запроса.

Разберем последний минус данного подхода. Предположим, что в таблице `employees` колонка `age` необязательна, т.е. может содержать `NULL`.

В случае, когда мы указываем колонки, мы можем сделать так:

```
insert into employees(id, name)
values(1, 'John')
```

Если использовать вариант без указания колонок, то мы будем вынуждены прописывать значения для всех колонок:

```
insert into employees(id, name, age)
values(1, 'John', null)
```

Попытка указать всего 2 значения при вставке приведет к ошибке `ORA-00947: not enough values`:

```
-- выдаст ошибку ORA-00947: not enough values
insert into employees
values(1, 'John')
```

То же самое будет и в случае, если мы добавим в таблицу необазательную колонку - написанный ранее запрос с перечислением колонок будет работать, а запрос, в котором колонки не указывались - нет.

22.3. INSERT INTO ... SELECT

Данный способ **очень** мощная и гибкая возможность. Она позволяет использовать значения, возвращаемые оператором `select` в качестве значений для вставки.

В общем виде подобный запрос выглядит так:

```
insert into table_1(column_1, column_2, column_3...)
select col_1,
       col_2,
       col_3,
       ....
from table_2
```

Более конкретный пример может выглядеть так: предположим, что нас попросили записать в таблицу `emp_report` список сотрудников, которые старше 40 лет. Из этой таблицы потом экспортируют данные в отчет для руководства.

```
insert into emp_report(emp_id, name)
select emp.id,
       emp.name
from employees emp
where emp.age > 40
```

В случае, если запрос `select` не вернул никаких данных, то в таблицу также не будет вставлено ни одной строки

В следующем примере данные не будут добавлены в таблицу(предполагается, что в таблице

employees нет сотрудников с отрицательным возрастом):

```
insert into emp_report(emp_id, name)
select emp.id,
       emp.name
from employees emp
where emp.age < 0
```

Запрос, получающий данные, может быть достаточно сложным - в нем могут использоваться соединения таблиц, различные условия, подзапросы и т.п.

Предположим, нас попросили добавить также и количество детей у сотрудника. Список детей по сотрудникам хранится в таблице emp_childs. Тогда запрос вставки данных мог выглядеть следующим образом:

```
insert into emp_report(emp_id, name, childs_count)
select emp.id,
       emp.name,
       count(chlds.id) childs_count
from employees emp
join emp_childs chlds on chlds.emp_id = emp.id
where emp.age > 40
group by emp.id, emp.name
```

23. Оператор UPDATE

Данный оператор изменяет уже существующие данные в таблице.

Общий синтаксис выглядит следующим образом:

```
UPDATE table_1 t
SET t.column_1 = val_1,
    t.columm_2 = val_2
WHERE <условия для фильтрации строк>
```

При обновлении можно ссылаться на текущие значения в таблице. Например, увеличим возраст всех сотрудников на 1 год:

```
update employees emp
set emp.age = emp.age + 1
```

Можно добавлять любые условия в where, как и в select- запросах, чтобы обновить не все строки в таблице, а только те, которые удовлетворяют определенным условиям.

```
-- Увеличить возраст сотруднику с именем Антон Иванов
update employees emp
set emp.age = emp.age + 1
where emp.name = 'Антон Иванов';

-- Увеличить возраст сотруднику с id = 10 и привести имя к верхнему регистру
update employees emp
set emp.age = emp.age + 1,
    emp.name = upper(emp.name)
where emp.id = 10;
```

При обновлении мы можем использовать подзапросы для получения новых значений:

```
update employees emp
-- для каждого сотрудника получаем возраст из таблицы страховой карточки
-- и присваиваем это значение в колонку emp таблицы employees
set emp.age = (select age from insurance_card ic where ic.emp_id = emp.id)
where emp.age is null
```

В данном случае использовался коррелированный подзапрос, чтобы получить возраст сотрудника из его страховой карточки.

С использованием подзапросов можно обновлять сразу несколько колонок в таблице:

```
update employees emp
set(
    emp.age,
    emp.passport_no
) = (select ic.age, ic.passport_no from insurance_card ic
     where ic.emp_id = emp.id)
```

Подобное обновление сразу нескольких колонок работает только с подзапросами, вручную установить значения не получится:

```
-- Получим ошибку!
update employees emp
set(
    emp.age,
    emp.passport_no
) = (20, '324589')
```

В результате получим ошибку `ORA-01767: UPDATE ⋯ SET expression must be a subquery`.

Но зато можно вот так:

```
update employees emp
set(
    emp.age,
    emp.passport_no
) = (select 20, '324589' from dual)
```

Последний запрос более демонстрационный, если нужно обновить несколько колонок заранее известными константами, то лучше прибегнуть к "классическому" варианту обновления таблицы - так запрос будет проще читаться.

24. Оператор DELETE

Данный оператор предназначен для удаления строк из таблицы.

Рассмотрим, как он работает, на примерах. В самом простом варианте он используется для удаления всех данных из таблицы:

```
--- Удалить все данные из таблицы employees
delete
from employees
```

Для удаления строк, попадающих под определенный критерий, как и в случае с оператором `UPDATE`, прописываем эти условия в `WHERE` части:

```
-- Удалить данные по сотруднику с id = 20
delete
from employees emp
where emp.id = 20;

-- Удалить данные по сотрудникам старше 70 лет
delete
from employees emp
where emp.age > 70;
```

24.1. Удаление данных из связанных таблиц

Если в БД есть таблицы, которые содержат строки, ссылающиеся на удаляемые, Oracle выдаст ошибку `ORA-02292 Constraint violation - child records found`.

В таком случае нужно сначала удалить данные из всех дочерних таблиц, и только потом удалить данные из главной таблицы:

```
-- Сначала удаляем данные страховой карточки по сотруднику с id = 10
delete
from insurance_card ic
where ic.emp_id = 10;

-- И только после этого удаляем данные по самому сотруднику
delete
from employees emp
where emp.id = 10;
```

25. Команда MERGE

Команда `MERGE` позволяет выбрать строки из одного источника и использовать их для обновления, вставки или удаления строк в таблице.

Вся прелесть данной команды в том, чтов некоторых она позволяет сделать все эти операции в одном выражении SQL.

25.1. Подготовка данных

Будем использовать следующие таблицы:

```
create table employees(
    id number not null,
    emp_name varchar2(200 char) not null,
    department varchar2(100 char) not null,
    position varchar2(100 char) not null
);

create table ted_speakers(
    emp_id number not null,
    room varchar2(30 char),
    conf_date date not null
);

insert into employees
values(1, 'Иван Иванов', 'SALARY', 'MANAGER');

insert into employees
values(2, 'Елена Петрова', 'SALARY', 'CLERK');


insert into employees
values(3, 'Алексей Сидоров', 'IT', 'DEVELOPER');

insert into employees
values(4, 'Михаил Иванов', 'IT', 'DEVELOPER');

insert into employees
values(5, 'Владимир Петров', 'IT', 'QA');

insert into ted_speakers
values(1, '201b', to_date('2020.01.01', 'yyyy.mm.dd'));

insert into ted_speakers
values(3, '101', to_date('2020.01.01', 'yyyy.mm.dd'));

insert into ted_speakers
values(5, '201b', to_date('2020.01.01', 'yyyy.mm.dd'));
)
```

Эти таблицы - списки сотрудников компании и список сотрудников, выступающих на конверенции TED.

25.2. Использование MERGE

Задача: Как стало известно, все сотрудники из небольшого подразделения IT будут выступать на конференции, только в аудитории `809`.

В таблицу `ted_speakers` уже указаны некоторые сотрудники безопасности, только у них указаны другие. То есть, чтобы решить задачу, нам нужно сделать несколько действий:

1. Обновить номера аудиторий у уже существующих записей в таблице `ted_speakers`
2. Добавить туда недостающих сотрудников из отдела IT

Данную задачу можно очень просто решить, используя 2 уже известных оператора `INSERT` и `UPDATE`:

```
-- Сначала обновим аудиторию у тех сотрудников, которые уже записаны
-- в список выступающих, и которые работают в отделе безопасности
update ted_speakers ts
set ts.room = '809'
where ts.emp_id in (select emp_id
    from employees
    where department = 'IT'
);

/* Затем добавим в список выступающих
   недостающих сотрудников из IT подразделения
*/
insert into ted_speakers(emp_id, room, conf_date)
select id, '809', to_date('2020.01.03', 'yyyy.mm.dd')
from employees
-- нужны сотрудники из IT подразделения
where department = 'IT'
/*
   которых еще нет в таблице выступающих
   Для наших небольших таблиц подойдет
   полная выборка сотрудников из ted_speakers
*/
and id not in (
     select emp_id
     from ted_speakers);
```

А вот как эту же задачу можно решить с помощью оператора `MERGE`:

```
merge into ted_speakers ts -- (1)
using (
    select id
    from employees
    where department = 'IT') eit -- (2)
on (eit.id = ts.emp_id) -- (3)
when matched then -- (4) Если есть такой сотрудник
    update set ts.room = '809' -- обновляем аудиторию
when not matched then -- (5) Если нет такого сотрудника
    insert (emp_id, room, conf_date)
      values(eit.id, '809', to_date('2020.04.03', 'yyyy.mm.dd')) -- добавляем его
```

Смотрим на результат:

```
select ts.emp_id, ts.room, emp.emp_name, emp.department
from ted_speakers ts
join employees emp on emp.id = ts.emp_id
```

EMP_ID	ROOM	EMP_NAME	DEPARTMENT
1	201b	Иван Иванов	SALARY
3	809	Алексей Сидоров	IT
4	809	Михаил Иванов	IT
5	809	Владимир Петров	IT

Все сотрудники из IT отдела были добавлены, у всех аудитория равна 809.

Давайте подробнее разберем, как этот запрос работает.

Для начала, в строке `(1)` мы указываем, в какую таблицу мы будем производить слияние(кстати, слово `merge` с английского так и переводится).

После этого, мы пишем обычный запрос, который будет являться источником данных для наших действий и указываем его в `USING`. Этому подзапросу нужно дать псевдоним, чтобы можно было в дальнейшем обращаться к данным, которые он возвращает(строка `(2)`). В нашем случае источником данных является запрос, который получает список всех сотрудников из подразделения IT.

После определения источника данных мы указываем условие, по которому таблица слияния будет соединяться с источником данных(строка `(3)`). Мы соединяемся по id сотрудника.

Далее мы указываем, что будем делать, если в исходной таблице есть строки, для которых условие `(3)` выполняется, и что делать, если это условие не выполняется. Это происходит в частях запроса `(4)` и `(5)` соответственно.

Теперь мы вместо двух разных команд SQL использовали всего одну. Также, мы сократили количество обращений к БД с двух до одного.

25.2.1. Использование DELETE в MERGE

Кроме операций вставки и обновления в `MERGE` можно использовать и операцию удаления. Но есть одна особенность:

Операция DELETE в MERGE будет производиться только для тех строк, которые были обновлены командой `UPDATE`

Решим следующую задачу: удалить из списка выступающих всех сотрудников IT подразделения, кроме сотрудника с id = 5; для этого сотрудника нужно сдвинуть дату выступления на 1 месяц вперед.

Напишем следующий запрос:

```
merge into ted_speakers ts
using (
    select id
    from employees
    where department = 'IT') eit
on (ts.emp_id = eit.id)
when matched then
    -- изменим дату выступления для сотрудника с id = 5
    update set ts.conf_date = add_months(ts.conf_date, 1)
    where ts.emp_id = 5
    -- всех остальных сотрудников из it отдела удалим из ted_speakers
    delete
    where ts.emp_id <> 5
```

```
select ts.emp_id, ts.room, emp.emp_name, emp.department
from ted_speakers ts
join employees emp on emp.id = ts.emp_id
```

EMP_ID	CONF_DATE	EMP_NAME	DEPARTMENT
1	01-JAN-20	Иван Иванов	SALARY
3	01-JAN-20	Алексей Сидоров	IT
4	03-APR-20	Михаил Иванов	IT
5	01-FEB-20	Владимир Петров	IT

Как видно, мы дата выступления для сотрудника с `id = 5` была изменена, но остальные сотрудники из it подразделения остались в таблице.

Это произошло потому, что в команде `UPDATE` мы написали условие `where ts.emp_id = 5`. Это значит, что команда `delete` будет выполняться только для одной этой строки. А так как в условии удаления мы написали условие `where ts.emp_id <> 5`, то она ничего не удалит. Следующий код поможет понять, почему:

```
delete from ted_speakers
where emp_id = 5 and emp_id <> 5 -- это условие всегда ложно!
```

Зная, что `delete` выполняется только для обновленных строк, перепишем запрос:

```
merge into ted_speakers ts
using (
    select id
    from employees
    where department = 'IT') eit
on (ts.emp_id = eit.id)
when matched then
    update set ts.conf_date = add_months(ts.conf_date, 1)
    delete
    where ts.emp_id <> 5
```

EMP_ID	CONF_DATE	EMP_NAME	DEPARTMENT
1	01-JAN-20	Иван Иванов	SALARY
5	01-MAR-20	Владимир Петров	IT

Здесь дата 1 марта для сотрудника с `id = 5` была получена в результате последовательного выполнения запросов из учебника. В случае, если бы мы сразу запустили данный запрос, дата была бы на 1 месяц больше, чем было изначально - 1 февраля 2020 года.

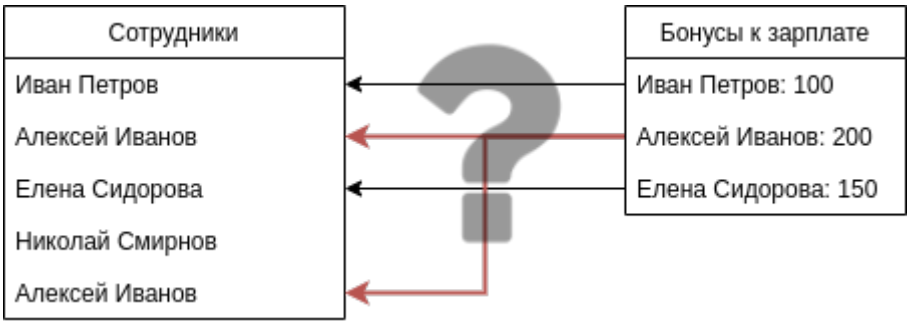
Теперь все работает, как нам нужно. Все потому, что в update мы не прописывали никаких условий, т.е он обновил месяц выступления для каждого сотрудника IT подразделения, выступающего на конференции. Далее, команда `DELETE` была запущена для каждой строки с выступающими из IT отдела. Вот уже в ней мы добавили условие `where ts.emp_id <> 5`. То есть она удалила всех сотрудников IT отдела за исключением сотрудника с ID = 5.

26. Первичные ключи

Рассмотрим следующую ситуацию: пусть у нас есть 2 таблицы. Первая содержит список сотрудников, вторая - размер бонусов к зарплате для какой-то части этих сотрудников.

В жизни есть определенная вероятность того, что двух разных людей могут звать одинаково. Так вышло и у нас - 2 абслютно разных сотрудника имеют одинаковое имя - **Алексей Иванов**.

Предположим, что мы хотим одному из них начислить бонус в размере 200$. Глядя на список сотрудников с бонусами, можем ли мы сказать, *какому именно Алексею Иванову мы должны начислить бонус в размере 200$*? Однозначный ответ - нет.



Каждый отдельно взятый сотрудник - отдельная сущность, и всегда нужно иметь возможность различать их между собой. Именно для этого и используются первичные ключи.

Первичный ключ - это такой атрибут, который позволяет однозначно идентифицировать отдельно взятую строку в таблице.

Исходя из этого можно выделить еще некоторые свойства первичного ключа:

1. Он не может быть пустым
2. Он уникален в пределах отдельно взятой таблицы
3. В таблице может быть только один первичный ключ

26.1. Добавление первичного ключа в таблицу

Добавить первичный ключ в таблицу можно несколькими способами. Первый - добавление при создании таблицы:

```
create table employees(
    id number primary key, -- Колонка id будет являться первичным ключом
    emp_name varchar2(100 char) not null,
    birth_date date not null
);
```

Данный способ - самый простой. Мы просто добавляем к нужной колонке `primary key`, и Oracle наделит ее всеми необходимыми свойствами.

Теперь давайте убедимся, что это действительно первичный ключ - попробуем добавить 2 строки в таблицу с одинаковым значением колонки `id`:

```
insert into employees(id, emp_name, birth_date)
values(1, 'Андрей Иванов', to_date('1984.12.04', 'yyyy.mm.dd'));

insert into employees(id, emp_name, birth_date)
values(1, 'Петр Иванов', to_date('1990.01.30', 'yyyy.mm.dd'));
```

Первая строка вставится без ошибок, но при попытке добавить еще одну с уже существующим `id` получим ошибку `ORA-00001: unique constraint (SQL_PXTWBEIMXHBUXOWCVTDQXEQKK.SYS_C0029851757) violated ORA-06512`. Эта ошибка говорит о том, что произошла попытка нарушить свойство уникальности нашего ключа. Длинная строка в скобках - это название нашего ключа. При создании его таким способом Oracle автоматически назначает каждому первичному ключу уникальное имя. В таких небольших примерах нам легко понять, где именно произошла ошибка, но в сложных системах с сотнями таблиц, с большим количеством запросов на вставку в БД понять, на каком ключе происходит сбой очень трудно.

К счастью, мы можем сами назначать имя для первичного ключа при создании таблицы:

```
create table employees(
    id number,
    emp_name varchar2(100 char) not null,
    birth_date date not null,
    constraint employees_PK primary key(id) -- создаем первичный ключ и назначаем ему
имя
)
```

Теперь попробуем вставить дублирующие значения в колонку `id`:

```
insert into employees(id, emp_name, birth_date)
values(1, 'Андрей Иванов', to_date('1984.12.04', 'yyyy.mm.dd'));

insert into employees(id, emp_name, birth_date)
values(1, 'Петр Иванов', to_date('1990.01.30', 'yyyy.mm.dd'));
```

На этот раз сообщение об ошибке будет немного другим: `ORA-00001: unique constraint (SQL_EAIYWBGLYOEYCEZDANCUIWUWH.EMPLOYEES_PK) violated`. Теперь мы явно видим, что ошибка в ключе `EMPLOYEES_PK`.

26.2. Составные первичные ключи

Первичный ключ может состоять из нескольких колонок. Подобный ключ обладает теми же особенностями, что и ключ из одной колонки.

Рассмотрим примеры создания таблицы с составным первичным ключом.

Предположим, что мы хотим начислять дополнительные бонусы сотрудникам каждый месяц. Одному сотруднику в месяц может быть начислено не более одного бонуса. Данные в этой таблице могли бы выглядеть вот так:

id сотрудника	месяц	Размер бонуса
1	2020.01.01	300
1	2020.02.01	150
2	2020.02.01	240
3	2020.02.01	100

Сделать колонку с id сотрудника первичным ключом нельзя, т.к. в таком случае в таблице можно будет иметь лишь по одной строке на каждого сотрудника. Но ключ из колонок с id сотрудника и месяца бонуса отлично подойдет - на один месяц можно будет давать бонус только одному сотруднику, в противном случае уникальность ключа будет нарушена.

```
create table month_bonuses(
    emp_id number not null,
    month_bonus date not null,
    bonus_value number not null,
    constraint month_bonuses_pk primary key(emp_id, month_bonus)
)
```

Указать `primary key` напротив нескольких колонок нельзя, т.к. Oracle будет пробовать каждую из этих колонок сдалеть первичным ключом, а он может быть только один. В итоге мы получим ошибку `ORA-02260: table can have only one primary key`:

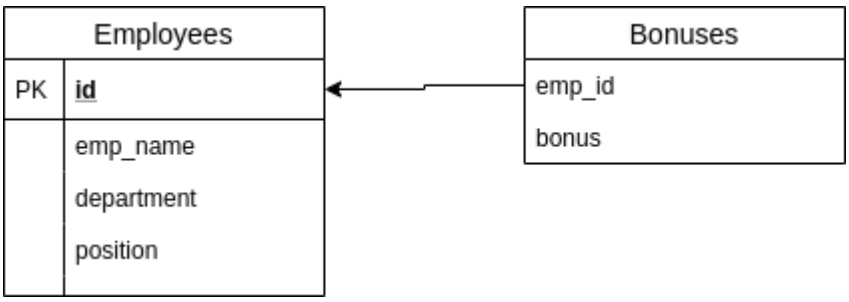
```
-- Получим ошибку при создании таблицы!
create table month_bonuses(
    emp_id number not null primary key,
    month_bonus date not null primary key,
    bonus_value number not null
)
```

27. Внешние ключи

Рассмотрим пример из части про первичные ключи.

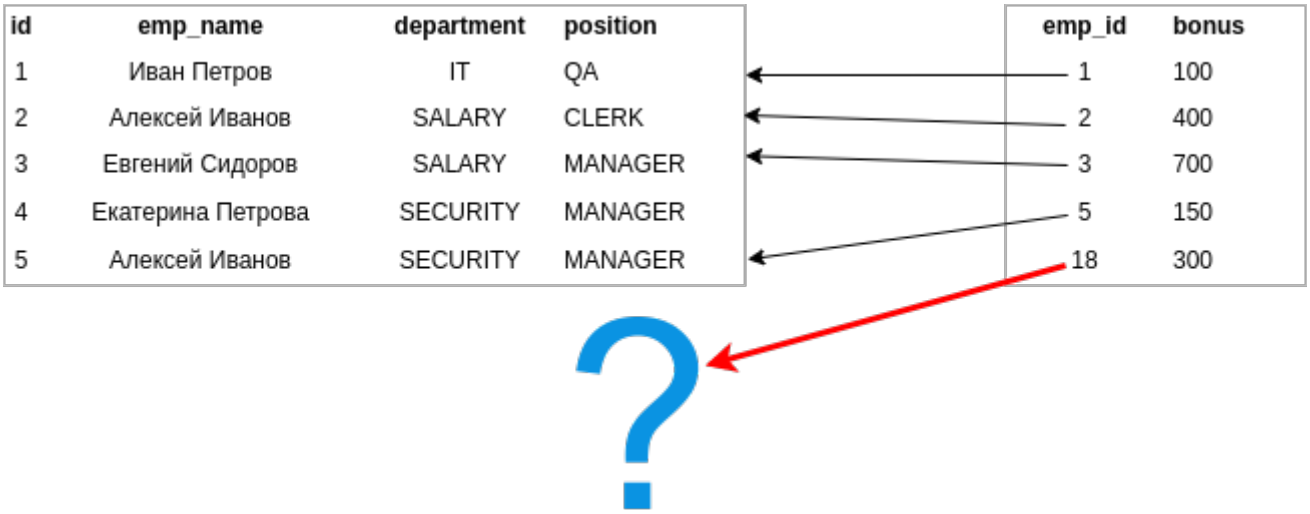
У нас было две таблицы - список сотрудников и единовременные бонусы для них. С помощью первичного ключа в таблице сотрудников мы решили проблему соотношения между бонусами и сотрудниками.

Схематично наши таблицы выглядят вот так:



Благодаря наличию первичного ключа мы однозначно можем сказать, какому сотруднику какой бонус начисляется.

А теперь посмотрим на следующую ситуацию: в таблицу `bonuses` добавляется запись со значением `emp_id`, которому нет соответствия в таблице сотрудников.



Как такое может быть? Мы начисляем бонусы сотруднику, которого у нас нет! Если нас попросят сказать, на какую сумму было выдано единовременных бонусов, или сколько их было выдано, то мы не сможем ответить, т.к. не будем уверены, что данные в таблице с бонусами вообще корректны.

Так вот, внешние ключи используются как для решения подобной проблемы.

Внешние ключи используются для того, чтобы указать, что данные в колонках одной таблицы могут содержать только определенные значения из другой таблицы.

В отличие от первичного ключа, значение внешнего не обязано быть уникальным. Более того, оно даже может содержать `NULL`. Главное требование - это наличие значения внешнего ключа в ссылаемой таблице.

27.1. Создание внешних ключей

Общий синтаксис следующий:

```
create table detail(
    master_id number,
    value_1 number,
    value_2 number,
    -- Внешний ключ из таблицы detail к таблице master
    constraint detail_master_id_fk
        foreign key(master_id)
        references master(id)
);
```

Здесь `detail_master_id_fk` - название внешнего ключа.

Также, как и у первичных ключей, длина имени внешнего ключа ограничена 30 символами.

У внешних ключей есть еще одна особенность - они могут ссылаться только на первичные или уникальные ключи. Если попытаться создать внешний ключ, который будет ссылаться на колонку, которая не является первичным или уникальным ключом БД выдаст ошибку.

Попробуем создать наши таблицы из примера:

```
create table employees(
    id number primary key,
    emp_name varchar2(100 char) not null,
    department varchar2(50 char) not null,
    positions varchar2(50 char) not null
);

create table bonuses(
    emp_id number not null,
    bonus number not null,
    constraint bonuses_emp_id_fk
        foreign key(emp_id)
        references employees(id)
);
```

В данном случае таблица `bonuses` является дочерней по отношению к таблице `employees`, т.к. содержит внешний ключ, который ссылается из `bonuses` на `employees`.

После этого заполним данными эти таблицы:

```
-- Сначала добавляем сотрудников

insert into employees(id, emp_name, department, position)
values(1, 'Иван Петров', 'IT', 'QA');

insert into employees(id, emp_name, department, position)
values(2, 'Алексей Иванов', 'SALARY', 'CLERK');

insert into employees(id, emp_name, department, position)
values(3, 'Евгений Сидоров', 'SALARY', 'MANAGER');

insert into employees(id, emp_name, department, position)
values(4, 'Екатерина Петрова', 'SECUTIRY', 'MANAGER');

-- После - бонусы для них

insert into bonuses(emp_id, bonus)
values(1, 100);

insert into bonuses(emp_id, bonus)
values(2, 400);

insert into bonuses(emp_id, bonus)
values(3, 700);
```

Порядок добавления данных в таблицы важен: нельзя сначала добавить новые данные в таблицу `bonuses`, а потом в таблицу `employees`, т.к. попытка добавить бонус для сотрудника, которого еще нет в таблице `employees` приведет к ошибке из-за наличия внешнего ключа.

```
-- Вот так будет ошибка, т.к. сотрудник с id = 5
-- еще не добавлен в таблицу employees
insert into bonuses(emp_id, bonus)
values(5, 500);

-- А вот так ошибки не будет
-- Сотрудник с id = 4 уже есть в таблице сотрудников
insert into bonuses(emp_id, bonus)
values(4, 500);
```

27.2. Составные внешние ключи

Подобно первичным ключам, внешние ключи также могут состоять из нескольких колонок.

Возьмем в качестве родительской таблицу `month_bonuses` из части про первичные ключи.

Создадим таблицу `month_bonus_reasons`, в которой будем хранить причины, по которым сотруднику перевели такой бонус к зарплате.

```
create table month_bonus_reasons(
    emp_id number not null,
    month date not null,
    reason varchar2(300 char) not null,
    constraint month_bonus_reasons_bonus_fk
        foreign key(emp_id, month)
        references month_bonuses(emp_id, month);


-- Один бонус сотруднику был выдан по нескольким причинам
insert into month_bonus_reasons
values(1, to_date('2020-01-01', 'yyyy-mm-dd'), 'Заключил договор с ООО "Рога и
Копыта"');

insert into month_bonus_reasons
values(1, to_date('2020-01-01', 'yyyy-mm-dd'), 'Победил в соревнованиях по дартсу
среди отдела');
);
```

28. Уникальные ключи

Возьмем нашу таблицу с сотрудниками и добавим туда колонку с номером паспорта сотрудника. Может ли у двух разных людей быть одинаковый номер паспорта? Однозначно нет. Если в наших данных возникнет такая ситуация, когда у нескольких сотрудников по ошибке указали один и тот же номер паспорта, это может обернуться серьезными ошибками - клиентская программа выдаст по поиску несколько записей вместо одной, либо вообще выдаст ошибку и закроется. Или в бухгалтерии переведут деньги не тому сотруднику, или наоборот, всем.

В любом случае, подобной ситуации нужно избежать. Это помогут сделать уникальные ключи.

На колонки с уникальными ключами, как и на колонки с первичными ключами, можно ссылаться из других таблиц по внешним ключам.

В отличие от первичных, в одной таблице может быть несколько уникальных ключей.

28.1. Создание уникальных ключей

```
create table employees(
    id number primary key,
    emp_name varchar2(200 char) not null,
    pas_no varchar2(30),
    constraint employees_pas_no_uk unique(pas_no)
)
```

Теперь попробуем добавить нескольких сотрудников с одинаковыми номерами паспортов:

```
-- Эта строка добавляется в таблицу без проблем
insert into employees(id, emp_name, pas_no)
values (1, 'Евгений Петров', '01012020pb8007');

-- А вот эту уже добавить нельзя - уникальный ключ
-- в таблице будет нарушен
insert into employees(id, emp_name, pas_no)
values (2, 'Алексей Иванов', '01012020pb8007');
```

Уникальные ключи на строковых данных чувствительны к регистру.

```
-- Эта строка добавляется без проблем
insert into employees(id, emp_name, pas_no)
values (2, 'Алексей Иванов', '01012020PB8007');
```

Пробелы вначале и конце строк также учитываются, поэтому следующие данные также успешно добавятся в таблицу:

```
insert into employees(id, emp_name, pas_no)
values (3, 'Петр Иванов', '   01012020PB8007');

insert into employees(id, emp_name, pas_no)
values (4, 'Иван Петров', '01012020PB8007  ');

insert into employees(id, emp_name, pas_no)
values (5, 'Светлана Сидорова', '  01012020PB8007  ');
```

Наличие подобных данных в таблице также ошибка - как ни крути, номер паспорта у всех этих сотрудников все равно совпадает. Поэтому в подобных случаях, когда регистр строк и наличие пробелов в начале или конце строки не должны учитываться, строки хранят в верхнем или нижнем регистре, а пробелы обрезают перед вставкой.

Т.е. вставка данных в таблицу выглядит подобным образом:

```
-- Сначала удаляем пробелы(TRIM), потом приводим к верхнему
-- регистру(UPPER)
insert into employees(id, emp_name, pas_no)
values (6, 'Светлана Сидорова', UPPER(TRIM('  01012020PB8007  ')));
```

Значения в колонке с уникальным ключом могут содержать `NULL`, причем строк с пустыми значениями может сколько угодно.

Следующий запрос выполнится без ошибок, и добавит 2 сотрудника с пустыми номерами

паспортов:

```
insert into employees(id, emp_name, pas_no)
values (7, 'Иван Иванов', NULL);

insert into employees(id, emp_name, pas_no)
values (8, 'Петр Петров', NULL);
```

Следует отметить, что это сработает только в том случае, если NULL-значения разрешены в колонке, как в нашем случае. Если бы колонка была NOT NULL, то в таком случае, конечно, пустые значения туда не положишь.

28.1.1. Составные уникальные ключи

Создадим таблицу месячных бонусов сотрудников с использованием уникального ключа, а не первичного:

```
create table bonuses(
    emp_id number,
    mnth date,
    bonus number,
    constraint bonuses_uk unique(emp_id, mnth)
);
```

Также, как и с первичным, вставить 2 строки с одинаковыми значениями не получится:

```
insert into bonuses(emp_id, mnth, bonus)
values(1, to_date('2020.01.01', 'yyyy.mm.dd'), 100);

-- Будет нарушена уникальность ключа bonuses_uk
insert into bonuses(emp_id, mnth, bonus)
values(1, to_date('2020.01.01', 'yyyy.mm.dd'), 200);
```

Но т.к. в уникальном ключе разрешены NULL-значения (и они разрешены в нашей таблице), следующие строки добавятся без проблем:

```
insert into bonuses(emp_id, mnth, bonus)
values(2, to_date('2020.01.01', 'yyyy.mm.dd'), 200);

insert into bonuses(emp_id, mnth, bonus)
values(null, null, 200);

insert into bonuses(emp_id, mnth, bonus)
values(null, to_date('2020.01.01', 'yyyy.mm.dd'), 200);
```

Более того, в случае, когда все колонки уникального ключа пусты, добавлять строк можно

сколько угодно(при условии, что не будет нарушена целостность других существующих в таблице ключей):

```
insert into bonuses(emp_id, mnth, bonus)
values(null, null, 200);

insert into bonuses(emp_id, mnth, bonus)
values(null, null, 200);

insert into bonuses(emp_id, mnth, bonus)
values(null, null, 200);
```

29. Представления

Представления(Views) - это такой объект в БД, который:

1. Выглядит как таблица
2. Внутри себя содержит SQL запрос, которым заменяется таблица при обращении к ней.

Во многом представления работают также, как и обычные таблицы. В них можно(правда с определенными ограничениями) вставлять, изменять и удалять данные.

29.1. Создание представлений

Общий синтаксис создания представления следующий:

```
create view viewname as
select ...
....
....;
```

Т.е. для создания представления достаточно написать запрос, который возвращать нужные данные.

Можно создавать представления с опцией `or replace`, тогда в том случае, если такое представление уже существует, оно будет заменено на новое.

```
create or replace view viewname as
select ...
....
...;
```

Создадим таблицу с сотрудниками, должностями и подразделениями:

```
create table employees(
    id number,
    emp_name varchar2(100 char),
    dept_id number,
    position_id number
);

create table departments(
    id number,
    dept_name varchar2(100)
);

create table positions(
    id number,
    position_name varchar2(100)
);

insert into departments values(1, 'IT');
insert into departments values(2, 'SALARY');

insert into positions values(1, 'MANAGER');
insert into positions values(2, 'CLERK');

insert into employees values(1, 'Иван Петров', 1, 1);
insert into employees values(2, 'Петр Иванов', 1, 2);
insert into employees values(3, 'Елизавета Сидорова', 2, 1);
insert into employees values(4, 'Алексей Иванов', 2, 2);
```

Создадим представление `vemployees`, которое будет выводить данные по сотрудникам в уже "соединенном" виде:

```
create view vemployees as
select e.id,
       e.emp_name,
       d.dept_name,
       p.position_name
from employees e
join departments d on d.id = e.dept_id
join positions p on p.id = e.position_id;

comment on table vemployees is 'сотрудники';
comment on column vemployees.id is 'id сотрудника';
comment on column vemployees.emp_name is 'имя сотрудника';
comment on column vemployees.dept_name is 'подразделение';
comment on column vemployees.position_name is 'должность';
```

Следует обратить внимание на то, что представлениям и колонкам в них можно задавать комментарии как и обычным таблицам.

Теперь, чтобы получить нужные нам данные, нам не нужно заново писать запрос, достаточно сразу выбрать данные из представления:

```
select *
from vemployees
```

ID	EMP_NAME	DEPT_NAME	POSITION_NAME
1	Иван Петров	IT	MANAGER
3	Елизавета Сидорова	SALARY	MANAGER
2	Петр Иванов	IT	CLERK
4	Алексей Иванов	SALARY	CLERK

При создании представлений можно использовать уже существующие представления:

```
create view vemployees_it as
select a.*
from vemployees a
where a.dept_name = 'IT';
```

Следует с осторожностью использовать уже созданные представления при создании других представлений. Может случиться так, что написать новый запрос будет куда лучше, чем использовать существующие, но не полностью подходящие.

29.1.1. Символ * при создании представлений

Когда при создании представления используется символ *, то Oracle заменяет звездочку на список столбцов. Это означает, что если в таблицу будет добавлена новая колонка, то она не будет автоматически добавлена в представление.

Это очень просто проверить:

```
create table tst(
    n1 number,
    n2 number
);

insert into tst values(1, 2);

create view v_tst as
select *
from tst;
```

Посмотрим, какие данные содержатся в представлении:

```
select *
from v_tst
```

N1	N2
1	2

Теперь добавим в таблицу `tst` еще одну колонку(изменение таблиц будет рассматриваться позже, сейчас достаточно понимать, что данный запрос добавляет новую колонку в таблицу):

```
alter table tst
add (n3 number);
```

Если сейчас получить все данные из представления, мы увидим, что список колонок в ней не изменился:

N1	N2
1	2

Чтобы добавить колонку `n3` в представление, можно изменить его, добавив в список колонок нужную, либо заново создать(с использованием `create or replace`):

```
create or replace view v_tst as
select *
from tst
```

29.2. Изменение данных представления

Таблицы, которые используются в запросе представления, называются **базовыми таблицами**.

Представления, которые созданы на основании одной базовой таблицы, можно изменять также, как и обычную таблицу.

Например, создадим представление `vdepartments` и добавим в него несколько записей.

```
-- создаем представление
create view vdepartments as
select id, dept_name
from departments;

-- добавляем данные через представление, а не таблицу
insert into vdepartments(id, dept_name)
values(10, 'SALES');
```

Конечно, фактически данные добавляются не в представление, а в базовую таблицу(в данном случае `departments`):

```
select *
from departments
```

ID	DEPT_NAME
10	SALES
1	IT
2	SALARY

Строки можно и удалять, а также и изменять:

```
delete from vdepartments
where id = 10;

update vdepartments
set dept_name = 'SECURITY'
where id = 1;
```

Посмотрим на результаты:

```
select *
from vdepartments
```

ID	DEPT_NAME
1	SECURITY
2	SALARY

29.2.1. Представления с проверкой (WITH CHECK OPTION)

Можно создавать представления, которые будут ограничивать изменение данных в базовых таблицах. Для этого используется опция `WITH CHECK OPTION` при создании представления.

Создадим представление, которое содежит в себе только менеджеров:

```
create view vemp_managers as
select *
from employees
where position_id = 1;
```

Данное представление содержит только менеджеров, но это не означает, что в него нельзя

добавить сотрудников других профессий:

```
-- Добавим сотрудника с position_id = 2
insert into vemp_managers(id, emp_name, dept_id, position_id)
values(10, 'Иван Иванов', 1, 2);
```

Данные в представлении остались те же, что и были:

```
select *
from vemp_managers
```

ID	EMP_NAME	DEPT_ID	POSITION_ID
1	Иван Петров	1	1
3	Елизавета Сидорова	2	1

А вот в таблицу `employees` был добавлен новый сотрудник Иван Иванов:

```
select *
from employees
where id = 10
```

ID	EMP_NAME	DEPT_ID	POSITION_ID
10	Иван Иванов	1	2

Для того, чтобы через представление можно было изменять только те данные, которые в нем содержатся(а точнее, которые можно получить через представление), при его создании следует указать опцию `WITH CHECK OPTION`.

Создадим заново представление `vemp_managers`, только с добавлением `with check option`, и попробуем снова добавить в него запись:

```
create or replace view vemp_managers as
select *
from employees
where position_id = 1
with check option;

-- Попробуем добавить запись с position_id = 2
insert into vemp_managers(id, emp_name, dept_id, position_id)
values(11, 'Иван Иванов Второй', 1, 2);
```

При попытке это сделать, мы получим ошибку `view WITH CHECK OPTION where-clause violation`.

Но зато добавить сотрудника с `position_id = 1` можно без проблем:

```
-- Запись успешно добавится в таблицу employees
insert into vemp_managers(id, emp_name, dept_id, position_id)
values(11, 'Иван Иванов Второй', 1, 1);
```

29.2.2. Изменение представлений из нескольких таблиц

В Oracle можно изменять данные через представления, которые получают данные из нескольких таблиц.

Но есть определенные ограничения:

1. Изменять можно данные только одной базовой таблицы
2. Изменяемая таблица должна быть т.н. **key preserved table** (таблица с сохранением ключа).

Второй пункт возможно самый важный для понимания того, можно ли изменять данные в представлении из нескольких таблиц или нет.

Так вот, таблица называется key preserved, если каждой ее строке соответствует **максимум одна строка** в представлении.

Следует помнить, что свойство сохранения ключа в представлениях не зависит от данных, а скорее от структуры таблиц и их отношений между собой. Фактически в представлении данные могут выглядеть так, что для одной строки базовой таблицы есть лишь одна строка представления, но это не означает, что этот вид не изменится при изменении данных в таблицах представления.

Для примера создадим представление `vemp_depts`, которое будет содержать информацию о сотрудниках и подразделениях, в которых они работают:

```
create or replace view vemp_depts as
select e.id,
       e.emp_name,
       e.dept_id,
       e.position_id,
       d.id department_id,
       d.dept_name
from employees e
join departments d on e.dept_id = d.id
```

Посмотрим, какие данные там находятся:

```
select *
from vemp_depts
```

ID	EMP_NAME	DEPT_ID	POSITION_ID	DEPARTMENT_ID	DEPT_NAME
1	Иван Петров	1	1	1	IT
2	Петр Иванов	1	2	1	IT
3	Елизавета Сидорова	2	1	2	SALARY
4	Алексей Иванов	2	2	2	SALARY

Как мы видим, каждая строка из базовой таблицы `employees` встречается в представлении всего один раз. Попробуем добавить нового сотрудника через это представление:

```
insert into vemp_depts(id, emp_name, dept_id, position_id)
values(20, 'Иван Василенко', 1, 1);
```

В результате получаем ошибку `cannot modify a column which maps to a non key-preserved table`, которая говорит о том, что таблица не обладает нужными свойствами для обновления через представление.

Зная, что проблему нужно искать не в самих данных, а в схеме БД, посмотрим, как мы создавали наши таблицы и как выглядит наш запрос в представлении.

```
select e.id,
       e.emp_name,
       e.dept_id,
       e.position_id,
       d.id department_id,
d.dept_name
from employees e
join departments d on e.dept_id = d.id
```

Здесь мы берем каждую строку из таблицы `employees` и соединяем с таблицей `departments` по полю `dept_id`. В каком случае может произойти так, что в представлении для одной строки из таблицы `employees` окажутся 2 строки после соединения с таблицей `departments`? Правильно, в том случае, если в таблице `departments` будут 2 строки с одинаковым значением в колонке `id`. Сейчас таких данных в таблице нет, но это не означает, что они не могут появиться. Посмотрим, как мы создавали таблицу `departments`:

```
create table departments(
    id number,
    dept_name varchar2(100)
);
```

Как видно, нет никаких ограничений на колонку `id`. Но мы можем сделать ее уникальной, добавив первичный или уникальный ключ.

```
alter table departments
add (
    constraint departments_pk primary key(id)
);
```

Теперь снова попробуем добавить нового сотрудника:

```
-- Запись будет добавлена без ошибок
insert into vemp_depts(id, emp_name, dept_id, position_id)
values(20, 'Иван Василенко', 1, 1);
```

Добавить данные в таблицу `departments` через это представление не получится:

```
--  cannot modify a column which maps to a non key-preserved table
insert into vemp_depts(department_id, dept_name)
values(7, 'HEAD DEPARTMENT');
```

Причина здесь та же: нельзя гарантировать, что в таблице `employees` каждый сотрудник имеет уникальное значение `dept_id`.

29.2.3. Ограничения в изменяемых представлениях

Изменения в представлениях возможны не всегда. Есть определенные условия, при которых они запрещены:

1. Наличие в представлении агрегатных функций, конструкции `group by`, оператора `distinct`, операторов для работы с множествами(`union`, `union all`, `minus`).
2. Если данные не будут удовлетворять условию, прописанному в опции `WITH CHECK OPTION`.
3. Если колонка в базовой таблице `NOT NULL`, не имеет значения по-умолчанию, и отсутствует в представлении.
4. Если колонки в представлении представляют собой выражения (Например что-то вроде `nvl(a.value, -1)`).

29.2.4. Запрет изменения представления

Чтобы создать представление, которое нельзя будет изменять, нужно создать его с опцией `with read only`.

Пересоздадим представление `vdepartments` и попробуем добавить туда данные:

```
create or replace view vdepartments as
select id, dept_name
from departments
with read only;

-- Попробуем добавить данные
insert into vdepartments(id, dept_name)
values(11, 'SECURITY');
```

В результате получим ошибку `cannot perform a DML operation on a read-only view`.

30. Изменение структуры таблицы

Уже созданные таблицы можно изменять. Для этого используется команда SQL `ALTER`. Данная команда относится к группе `DDL`.

30.1. Подготовка данных

Тестировать будем на таблице `employees`. Изначально она будет состоять только из одной колонки `id`:

```
create table employees(
    id number not null primary key
);

insert into employees(id)
values(1);

insert into employees(id)
values(2);

insert into employees(id)
values(3);

insert into employees(id)
values(4);
```

30.2. Добавление колонки в таблицу

Добавим в таблицу сотрудников колонку для хранения дня рождения:

```
alter table employees
add (birthday date)
```

По умолчанию все строки таблицы будут иметь `null` в новой колонке. Но если при ее добавлении указать значение по-умолчанию, то все строки будут содержать его в новой колонке.

Добавим колонку `notify_by_email`, которая будет по-умолчанию содержать в себе `1`, если сотруднику нужно отправлять уведомления по почте, и `0`, если нет:

```
alter table employees
add (
    notify_by_email number default 0
);

comment on column employees.notify_by_email is
'Уведомлять по почте(1-да, 0-нет)';
```

Посмотрим, как сейчас выглядят данные в таблице:

ID	BIRTHDAY	NOTIFY_BY_EMAIL
1	-	0
2	-	0
3	-	0
4	-	0

Как видно, каждая строка содержит `0` в колонке `notify_by_email`.

Нельзя добавить колонку `NOT NULL` в таблицу с данными без значения по-умолчанию.

```
-- Ошибка! Нельзя добавить колонку
-- без default-значения
alter table employees
add(
    not_null_col number(1) not null
)
```

В результате получим ошибку `ORA-01758: table must be empty to add mandatory (NOT NULL) column`.

Но если указать значение по-умолчанию, ошибки не будет:

```
-- Ошибки не будет, каждая строка будет
-- содержать 1 в колонке
alter table employees
add(
    not_null_col number(1) default 1 not null
)
```

Колонка добавляется без ошибок:

ID	BIRTHDAY	NOTIFY_BY_EMAIL	NOT_NULL_COL
1	-	0	1
2	-	0	1
3	-	0	1
4	-	0	1

30.2.1. Добавление нескольких колонок в таблицу

Чтобы добавить несколько колонок в таблицу, нужно просто перечислить их через запятую:

```
alter table employees
add ( emp_lastname varchar2(100 char),
      emp_firstname varchar2(100 char),
      dept_id number(2) default 10 not null,
      is_out varchar2(1) default 'Y' not null);

comment on column employees.emp_lastname is
'Фамилия';

comment on column employees.emp_firstname is
'Имя';

comment on column employees.dept_id is
'id подразделения';

comment on column employees.is_out is
'Больше не работает?';
```

30.3. Удаление колонки из таблицы

Удалим только что добавленную колонку `emp_lastname` из таблицы:

```
alter table employees
drop column emp_lastname
```

Следует учитывать, что если на удаляемую колонку ссылаются строки из другой

таблицы(посредством внешнего ключа), то удалить колонку не получится.

Убедимся в этом, создав таблицу `emp_bonuses`, которая будет ссылаться на колонку `id` в таблице `employees`:

```
create table emp_bonuses(
    emp_id number not null,
    bonus number not null,
    constraint emp_bonuses_emp_fk
        foreign key(emp_id) references employees(id)
)
```

Теперь попробуем удалить колонку `id`:

```
alter table employees
drop column id
```

В результате мы получим ошибку `ORA-12992: cannot drop parent key column`, которая говорит о том, что удаляемая колонка является родительской для другой таблицы.

30.3.1. Удаление нескольких колонок в таблице

Удалим колонки `emp_firstname' и `is_out` из таблицы:

```
alter table employees
drop (emp_firstname, is_out)
```

Удалять все колонки из таблицы нельзя, получим ошибку `ORA-12983: cannot drop all columns in a table`.

30.4. Переименование колонки

Переименуем колонку `birthday` в `bd`:

```
alter table employees
rename column bitrhday to bd
```

30.5. Изменение типа данных колонки

Изменим тип колонки `dept_id` с числового на строковый:

```
alter table employees
modify(
    dept_id varchar2(10)
)
```

Здесь нужно обратить внимание на то, что при изменении типа мы не добавляли ключевое слово `NOT NULL`. В `MODIFY` мы должны указать действия, которые действительно что-то изменят. Колонка `dept_id` и так была `not null`, и при изменении типа это свойство не нужно указывать.

Если попробовать добавить `not null`, получим ошибку `ORA-01442: column to be modified to NOT NULL is already NOT NULL`:

```
alter table employees
modify(
    dept_id varchar2(10) not null -- получим ошибку
)
```

Следует учитывать одну важную деталь при изменении типа данных - **изменяемая колонка должна быть пуста**.

Рассмотрим более подробно процесс изменения типа колонки, если в ней уже содержатся данные.

Спустя какое-то время мы решили, что не хотим использовать числовое поле для `boolean` значений. Вместо этого было решено использовать более понятный строковый тип.

Итак, для начала добавим колонку с нужным нам типом данных. Так как мы не можем назвать ее `notify_by_email`(такая уже есть на данный момент), то назовем ее `notify_by_email_new`:

```
alter table employees
add(
    notify_by_email_new varchar2(1)
     default 'N' not null
)
```

После этого нужно заполнить эту колонку данными. Алгоритм прост - значение `1` в колонке `notify_by_email` должно быть перенесено как значение `Y` в колонку `notify_by_email_new`, а значение `0` нужно перенести в виде 'N`. Так как при добавлении колонки мы указали значение по-умолчанию, то в таблице каждая строка содержит значение `N` в этой колонке. Все, что осталось - это изменить значение на `Y`, где `notify_by_email` равен 1:

```
update employees e
set e.notify_by_email_new = 'Y'
where e.notify_by_email = 1
```

Затем удаляем колонку `notify_by_email`:

```
alter table employees
drop column notify_by_email
```

Теперь можно переименовать `notify_by_email_new` в `notify_by_email`:

```
alter table employees
rename column notify_by_email_new to
notify_by_email
```

Смотрим на результат:

ID	BIRTHDAY	NOT_NULL_COL	DEPT_ID	NOTIFY_BY_EMAIL
1	-	1	-	N
2	-	1	-	N
3	-	1	-	N
4	-	1	-	N

30.6. Изменение атрибута NOT NULL в колонке

Сделаем так, чтобы в колонку `dept_id` можно было сохранять `null`:

```
alter table employees
modify(dept_id null);
```

А теперь снова сделаем ее `NOT NULL`:

```
alter table employees
modify(dept_id not null);
```

Нельзя изменить колонку на `NOT NULL`, если в ней уже содержатся `NULL`-значения.

30.7. Переименование таблицы

Следующий запрос переименует таблицу `employees` в `emps`:

```
rename employees to emps
```

Стоит отметить, что переименование таблицы не приведет к ошибке при наличии ссылок на нее. В нашем примере таблица успешно переименуется, несмотря на дочернюю таблицу `emp_bonuses`. Внешний ключ при этом никуда не девается, в таблицу `emp_bonuses` по-прежнему нельзя добавить значения, нарушающие условия внешнего ключа.