

Учебник SQL

в БД Oracle

Table of Contents

Главная	1.1
---------	-----

Учебник SQL

Введение	2.1
Введение в SQL	2.1.1
DML, DDL	2.1.2
Выполнение SQL. Облачные сервисы	2.1.3
Инструменты для работы с БД Oracle	2.1.4
Ссылки на полезные ресурсы	2.1.5
Основы	3.1
Таблицы	3.1.1
Основные типы данных	3.1.2
Пример SELECT запроса	3.1.3
Написание SQL- кода	3.1.4
Сортировка результатов. Order by	3.1.5
Оператор WHERE. Операторы сравнения	3.1.6
Проверка нескольких условий. AND, OR	3.1.7
Проверка значения на NULL	3.1.8
IN, NOT IN	3.1.9
Вхождение в диапазон. BETWEEN. NOT BETWEEN	3.1.10
Соединения таблиц	3.1.11
Древовидные структуры данных. Рекурсивные запросы	3.1.12
Подзапросы в Oracle	3.1.13
Exists. Наличие строк в подзапросе	3.1.14
Subquery factoring. WITH	3.1.15
Работа с множествами	4.1
Объединение. UNION	4.1.1
Разница запросов. MINUS	4.1.2
Пересечение запросов	4.1.3
Работа с множествами. Общая информация	4.1.4
Стандартные функции	5.1
Функции для работы со строками	5.1.1

Функции для работы с NULL	5.1.2
Условные функции	5.1.3
Битовые операции	5.1.4
Агрегирующие функции	5.1.5
Работа с датами в Oracle	5.1.6
Аналитические функции	5.1.7
Distinct. Удаление дубликатов	5.1.8
DML	6.1
Оператор INSERT	6.1.1
Изменение данных. UPDATE	6.1.2
Удаление данных. DELETE	6.1.3
Слияние данных. MERGE	6.1.4
Изменение структуры таблицы. ALTER TABLE	6.1.5
Объекты БД	7.1
Первичные ключи	7.1.1
Внешние ключи	7.1.2
Уникальные ключи	7.1.3
Представления	7.1.4
Индексы	7.1.5
Виртуальные колонки	7.1.6
Псевдостолбцы в Oracle	7.1.7
Транзакции	8.1
Транзакции в Oracle	8.1.1

Учебник PL/SQL

Введение	9.1
Что такое PL/SQL	9.1.1
Когда использовать PL/SQL	9.1.2
Основы	10.1
Переменные, константы. Простые типы данных	10.1.1
Функции в PL/SQL	10.1.2
Схема БД. Её объекты	10.1.3
Циклы в PL/SQL	10.1.4
Вложенные и именованные блоки	10.1.5
Первая программа на PL/SQL	10.1.6

Условное ветвление. If...else...elsif	10.1.7
Взаимодействие PL/SQL и SQL. Переключение контекста	10.1.8
DBMS_OUTPUT.PUT_LINE. Вывод на экран	10.1.9
Анонимные блоки	10.1.10
Пакеты	10.1.11
Процедуры в PL/SQL	10.1.12
Обработка ошибок	11.1
Обработка ошибок в PL/SQL	11.1.1
Взаимодействие с данными	12.1
Взаимодействие SQL и PL/SQL	12.1.1

Главная

Добро пожаловать на сайт учебника по языкам SQL и PL/SQL в БД Oracle.

Учебник по SQL, за исключением нескольких частей, может подойти и для изучения SQL без привязки к какой-либо базе данных.

Введение

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

Диалекты SQL

Реляционных 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**.

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.

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.

Команды языка SQL можно разбить на две группы - DML и DDL.

Кроме DML и DDL существуют еще команды DCL и TCL. На текущий момент они не рассматриваются в этом учебнике.

DML расшифровывается как Data Manipulation Language (Язык манипулирования данными). В него входят те команды SQL, которые могут изменять уже имеющиеся данные в БД. Под изменением следует понимать также добавление новой информации в БД и удаление уже существующей.

К командам DML относятся:

- `SELECT`
- `INSERT`
- `UPDATE`
- `DELETE`
- `MERGE`

Интересный момент - команда `SELECT` не изменяет данные, а только получает, но она все равно относится к категории DML.

DDL расшифровывается как Data Definition Language (Язык определения данных). В него входят те команды, которые отвечают за создание или изменение структуры данных или новых объектов в БД.

К DDL командам языка SQL относятся:

- `CREATE`
- `RENAME`
- `ALTER`
- `DROP`
- `RENAME`
- `TRUNCATE`
- `COMMENT`

Более подробно большая часть этих команд будет рассмотрена далее в этом учебнике.

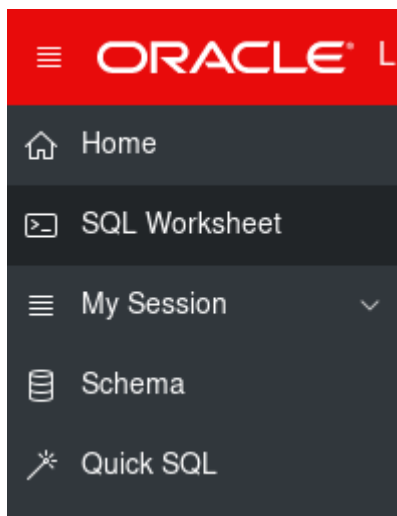
Для того, чтобы начать работу с БД(причем любой), она должна быть где-либо установлена, и к ней должен быть доступ на подключение и выполнение запросов.

LiveSQL

В этом учебнике 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":

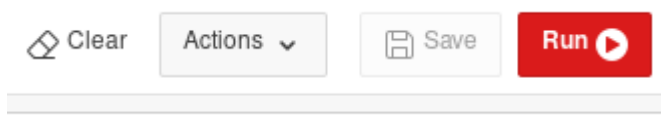


В открывшемся окне вводим наши SQL-запросы:

SQL Worksheet

```
1 create table test(  
2     id number(10) not null primary key,  
3     value varchar2(100)
```

Чтобы выполнить запрос, написанный в SQL Worksheet, нажимаем на 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://youtu.be/4oxsxJQQC-s>.

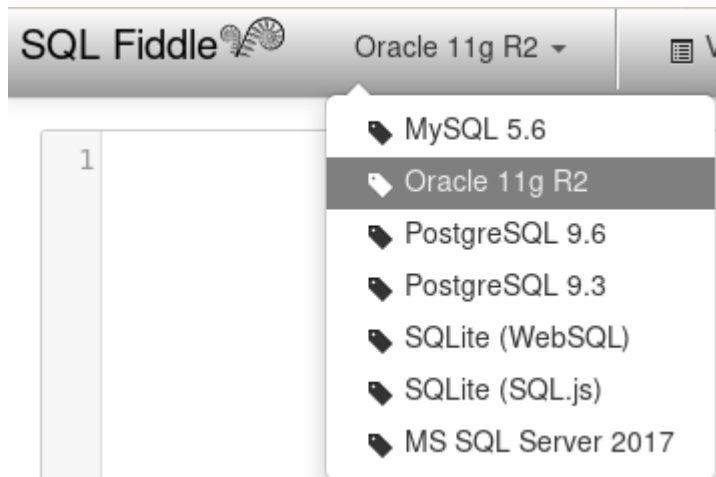
SQL Fiddle

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

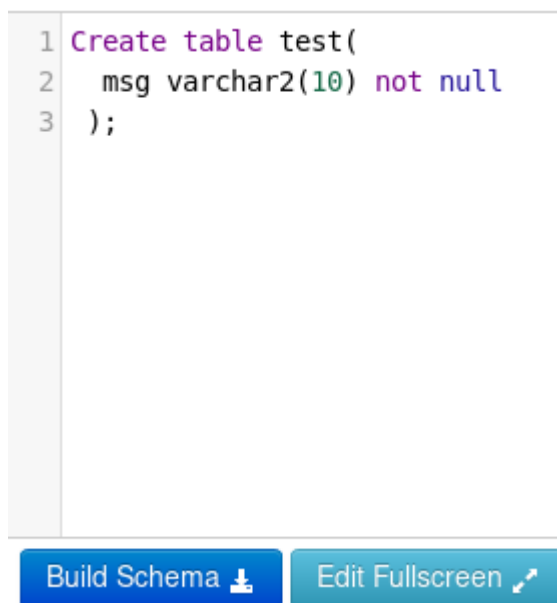
Далее будет описано, как работать с данным сервисом.

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":



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":

```
1 select *
2 from dual
```

Run SQL ▶ Edit Fullscreen ↗ [;] ▾

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

```
DUMMY
X
```

Запуск примеров учебника

Запускать примеры из учебника можно в любой среде. Тем не менее, в силу того, что тема транзакций будет рассматриваться в самом конце, лучше всего(и удобнее) использовать сервис LiveSQL.

В дальнейшем, при изучении PL/SQL, придется выбрать какую-нибудь IDE, но при изучении базового SQL это необязательно.

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

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.

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.

Примечание: Все приведённые ниже ресурсы - англоязычные.

Youtube

- [Practically Perfect PL/SQL](#)
- [The Magic of SQL](#)
- [Connor McDonald](#)
- [Oracle Developers](#)

Блоги/сайты

- [Ask Tom](#)
- <https://connor-mcdonald.com/>
- <https://stevenfeuersteinonplsql.blogspot.com/>
- <https://oracle-base.com>
- <https://blogs.oracle.com/sql/>
- <http://www.dba-oracle.com>

Книги

Книг написано достаточно много, и нет смысла их перечислять. Единственное, что можно порекомендовать — по возможности читайте англоязычные книги, так как русскоязычные переводы могут быть с ошибками.

ОСНОВЫ

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.

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) длиной 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)
```

Создание таблицы с несколькими полями

В таблице может много столбцов. Например, 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  
|BA3   |Россия  
|Tesla |  
|      |
```

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 абсолютно никаких данных.

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.

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 абсолютно 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, к описанию колонки добавляется `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  
);
```

Комментарии к таблице, колонкам

Для создаваемых таблиц и их колонок можно указывать комментарии. Это значитально 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 '';
```

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

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
```

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 разд
```

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 разделителю 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

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.

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

Рассмотрим простой SQL запрос:

```
-- данные, которые мы извлекаем
select emp.name,
       emp.last_name,
       emp.age,
       dept.name
from employees emp
join departments dept on dept.id = emp.department_id -- соединение
where (emp.id = 10 and emp.age > 25) -- условие выборки
order by name desc -- сортировка
```

Конечно, запрос может выглядеть и по-другому, но в целом данный пример раскрывает большую часть из структуры SELECT запроса.

Порядок выполнения

Очень важно понимать, в каком порядке выполняется запрос.

1. Сначала происходит соединение таблиц, если таковое имеется
2. Затем выборка фильтруется условием `where`
3. После этого набор данных извлекается из БД. Все функции, которые указаны в части `SELECT`, выполняются для каждой строки из набора данных
4. И только затем этот набор данных сортируется в указанном порядке

Сейчас это может показаться не столь важным, но знания о порядке выполнения запроса пригодятся, когда мы будем рассматривать запросы с использованием псевдостолбца

`ROWNUM` .

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).

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 комментарий может быть и однострочным:

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

Разделение команд SQL

В одном скрипте может находиться несколько команд SQL. Между собой эти команды должны разделяться символом ";" (точка с запятой).

Например, скрипт ниже создает таблицу `tst`, после чего добавляет в нее данные.

```
create table tst(x number);

insert into tst values(1);
insert into tst values(2);
insert into tst values(3);
```

Многие IDE позволяют выполнять только одну определенную команду из всех присутствующих в скрипте. В каждой это работает по-своему, но как правило это та команда, на которой сейчас стоит курсор.

Регистр

В Oracle ключевые слова(`create`, `delete`, `select`, `join` и т.д.) являются регистронезависимыми. С названиями таблиц и колонок дела обстоят иначе. По умолчанию, наименования таблиц и колонок регистронезависимы. Но если мы будем заключать их в кавычки, в них будет учитываться регистр. Сейчас мы не будем разбирать детали, приведём лишь пример двух абсолютно разных запросов:

```
select *
from "Users";
```

```
select *
from users;
```

Каждый из этих запросов получает данные из абсолютно разных таблиц — `Users` и `users` (или `USERS`). Ещё один пример с использованием регистрозависимых колонок:

```
create table orders(
    order_num varchar2(20),
    "Order_num" varchar2(20)
);

insert into orders(order_num, "Order_num")
values('123', '456');
```

Здесь мы создали таблицу с двумя разными колонками, одна из которых(`"Order_num"`) является регистрозависимой, после чего вставили в таблицу одну строку. Посмотрим на данные в таблице `orders`:


```
select *  
from orders;
```

ORDER_NUM	Order_num
123	456

В учебнике мы не будем использовать эту возможность, и все объекты у нас будут использовать регистронезависимыми.

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.

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`.

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 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
Свинная 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
Свинная отбивная	30.5	320
Овощной салат	5.7	-

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

Сортировка по убыванию. Desc

Для сортировки по убыванию используется `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
Свинная отбивная	30.5	320
Чай с молоком	1.2	280
Чай с лимоном	1.34	270
Борщ	10	130

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
```

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.

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	-

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

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);
```

Операторы сравнения

В `where` можно использовать следующие реляционные операторы:

```
<, >, <=, >=, !=, <>
```

Последние 2 оператора обозначают одно и то же - "Не равно".

Рассмотрим применение данных операторов на примерах.

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

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
Свинная отбивная	30.5	320

Оператор "Больше либо равно"(\geq)

```
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

Оператор "Меньше либо равно"(\leq)

```
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

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 истинным 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.

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 указан, 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` .

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	-

Следует 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	-

Отсутствие в наборе данных. NOT IN

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

Например, 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
Свинная 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.

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`.

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

Подготовка данных

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);

```

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

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 значения 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).

Рассмотрим соединение строк для пользователя с ником `johndoe`: Сначала соединяются таблицы `app_users` и `user_roles`. В результат соединения попадают строки, у которых совпадает логин пользователя.

LOGIN	REGISTRATION_DATE	EMAIL
johndoe	15-NOV-21	johndoe@johndoemail.com

LOGIN	ROLE_ID
johndoe	1
johndoe	2
johndoe	3
alex	3
kate	2
mike	2
dmitry	3

В результате соединения мы получим следующий набор данных(колонку с датой регистрации не показываем):

APP_USERS		USER_ROLES	
LOGIN	EMAIL	LOGIN	ROLE_ID
johndoe	johndoe@johndoemail.com	johndoe	1
johndoe	johndoe@johndoemail.com	johndoe	2
johndoe	johndoe@johndoemail.com	johndoe	3

Здесь следует обратить внимание на то, что значения строк из таблицы `app_users` повторяются для каждой из строк в таблице `user_roles`. Затем мы соединяем получившийся набор данных с таблицей `APP_ROLES`, и к выборке приклеиваются ещё три строки, имеющие совпадение, на этот раз по значению в колонке `ROLE_ID`. При этом строки, которые не имеют совпадения, в выборку не добавляются:

APP_USERS		USER_ROLES		APP_ROLES	
LOGIN	EMAIL	LOGIN	ROLE_ID	ROLE_ID	ROLE_NAME
johndoe	johndoe@johndoemail.com	johndoe	1	1	admin
johndoe	johndoe@johndoemail.com	johndoe	2	2	boss
johndoe	johndoe@johndoemail.com	johndoe	3	3	employee
				4	support

Left join

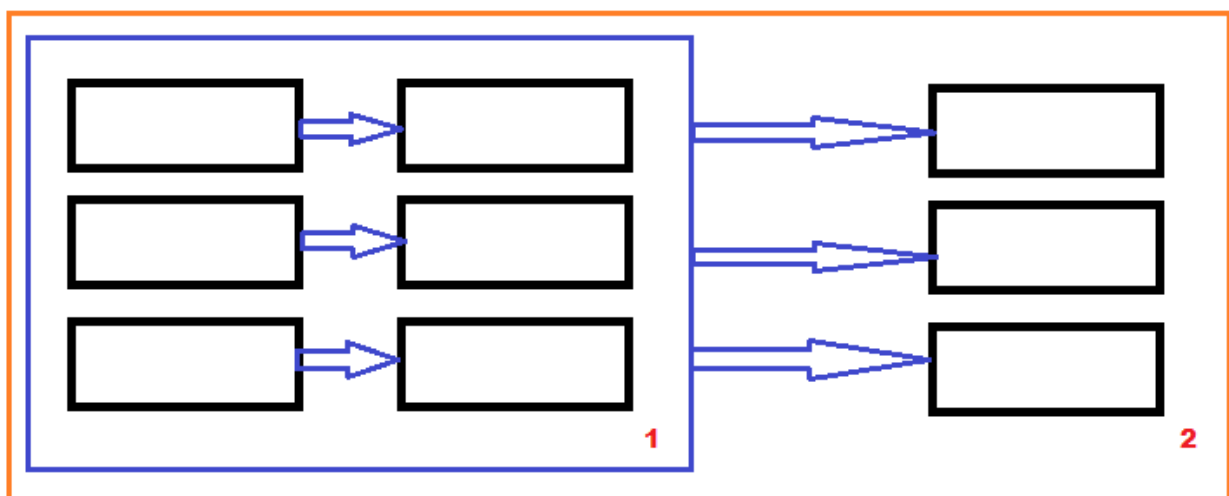
Предыдущий запрос выводил только тех пользователей, у которых действительно были назначены некие роли в приложении. Теперь покажем всех пользователей и их роли. Для этого будет использоваться `LEFT JOIN`. Он отличается от обычного `JOIN` тем, что он не убирает строки из уже имеющегося набора данных когда "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@somemail.com	employee
dmitry	dmitry@somemail.com	employee
mr_dude	mr_dude@email.dude	-

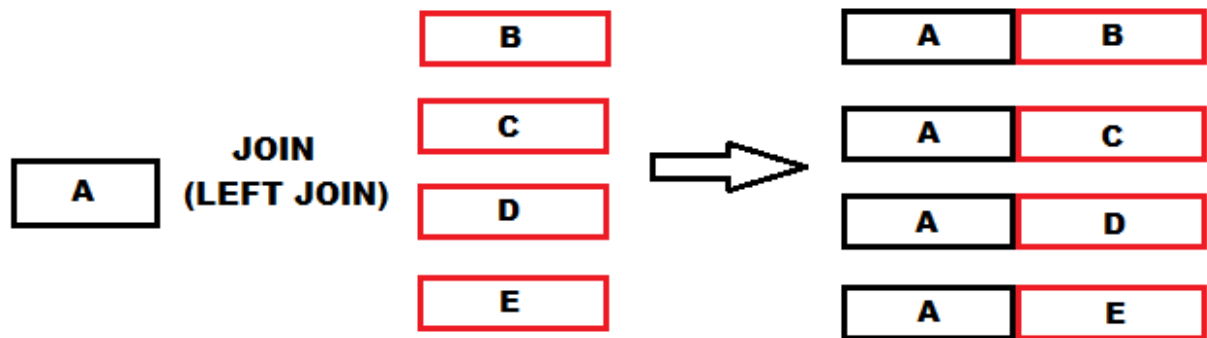
Как видно, теперь к результирующей выборке добавился пользователь `mr_dude`, которому не были назначены права.

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.

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.

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.

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, 'ИТ-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);

```

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.

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.

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

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.

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, почему 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.

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.

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'));
```

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`, равное 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.

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` :

```
select b.*,
       (select book_id from book_orders where 2 > 10) book_id_subq
from books b
```

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` .

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 подзапросе 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?).

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`."

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.

Оператор `EXISTS` имеет вид `EXISTS(подзапрос)`, и возвращает истинное значение в том случае, если подзапрос в скобках возвращает хотя бы одну строку. Может использоваться с оператором `NOT`.

Примеры будем разбирать на данных из части про [подзапросы](#).

Пример №1: Получить список книг, которые заказывались хотя бы раз.

Очевидно что ответом будут те книги, ссылки на которые есть в таблице `book_orders`. Нас устроит любая книга, которая имеет хотя бы один заказ, и поэтому задача легко решается с использованием `EXISTS`:

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

BOOK_ID	BOOK_NAME	AUTHOR	RELEASE_YEAR
1	Властелин колец	Толкин	1954
2	Гордость и предубеждение	Джейн Остин	1813
3	Тёмные начала	Филип Пулман	1995

Здесь использовался коррелированный подзапрос, который проверяет наличие данных в таблице с заказами. Подзапрос может быть любым, и он не обязан быть коррелированным. Все, что проверяет `EXISTS` — это то, вернул ли подзапрос какие-либо данные, или нет.

Следует обратить внимание на то, что мы не указываем наименования колонок в подзапросе, так как сами данные нам не интересны — нас интересует лишь факт наличия данных, которые возвращает подзапрос.

Следующий запрос идентичен предыдущему:

```
select *
from books b
where exists(
    select bo.book_id, bo.quantity
    from book_orders bo where bo.book_id = b.book_id
)
```

Только здесь нет смысла в получении колонок `book_id` и `quantity`, так как они никак не будут использованы.

Пример №2: Получить список книг, которые ни разу не покупали.

Решение здесь прямо обратное предыдущему — нас интересуют такие книги, на которые из таблицы с заказами нет ссылок:

```
select *  
from books b  
where not exists(  
    select null  
    from book_orders bo where bo.book_id = b.book_id  
)
```

BOOK_ID	BOOK_NAME	AUTHOR	RELEASE_YEAR
4	Автостопом по галактике	Дуглас Адамс	1979

"Subquery factoring. WITH "

Часть `WITH` SQL запроса используется для реализации так называемого *Subquery factoring*. Эта возможность позволяет задать подзапрос, который будет доступен в любом месте SQL запроса. Subquery factoring в некоторых случаях значительно упрощает чтение и написание запросов. Более того, велика вероятность того, что при использовании subquery factoring БД построит более оптимальный план выполнения запроса.

Используя Subquery factoring, вы говорите БД о том, что указанный подзапрос вероятно будет использоваться несколько раз в одном запросе, и БД сможет предпринять действия, чтобы более эффективно повторно использовать его.

Подготовка данных

```
create table books(  
    id number primary key,  
    book_name varchar2(400) not null,  
    previous_id number,  
    constraint books_prev_fk foreign key(previous_id)  
        references books(id)  
);
```

```
create table book_orders(  
    id number primary key,  
    book_id number not null,  
    quantity number not null,  
    order_date date not null  
);
```

```
insert into books  
values(1, 'book 1', null);
```

```
insert into books  
values(2, 'Код Да Винчи', null);
```

```
insert into books  
values(3, 'Ангелы и Демоны', 2);
```

```
insert into books  
values(4, 'Инферно', 3);
```

```
insert into books  
values(5, 'Утраченный символ', 4);
```

```
insert into book_orders  
values(1, 1, 1, sysdate);
```

```
insert into book_orders  
values(2, 2, 1, sysdate);
```

```
insert into book_orders  
values(3, 2, 2, sysdate);
```

```
insert into book_orders  
values(4, 3, 1, sysdate - 2);
```

```
insert into book_orders
```



```
values(5, 3, 1, sysdate - 3);

insert into book_orders
values(6, 4, 5, sysdate);
```

Здесь у нас список книг и список заказов. Дополнительно, мы отслеживаем книги из одной серии, используя связь книги и её [предыдущей части](#) (подробнее про иерархические данные можно почитать в соответствующем разделе, ровно как и про [внешние ключи](#)).

Теперь определим задачу: Вывести список книг, количество проданных экземпляров, наряду с количеством проданных экземпляров её предыдущей части, если таковая имеется.

Вариант без With

Вариант 1:

```
select bo.book_name,
       (select sum(quantity) from book_orders bord
        where bord.book_id = bo.id) sold_qty,
       pb.book_name prev_book_name,
       (select sum(quantity) from book_orders bord
        where bord.book_id = pb.id) prev_sold_qty
from books bo
left join books pb on pb.id = bo.previous_id
order by bo.id
```

Вариант 2:

```

select bo.book_name,
       bc.s sold_qty,
       pb.book_name prev_book_name,
       pbc.s prev_sold_qty
from books bo
left join books pb on pb.id = bo.previous_id
left join (
  select sum(quantity) s, book_id
  from book_orders
  group by book_id) bc on bc.book_id = bo.id
left join (
  select sum(quantity) s, book_id
  from book_orders
  group by book_id) pbc on pbc.book_id = pb.id
order by bo.id

```

В обоих случаях мы используем подзапросы — в первом в части SELECT, во втором — в части FROM. Оба варианта дадут один и тот же результат:

BOOK_NAME	SOLD_QTY	PREV_BOOK_NAME	PREV_SOLD_QTY
book 1	1	-	-
Код Да Винчи	3	-	-
Ангелы и Демоны	2	Код Да Винчи	3
Инферно	5	Ангелы и Демоны	2
Утраченный символ	-	Инферно	5

Вариант с WITH

```

with book_cnt as(
    select book_id, sum(quantity) s
    from book_orders
    group by book_id
)
select bo.book_name,
       bc.s sold_qty,
       pb.book_name prev_book_name,
       pbc.s prev_sold_qty
from books bo
left join book_cnt bc on bc.book_id = bo.id
left join books pb on pb.id = bo.previous_id
left join book_cnt pbc on pbc.book_id = bo.previous_id
order by bo.id

```

Получим тот же результат:

BOOK_NAME	SOLD_QTY	PREV_BOOK_NAME	PREV_SOLD_QTY
book 1	1	-	-
Код Да Винчи	3	-	-
Ангелы и Демоны	2	Код Да Винчи	3
Инферно	5	Ангелы и Демоны	2
Утраченный символ	-	Инферно	5

При использовании subquery factoring, мы заранее указываем подзапрос, к который хотим использовать повторно, даём ему псевдоним(`book_cnt`) в нашем случае, и далее обращаемся в основном запросе к этому псевдониму как к обычной таблице.

Работа с множествами

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 псевдонимы 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

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:

Таблица cars

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

Таблица car_offers

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

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
Mercedes SL	2008
Mercedes 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.

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).

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.

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
```

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
```

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`.

Стандарные функции

Функции для работы со строками

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 ф  
insert into articles values ('Насобираю денег', 'johndoe',  
    'Целый год я шел к этой цели, и вот наконец-то у меня все получилось, и заветная с  
insert into articles values ('Задался целью', 'johndoe',  
    'Итак, я задался целью купить себе фотоаппарат. Для начала нужно насобирать денег  
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.

UPPER, LOWER

Данные функции уже описывались раньше.

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

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

Конкатенация строк

Конкатенация - это "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!"

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 возвращён 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
Насоби́рал денег	4
Зада́лся целью	0
Сходи́л в ресторан!	3
Съезди́ли 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 отсчитывает 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
Насоби́р ^а л денег	8
Зада́лся целью	4
Сходи́л в ресторан!	0
Съезди́ли 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

Подобие строк. Like

Для рассмотрения этой темы будем использовать данные из части про [сортировку](#).

Предположим, нам понадобилось 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

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.

Приведение к верхнему регистру. 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.

Замена подстроки. REPLACE

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 удалены 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
Насобираю денег	Целый год я и
Задался целью	Итак, я задаю

Удаление пробелов. TRIM

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;
```

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` .

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

Подготовка тестовых данных

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 вопро  
  
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);
```

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, '') 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 "".

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

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
john doe	статус пуст	статус пуст	статус всегда заполнен
admin	Я админ. Все вопросы ко мне	Я админ. Все вопросы ко мне	статус всегда заполнен
nelson	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 dual
```

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

Условные функции

Условные функции - это такие функции, которые могут возвращать разные результаты в зависимости от выполнения тех или иных условий.

В качестве тестовых данных будем использовать таблицу из части про [функции для работы с NULL](#).

DECODE

Функция DECODE в общем случае имеет следующий вид:

```
DECODE(что сравниваем,  
      значение1, результат1,  
      значение2, результат2,  
      значение3, результат3,  
      ....  
      значениеN, результатN,  
      значение по-умолчанию)
```

Первым DECODE принимает параметр, значение которого будет сравниваться по очереди со списком значений, и в случае, когда он совпадет с одним из перечисленных, будет возвращен соответствующий результат. Если совпадений не найдено, будет возвращено значение по-умолчанию. Если значение по-умолчанию не указано, будет возвращен `NULL`.

Аргументы могут быть числового, строкового типа, или датой.

DECODE может сравнивать NULL значения:

```
select login,  
       status,  
       decode(status, -- <- Что сравниваем  
              null, 'Статус не указан', -- <- пара Значение-Результат  
              'Статус указан') has_status -- <- Значение по-умолчанию  
from profiles
```

LOGIN	STATUS	HAS_STATUS
johndoe	-	Статус не указан
admin	Я админ. Все вопросы ко мне	Статус указан
nelson	-	Статус не указан

Перед сравнением Oracle автоматически приводит первый параметр и все значения к типу первого значения в списке параметров. Результат функции автоматически приводится к типу первого результата в списке параметров. Если первый результат в списке - `NULL`, результат функции `DECODE` будет приведен к строковому типу `VARCHAR2`.

Например, следующий запрос не выполнится из-за ошибки `ORA-01722: invalid number`:

```
select decode(login,
               'admin', 10,
               'Администратор') has_status
from profiles
```

Тип возвращаемого значения определяется первым результатом в списке параметров, в данном случае - числом "10". Но значение по-умолчанию имеет строковый тип, что и приводит к ошибке. Чтобы ошибки не было, нужно либо значение по-умолчанию заменить на число, либо заменить число 10 на любой строковый тип.

Любой из следующих запросов отработает без ошибок:

```
select decode(login,
               'admin', 'Администратор',
               'Не администратор') admin_login
from profiles;

select decode(login,
               'admin', 10,
               20) admin_login_flag
from profiles;
```

В качестве проверяемого значения не обязательно должна быть колонка таблицы. В следующем примере проверяем длину логина пользователя:


```
select login,
       length(login) login_length,
       decode(length(login),
              5, 'Пять',
              6, 'Шесть',
              'Не пять и не шесть') admin_login_length
from profiles
```

LOGIN	LOGIN_LENGTH	ADMIN_LOGIN_LENGTH
admin	5	Пять
johndoe	7	Не пять и не шесть
nelsol	6	Шесть

Максимальное количество параметров в функции `DECODE` - 255.

Предыдущий пример, только с использованием вложенного `DECODE` :

```
select login,
       length(login) login_length,
       decode(length(login),
              5, 'Пять',
              decode(length(login),
                     6, 'Шесть',
                     'Не пять и не шесть')) admin_login_length
from profiles
```

Здесь в качестве значения по-умолчанию выступает еще один `DECODE` .

На практике вложенных `decode` следует избегать, ровно как и `decode` с большим количеством параметров.

Одна из распространенных ошибок - использовать `DECODE` для того, чтобы преобразовать какие-либо флаги в их строковые эквиваленты (при их большом количестве):

```
select a.*,
       decode(a.status,
              1, 'Закрыт',
              2, 'Отменен',
              3, 'Новый',
              4, 'В обработке'
              ) status_name
from some_table a
```

Для подобных ситуаций лучше создать отдельную таблицу с кодом статуса и его строковым значением, и использовать [соединения](#):

```
select a.*,
       st.status_name
from some_table a
join statuses st on st.status_code = a.status
```

CASE

Выражение `CASE` во многом похоже на `DECODE`, но обладает большими возможностями. Данное выражение позволяет реализовать полноценную условную логику в SQL запросе.

`CASE` может использоваться в двух вариантах - простом(англ. simple case expression) и поисковом(англ. searched case expression).

Простой `CASE` по принципу работы идентичен `DECODE`:

```
select login,
       case login -- <- что сравниваем
         when 'admin' then 'Администратор' -- Результат 1
         when 'johndoe' then 'Джон До' -- Результат 2
         else 'Другой пользователь' -- Значение по-умолчанию
       end user_flag
from profiles
```

LOGIN	USER_FLAG
admin	Администратор
johndoe	Джон До
nelsol	Другой пользователь

`user_flag` здесь - псевдоним для столбца. Само выражение начинается с ключевого слова `case` и заканчивается ключевым словом `end`.

Как и в `DECODE`, для проверяемого значения начинают производиться сравнения со значениями в блоках `when`. При первом же совпадении функция завершает работу и возвращает соответствующий результат (указанный после `then`). В случае, если ни одного совпадения не было найдено, возвращается значение, указанное в блоке `ELSE`. Если значение по-умолчанию не указано, будет возвращен `NULL`.

Searched case expression, в отличие от simple case expression, является куда более мощным инструментом. В отличие от последнего, в searched case expression в блоках `when` указываются условия, а не просто значения для сравнения:

```

select login,
       case
         when login = 'admin' then 'Администратор'
         else 'Не администратор'
       end is_admin
from profiles

```

LOGIN	IS_ADMIN	
admin	Администратор	
johndoe	Не администратор	
nelsol	Не администратор	

```

select login,
       case
         when length(login) = 5 then 'Пять'
         when length(login) > 5 then 'Больше пяти'
         when length(login) between 0 and 4 then 'От 0 до 4'
       end login_length_stats
from profiles

```

LOGIN	LOGIN_LENGTH_STATS	
admin	Пять	
johndoe	Больше пяти	
nelsol	Больше пяти	

В общем и целом, лучше использовать `DECODE` для небольших, простых сравнений, и `CASE` для более сложных, т.к. он лучше читается.

Условные функции в WHERE части

Условные функции спокойно могут использоваться в WHERE-части запроса, как и другие функции:

```
-- Выведет профили пользователей, которые
-- не являются администраторами
select *
from profiles
where case
    when login = 'admin' then 1
    else 0
end = 0
```

LOGIN	LAST_UPDATED	STATUS
johndoe	01-JAN-09	-
nelson	-	-

В примере выше выражение case вернет 0 в тех случаях, когда логин пользователя не будет логином администратора. Сразу после окончания выражения мы сравниваем его с нулем, тем самым получая только не-администраторов. Подобные способы, конечно, лучше не использовать, а вместо них прибегать к классическому варианту написания запроса, который будет более понятным:

```
select *
from profiles
where login <> 'admin'
```

При группировке условные функции, как и все другие, должны быть полностью продублированы в `GROUP BY`, использовать псевдоним колонки не получится:

```
-- Этот запрос не работает
select case
    when length(login) > 5 then '> 5'
    when length(login) < 5 then '< 5'
    when length(login) = 5 then '= 5'
end login_length,
count(*) cnt
from profiles
group by login_length;

-- А вот этот отработает корректно
select case
    when length(login) > 5 then '> 5'
    when length(login) < 5 then '< 5'
    when length(login) = 5 then '= 5'
end login_length,
count(*) cnt
from profiles
group by case
    when length(login) > 5 then '> 5'
    when length(login) < 5 then '< 5'
    when length(login) = 5 then '= 5'
end
```

LOGIN_LENGTH	CNT
= 5	1
> 5	2

Запрос выше выведет статистику о количестве логинов пользователей с определенной длиной - меньше пяти символов, больше пяти символов, или с длиной логина ровно в пять символов.

Последний запрос можно переписать с использованием [подзапроса](#), чтобы не дублировать

```
CASE B GROUP BY :
```

```
select login_length,  
       count(*)  
from (  
  select case  
    when length(login) > 5 then '> 5'  
    when length(login) < 5 then '< 5'  
    when length(login) = 5 then '= 5'  
  end login_length,  
  login,  
  status,  
  last_updated  
  from profiles  
)  
group by login_length
```

Битовые операции при работе с БД применяются редко. Тем не менее, работа с отдельными битами поддерживается в БД Oracle, и в некоторых случаях может быть использована в весьма элегантном виде.

Тестовые данные

```
create table docs(  
    id number not null primary key,  
    doc_num varchar2(100 char) not null,  
    bit_access number default 0 not null  
);  
  
comment on table docs is 'Документы';  
  
comment on column docs.bit_access is 'Уровни доступа(1 бит - чтение, 2 - редактирование)';  
  
insert into docs  
values(1, '1-1', 0);  
  
insert into docs  
values(2, '2-1', 1);  
  
insert into docs  
values(3, '2-2', 4);  
  
insert into docs  
values(4, '3-1', 3);  
  
insert into docs  
values(5, '4-1', 7);
```

Колонка `bit_access` хранит в себе число, каждый бит которого отвечает за наличие(1) или отсутствие(0) доступа на произведение операций с данной строкой таблицы(документом). То есть, если в числе, находящемся в колонке `bit_access`, первый бит равен 1, это означает, что данную запись можно показывать пользователю. Если второй бит равен 1, то данную строку можно редактировать, а если третий бит установлен в 1, то данную строку можно удалять. Если мы представим наше число в виде трех бит, оно будет иметь вид `000`. Порядок бит в записи числа как правило идет справа налево, т.е. если в числе установлен первый бит в 1, то оно записывается как `001`, если второй, то `010`.

При этом доступ может быть комбинированным - мы можем иметь доступ и на просмотр информации по документу, и на удаление самого документа, или, скажем, на просмотр и редактирование, но не на удаление. В таких случаях в нашем числе несколько бит числа

будут установлены в 1 (Само число в двоичной записи будет `101` в первом случае и `011` во втором).

Подобным образом можно закодировать несколько логических переменных в одно число, в котором каждый бит будет отвечать за соответствующее условие.

Для того, чтобы проверить, установлен ли определенный бит числа в 1, применяется операция, которая называется побитовой конъюнкцией, или побитовым "И". Результатом побитового "И" между числами `a` и `b` будет такое число `c`, у которого в 1 будут установлены только те биты, которые установлены в 1 как в `a`, так и в `b`. Вот как это будет выглядеть, если `a= 011`, а `b=110`:

```
| A | 0 | 1 | 1
| B | 1 | 1 | 0
| C | 0 | 1 | 0
```

Таким образом, чтобы убедиться, что интересующий нас набор бит (предположим, этот набор бит хранится в числе `b`) установлен в числе `a`, нужно применить побитовое "И" к числам `a` и `b`, и получившийся результат сравнить с числом `b` (еще это число называют битовой маской) - если результат совпал, значит, все биты, установленные в числе `b` установлены и в числе `a`.

В таблице числа мы храним в десятичной системе, и чтобы было проще ориентироваться, распишем, что означают текущие данные в таблице:

```
|=====
|bit_access | двоичное представление | Доступ
|0          | 000                      | Ничего нельзя делать
|1          | 001                      | Чтение
|3          | 011                      | Чтение и редактирование
|7          | 111                      | Чтение, редактирование, удаление
|=====
```

BIN_TO_NUM

Эта функция принимает список нулей и единиц, превращая их в десятичное число:


```
select bin_to_num(0,0,0) a,
       bin_to_num(0,0,1) b,
       bin_to_num(0,1,1) c,
       bin_to_num(1,1,1) d
from dual
```

A	B	C	D
0	1	3	7

BITAND. Побитовое "И"

Функция `BITAND` выполняет побитовое "И" между двумя числами.

Выведем список всех документов, которые доступны для чтения:

```
select *
from docs
where bitand(bit_access, 1) = 1
```

ID	DOC_NUM	BIT_ACCESS
2	2-1	1
4	3-1	3
5	4-1	7

Список всех документов, которые доступны для чтения и редактирования:

```
select *
from docs
where bitand(bit_access, 3) = 3
```

ID	DOC_NUM	BIT_ACCESS
4	3-1	3
5	4-1	7

Чтобы было более наглядно и читаемо, последний запрос можно переписать с использованием функции `bin_to_num` :

```
select *
from docs
where bitand(bit_access, bin_to_num(0,1,1)) = bin_to_num(0,0,1)
```

Для улучшения читаемости кода лучше использовать `bin_to_num` для записи битовых масок.

Выведем список всех документов, и добавим к выборке три колонки, каждая из которых будет отвечать за наличие доступа к определенному действию:

```
select id,
       doc_num,
       bit_access,
       case
         when bitand(bit_access,
                     bin_to_num(0,0,1)) = bin_to_num(0,0,1) then 'да'
       else 'Нет'
       end read_access,
       case
         when bitand(bit_access,
                     bin_to_num(0,1,0)) = bin_to_num(0,1,0) then 'да'
       else 'Нет'
       end edit_access,
       case
         when bitand(bit_access,
                     bin_to_num(1,0,0)) = bin_to_num(1,0,0) then 'да'
       else 'Нет'
       end delete_access
from docs
```

ID	DOC_NUM	BIT_ACCESS	READ_ACCESS	EDIT_ACCESS	DELETE_ACCESS
1	1-1	0	Нет	Нет	Нет
2	2-1	1	Да	Нет	Нет
3	2-2	4	Нет	Нет	Да
4	3-1	3	Да	Да	Нет
5	4-1	7	Да	Да	Да

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

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 функции 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

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.

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.

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.

Тип 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.

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.

```
-- 1 марта 2020 года
select to_date('2020-03-01', 'yyyy-mm-dd') d1,
-- 3 января 2020 года
to_date('2020-03-01', 'yyyy-dd-mm') d2 -- <2>
from dual
```

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` .

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
```

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

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.

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,
       -- полгода после текущей даты
       add_months(sysdate, 6) d2, -- <1>
       -- полгода до текущей даты
       add_months(sysdate, -6) d3 -- <2>
from dual
```

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

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-
       sysdate + 1 tomorrow, -- на 1 день больше
       sysdate - 1 yesterday-- на 1 день меньше
from dual
```

A	B	TOMORROW	YESTERDAY
4	.0416666666666666666666666666666666666667	12-APR-20	10-APR-20

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	1.70967741935483870967741935483870967742

Тип TIMESTAMP

Тип `TIMESTAMP` является расширением типа `DATE`. Он также, как и тип `DATE`, позволяет хранить год, месяц, день, часы, минуты и секунды. Но помимо 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.

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

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

Приведение строки к timestamp

Для приведения строки к типу `timestamp` используется функция `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](#).

Аналитические функции - очень мощный инструмент в SQL. Со слов Тома Кайта, можно написать отдельную книгу по аналитическим функциям, настолько они полезны.

Аналитические функции - это те же агрегирующие функции, но их главная особенность в том, что они работают без необходимости группировки строк.

Аналитические функции выполняются последними в запросе, поэтому они могут быть использованы только в `SELECT` части запроса, либо в `ORDER BY` .

Для примера возьмем данные, которые мы использовали при разборе агрегирующих функций:

```
alter table employees
add (exp number);

merge into employees emp
using (select level lvl, rownum * 10 exp
      from dual
      connect by level <= 4) val
on (emp.id = val.lvl)
when matched then
  update
  set emp.exp = val.exp;
```

Посмотрим, какие данные теперь хранятся в таблице:

```
select *
from employees
```

ID	FIRST_NAME	LAST_NAME	BD	JOB	EXP
1	Василий	Петров	07-OCT-90	Машинист	10
2	Александр	Сидоров	18-JUL-80	Бухгалтер	20
3	Евгения	Цветочкина	18-JUL-78	Бухгалтер	30
4	Владимир	Столяров	18-JUL-77	Слесарь	40

Теперь напомним запрос, который бы возвращал максимальный стаж среди всех сотрудников отдельной колонкой. Для этого можно использовать подзапрос:

```
select id,
       first_name,
       last_name,
       (select max(exp) from employees) max_exp
from employees
```

ID	FIRST_NAME	LAST_NAME	MAX_EXP
1	Василий	Петров	40
2	Александр	Сидоров	40
3	Евгения	Цветочкина	40
4	Владимир	Столяров	40

Усложним задачу: напишем запрос, который будет возвращать отдельной колонкой максимальный стаж на должности каждого сотрудника. Для этого также можно использовать подзапрос, только уже коррелированный:

```
select emp.first_name,
       emp.last_name,
       emp.job,
       (select max(exp) from employees where job = emp.job) max_exp
from employees emp
```

FIRST_NAME	LAST_NAME	JOB	MAX_EXP
Владимир	Столяров	Слесарь	40
Василий	Петров	Машинист	10
Александр	Сидоров	Бухгалтер	30
Евгения	Цветочкина	Бухгалтер	30

Теперь решим эти же задачи при помощи аналитических функций:

```
select emp.first_name,
       emp.last_name,
       emp.job,
       max(exp) over () max_exp
from employees emp

select emp.first_name,
       emp.last_name,
       emp.job,
       max(exp) over (partition by job) max_exp
from employees emp
```

Аналитические функции позволяют использовать агрегирующие функции без подзапросов, что уменьшает размер запроса(примеры, когда использование подзапроса усложняет чтение запроса, будут немного дальше).

Помимо этого, вот еще два примера запросов с аналитическими функциями.

```
select id,
       first_name,
       last_name,
       job,
       bd,
       exp,
       max(exp) over (order by first_name) max_exp_asc,
       max(exp) over (order by first_name desc) max_exp_desc
from employees emp
```

ID	FIRST_NAME	LAST_NAME	JOB	BD	EXP	MAX_EXP_ASC	MAX_EXP_DESC
3	Евгения	Цветочкина	Бухгалтер	18-JUL-78	30	40	30
4	Владимир	Столяров	Слесарь	18-JUL-77	40	40	40
1	Василий	Петров	Машинист	07-OCT-90	10	20	40
2	Александр	Сидоров	Бухгалтер	18-JUL-80	20	20	40

Две колонки, `max_exp_asc` и `max_exp_desc`, считают максимальный стаж среди сотрудников в порядке возрастания и убывания их имен соответственно.

С простыми примерами аналитических функций мы познакомились, теперь разберемся, как они работают.

Когда агрегирующая функция становится аналитической

В первом примере агрегирующая функция `max` превратилась в аналитическую после добавления к ней части `over()`. В итоге, было найдено максимальное значение колонки `exp` среди всего набора данных, и это значение было добавлено к каждой строке выборки, без группировки.

Подсчет результатов по группам. Partition by

Для того, чтобы результаты считались по определенным группам, нужно использовать конструкцию `partition by`, в которой нужно указать колонки, по которым будет производиться вычисление.

```

select emp.first_name,
       emp.last_name,
       emp.job,
       max(exp) over (partition by job) max_exp
from employees emp

```

В данном примере, который уже приводился раньше, максимальный стаж вычисляется в отдельности для каждой из профессий, и затем добавляется к каждой строке.

Посчитаем количество сотрудников по должностям и выведем отдельной колонкой:

```

select emp.first_name,
       emp.last_name,
       emp.job,
       count(*) over (partition by job) job_cnt
from employees emp

```

FIRST_NAME	LAST_NAME	JOB	JOB_CNT
Александр	Сидоров	Бухгалтер	2
Евгения	Цветочкина	Бухгалтер	2
Василий	Петров	Машинист	1
Владимир	Столяров	Слесарь	1

Результаты можно считать по нескольким группам. Выведем напротив каждого сотрудника общее число сотрудников, родившихся в том же месяце(колонка `mnth_cnt`) и количество сотрудников, родившихся в том же месяце и занимающих такую же должность:

```

select emp.id,
       emp.first_name,
       emp.last_name,
       emp.job,
       emp.bd,
       count(*) over(
           partition by extract(month from emp.bd)
       ) mnth_cnt,
       count(*) over (
           partition by extract(month from emp.bd), job
       ) mnth_cnt
from employees emp

```

ID	FIRST_NAME	LAST_NAME	JOB	BD	MNTH_CNT	MNTH_CNT
2	Александр	Сидоров	Бухгалтер	18-JUL-80	3	2
3	Евгения	Цветочкина	Бухгалтер	18-JUL-78	3	2
4	Владимир	Столяров	Слесарь	18-JUL-77	3	1
1	Василий	Петров	Машинист	07-OCT-90	1	1

Всего есть три сотрудника, которые родились в одном и том же месяце - июле. Поэтому в колонке `mnth_cnt` отображается число 3. В то же время, есть лишь два сотрудника, которые родились в одном и том же месяце, и при этом занимают одну и ту же должность - это сотрудники с id равными 2 и 3.

Порядок вычисления. Order by

В аналитических функциях можно указывать порядок, в котором они будут работать с итоговым набором данных. Для этого используется конструкция `order by`.

Пронумеруем строки в нашей таблице в порядке возрастания и убывания дней рождения сотрудников.

```
select emp.id,
       emp.first_name,
       emp.last_name,
       emp.job,
       emp.bd,
       row_number() over (order by bd) bd_asc,
       row_number() over (order by bd desc) bd_desc
from employees emp
```

ID	FIRST_NAME	LAST_NAME	JOB	BD	BD_ASC	BD_DESC
1	Василий	Петров	Машинист	07-OCT-90	4	1
2	Александр	Сидоров	Бухгалтер	18-JUL-80	3	2
3	Евгения	Цветочкина	Бухгалтер	18-JUL-78	2	3
4	Владимир	Столяров	Слесарь	18-JUL-77	1	4

Функция `row_number` возможно является одной из самых часто используемых аналитических функций. Она возвращает номер строки в итоговой выборке. До ее появления в Oracle подобного функционала можно было достичь лишь при использовании подзапросов и псевдостолбца `ROWNUM`.

Аналитические функции могут работать не только по группам или в определенном порядке, но и в определенном порядке в пределах заданной группы:


```

select emp.id,
       emp.first_name,
       emp.last_name,
       emp.job,
       emp.bd,
       row_number() over (
         partition by job order by bd
       ) bd_asc,
       row_number() over (
         partition by job order by bd desc
       ) bd_desc
from employees emp

```

ID	FIRST_NAME	LAST_NAME	JOB	BD	BD_ASC	BD_DESC
3	Евгения	Цветочкина	Бухгалтер	18-JUL-78	1	2
2	Александр	Сидоров	Бухгалтер	18-JUL-80	2	1
1	Василий	Петров	Машинист	07-OCT-90	1	1
4	Владимир	Столяров	Слесарь	18-JUL-77	1	1

Здесь нумерация производится отдельно для каждой группы. У двух сотрудников с одинаковой должностью нумерация была проставлена в порядке их дней рождения.

Диапазон работы аналитических функций

Аналитические функции всегда, явно или неявно, применяются к определенному набору строк, называемому окном аналитической функции.

Не во всех аналитических функциях можно указывать окно. Среди самых часто используемых функций, для которых можно указывать окно, находятся `MIN`, `MAX`, `SUM`, `AVG`, `COUNT`, `LAST_VALUE`, `FIRST_VALUE` и другие.

Чтобы примеры были немного более практичными, создадим еще одну таблицу, в которой будем хранить данные о начисленных зарплатах сотрудникам по месяцам:

```

create table emp_salary(
    emp_id number not null,
    sal_date date not null,
    sal_value number not null,
    -- Начисления в данной таблице должны быть
    -- "сбитыми" по месяцам, и чтобы в данных не
    -- возникло ошибки, создаем уникальный ключ на
    -- поля с id сотрудника и месяцем начисления
    constraint emp_salary_uk unique(emp_id, sal_date)
);

comment on table emp_salary is
    'Зачисленные средства по месяцам';
comment on column emp_salary.emp_id is
    'id сотрудника';
comment on column emp_salary.sal_date is
    'Месяц начисления';
comment on column emp_salary.sal_value is
    'Начисленные средства';

insert into emp_salary(emp_id, sal_date, sal_value)
values(1, to_date('01.01.2020', 'dd.mm.yyyy'), 1000);

insert into emp_salary(emp_id, sal_date, sal_value)
values(1, to_date('01.02.2020', 'dd.mm.yyyy'), 1320);

insert into emp_salary(emp_id, sal_date, sal_value)
values(1, to_date('01.03.2020', 'dd.mm.yyyy'), 850);

insert into emp_salary(emp_id, sal_date, sal_value)
values(2, to_date('01.01.2020', 'dd.mm.yyyy'), 1000);

insert into emp_salary(emp_id, sal_date, sal_value)
values(2, to_date('01.02.2020', 'dd.mm.yyyy'), 800);

insert into emp_salary(emp_id, sal_date, sal_value)
values(2, to_date('01.03.2020', 'dd.mm.yyyy'), 1200);

insert into emp_salary(emp_id, sal_date, sal_value)
values(3, to_date('01.01.2020', 'dd.mm.yyyy'), 1030);

```

```
insert into emp_salary(emp_id, sal_date, sal_value)
values(4, to_date('01.01.2020', 'dd.mm.yyyy'), 3700);
```

Общие данные выглядят следующим образом:

```
select e.first_name,
       e.last_name,
       e.job,
       es.sal_date,
       es.sal_value
from emp_salary es
join employees e on e.id = es.emp_id
```

FIRST_NAME	LAST_NAME	JOB	SAL_DATE	SAL_VALUE
Василий	Петров	Машинист	01-JAN-20	1000
Василий	Петров	Машинист	01-FEB-20	1320
Василий	Петров	Машинист	01-MAR-20	850
Александр	Сидоров	Бухгалтер	01-JAN-20	1000
Александр	Сидоров	Бухгалтер	01-FEB-20	800
Александр	Сидоров	Бухгалтер	01-MAR-20	1200
Евгения	Цветочкина	Бухгалтер	01-JAN-20	1030
Владимир	Столяров	Слесарь	01-JAN-20	3700

Теперь добавим колонку к выборке, которая будет показывать, как изменялась минимальная заработная плата сотрудников с течением времени.

```
select e.first_name,
       e.last_name,
       e.job,
       es.sal_date,
       es.sal_value,
       min(es.sal_value) over (
         order by sal_date
         rows between unbounded preceding and
                    current row
       ) min
from emp_salary es
join employees e on e.id = es.emp_id
```

FIRST_NAME	LAST_NAME	JOB	SAL_DATE	SAL_VALUE	MIN
Василий	Петров	Машинист	01-JAN-20	1000	1000
Александр	Сидоров	Бухгалтер	01-JAN-20	1000	1000
Евгения	Цветочкина	Бухгалтер	01-JAN-20	1030	1000
Владимир	Столяров	Слесарь	01-JAN-20	3700	1000
Александр	Сидоров	Бухгалтер	01-FEB-20	800	800
Василий	Петров	Машинист	01-FEB-20	1320	800
Александр	Сидоров	Бухгалтер	01-MAR-20	1200	800
Василий	Петров	Машинист	01-MAR-20	850	800

Сейчас может показаться, что результат, который получился в запросе такой же, как если бы мы и не задавали размер окна. Действительно, для текущих данных запрос без добавления лишних ключевых слов выдает такой же результат:

```
select e.first_name,
       e.last_name,
       e.job,
       es.sal_date,
       es.sal_value,
       sum(es.sal_value) over (order by sal_date) min
from emp_salary es
join employees e on e.id = es.emp_id
```

FIRST_NAME	LAST_NAME	JOB	SAL_DATE	SAL_VALUE	MIN
Василий	Петров	Машинист	01-JAN-20	1000	1000
Александр	Сидоров	Бухгалтер	01-JAN-20	1000	1000
Евгения	Цветочкина	Бухгалтер	01-JAN-20	1030	1000
Владимир	Столяров	Слесарь	01-JAN-20	3700	1000
Александр	Сидоров	Бухгалтер	01-FEB-20	800	800
Василий	Петров	Машинист	01-FEB-20	1320	800
Александр	Сидоров	Бухгалтер	01-MAR-20	1200	800
Василий	Петров	Машинист	01-MAR-20	850	800

Чуть позже станет понятно, что это 2 совершенно разных запроса, а пока разберем подробнее различные варианты указания окна в аналитических функциях.

Строки и значения

Строки, которые определяют окно работы аналитической функции, можно указывать физически, т.е. сказать БД: "Для текущей строки в выборке аналитическая функция должна обработать две строки перед ней и три строки после нее"; или: "Для текущей строки в выборке аналитическая функция должна обработать все строки начиная с текущей и заканчивая всеми последующими строками".

Вторым возможным способом определения окна является определение не по физическому расположению строки в выборке, а по значениям, которые строки в себе содержат. Мысленно это можно произнести: "Для текущей строки в выборке аналитическая функция должна обработать те строки, в которых значение колонки А будет больше, чем значение в колонке А текущей строки"; или: "Для текущей строки в выборке аналитическая функция должна обработать те строки, в которых значение колонки А будет в пределах от 10 до 20 включительно".

В первом случае, при указании физических строк, используется ключевое слово `rows`, во втором случае, при указании строк по их значениям, используется ключевое слово `range`.

Смещения при определении окна

Итак, при указании окна мы должны задать его верхнюю и нижнюю границу.

В общем виде указание границы выглядит следующим образом:

```
(range или rows) between "Верхняя граница" and "Нижняя граница"
```

Теперь рассмотрим варианты для этих границ:

- `UNBOUNDED PRECEDING` - указывает, что окно начинается с первой строки в разделе. Может быть указано только для верхней границы, в качестве нижней границы использовать нельзя.
- `UNBOUNDED FOLLOWING` - указывает, что окно заканчивается на последней строке в разделе. Может быть указано только для нижней границы.
- `CURRENT ROW` - обозначает текущую строку или значение. Может быть использовано как для нижней границы, так и для верхней.
- `PRECEDING` - значение в строке или физическая строка, которая предшествует текущей строке на
- `FOLLOWING` - значение в строке или физическая строка, которая находится впереди текущей строки на

Следует помнить, что если окно задается с использованием `rows`, т.е. указываются строки, то и границы окна будут задаваться в строках, и наоборот, если используется `range`, то границы окна будут учитываться по значениям в строках.

Если окно не указывается, то по-умолчанию оно имеет вид `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`.

Теперь посмотрим на один из предыдущих запросов:

```
select e.first_name,
       e.last_name,
       e.job,
       es.sal_date,
       es.sal_value,
       min(es.sal_value) over (
         order by sal_date
         rows between unbounded preceding
                and current row
       ) min
from emp_salary es
join employees e on e.id = es.emp_id
```

Рассмотрим, как будет работать аналитическая функция.

`PARTITION BY` не указан, значит результаты будут "сплошные" и не будут разбиваться по группам. Обработаться строки будут в порядке возрастания даты в колонке `sal_date`, а диапазон строк, для которых будет вычисляться функция, задается первой строкой во всем наборе данных и заканчивается текущей строкой.

Теперь должна быть понятна разница между данным запросом и запросом без указания окна, о которой говорилось в начале раздела - по-умолчанию окно задается по значению, а мы установили размер окна со смещениями в строках.

Еще один важный момент: значения в колонке `sal_date` не являются уникальными. Это означает, что результат будет недетерминированным, т.е. может отличаться от запуска к запуску, т.к. порядок следования строк в выборке может измениться.

Чтобы избавиться от такого эффекта, можно добавить еще одну колонку в конструкцию `order by`, чтобы сделать порядок следования строк уникальным и не меняющимся. В данном случае мы можем дополнительно сортировать данные по `id` сотрудника:

```
min(es.sal_value) over (
  order by sal_date, id
  rows between unbounded preceding
        and current row
) min
```

В общем, когда несколько колонок имеют одинаковые значения, аналитические функции работают по определенным правилам:

- Функции `CUME_DIST`, `DENSE_RANK`, `NTILE`, `PERCENT_RANK` и `RANK` возвращают одинаковый результат для всех строк

- Функция `ROW_NUMBER` присвоит каждой строке уникальное значение. Порядок присваивания будет зависеть от порядка обработки строк БД, который мы не можем предугадать
- Все остальные функции будут работать по-разному в зависимости от спецификации окна. Если окно задавалось при помощи `RANGE`, то функция вернет одинаковое значение для всех строк. Если использовалось ключевое слово `ROWS`, то результат нельзя будет предугадать - он опять же будет зависеть от порядка обработки строк базой данных, который может отличаться для одного и того же набора данных от запуска к запуску.

Размеры окна можно задавать в виде смещений:

```
select e.first_name,
       e.last_name,
       e.job,
       es.sal_date,
       es.sal_value,
       round(
         avg(es.sal_value)over (
           order by sal_date, id
           rows between 2 preceding and current row
         ),
       2) avg_sal
from emp_salary es
join employees e on e.id = es.emp_id
```

FIRST_NAME	LAST_NAME	JOB	SAL_DATE	SAL_VALUE	AVG_SAL
Василий	Петров	Машинист	01-JAN-20	1000	1000
Александр	Сидоров	Бухгалтер	01-JAN-20	1000	1000
Евгения	Цветочкина	Бухгалтер	01-JAN-20	1030	1010
Владимир	Столяров	Слесарь	01-JAN-20	3700	1910
Александр	Сидоров	Бухгалтер	01-FEB-20	800	1843.33
Василий	Петров	Машинист	01-FEB-20	1320	1940
Александр	Сидоров	Бухгалтер	01-MAR-20	1200	1106.67
Василий	Петров	Машинист	01-MAR-20	850	1123.33

Здесь в колонке `avg_sal` считается средняя заработная плата по трем строкам - двум предшествующим и текущей. Порядок следования, как мы помним, задается при помощи `ORDER BY`, поэтому две предшествующие строки

- это строки, у которых значение в колонках `sal_date` будет меньше либо равным значению в текущей строке.

Значение функции округляется до двух знаков после запятой при помощи функции `round` . Аналитическая функция берется в скобки полностью, начиная от имени функции и заканчивая определением окна. К значениям, полученным при помощи аналитических функций можно применять другие функции или операторы - например, можно было бы добавить 100 к среднему значению:

```
avg(es.sal_value)over (
    order by sal_date
    rows between 2 preceding and current row
) + 100 avg_sal
```

Или даже получить разность между значениями двух аналитических функций:

```
max(es.sal_value)
over (
    order by sal_date
    range between 1 preceding and current row
) -
min(es.sal_value)
over (
    order by sal_date
    rows between 1 preceding and current row
)
```

В следующем примере смещение задается не в строках, а в диапазоне значений, которые содержит колонка `sal_value`:

```
select e.first_name,
       e.last_name,
       e.job,
       es.sal_date,
       es.sal_value,
       sum(es.sal_value) over (
           order by sal_value
           range between 1000 preceding and current row
       ) max_sal
from emp_salary es
join employees e on e.id = es.emp_id
```


FIRST_NAME	LAST_NAME	JOB	SAL_DATE	SAL_VALUE	SUM_SAL
Александр	Сидоров	Бухгалтер	01-FEB-20	800	800
Василий	Петров	Машинист	01-MAR-20	850	1650
Василий	Петров	Машинист	01-JAN-20	1000	3650
Александр	Сидоров	Бухгалтер	01-JAN-20	1000	3650
Евгения	Цветочкина	Бухгалтер	01-JAN-20	1030	4680
Александр	Сидоров	Бухгалтер	01-MAR-20	1200	5880
Василий	Петров	Машинист	01-FEB-20	1320	7200
Владимир	Столяров	Слесарь	01-JAN-20	3700	3700

Т.к. использовался `RANGE`, то сумма рассчитывается для всех строк, значение которых находится в диапазоне от 1000 до значения в текущей строке.

Еще раз, следует обратить внимание, что строки, которые находятся после текущей, также обрабатываются функцией, если значение колонки `sal_value` входит в заданный диапазон. Это можно видеть на изображении выше, в строках, где значение `sal_value` равно 1000 - для первой строки в сумму посчиталось и значение следующей.

Следующий пример считает сумму по четырем строкам - в окно входят 2 предшествующие строки, текущая строка и одна строка, следующая за текущей:

```
select e.first_name,
       e.last_name,
       e.job,
       es.sal_date,
       es.sal_value,
       sum(es.sal_value) over (
         order by sal_value
         rows between 2 preceding and 1 following
       ) sum_sal
from emp_salary es
join employees e on e.id = es.emp_id
```

FIRST_NAME	LAST_NAME	JOB	SAL_DATE	SAL_VALUE	SUM_SAL
Александр	Сидоров	Бухгалтер	01-FEB-20	800	1650
Василий	Петров	Машинист	01-MAR-20	850	2650
Василий	Петров	Машинист	01-JAN-20	1000	3650
Александр	Сидоров	Бухгалтер	01-JAN-20	1000	3880
Евгения	Цветочкина	Бухгалтер	01-JAN-20	1030	4230
Александр	Сидоров	Бухгалтер	01-MAR-20	1200	4550
Василий	Петров	Машинист	01-FEB-20	1320	7250
Владимир	Столяров	Слесарь	01-JAN-20	3700	6220

Т.к. окно задавалось с использованием `rows`, сумма считается именно по строкам, а не по их значениям. Для первой строки в сумму были взяты данные из нее самой и следующей, т.к. предыдущих строк у нее нет. Для второй строки была лишь одна предыдущая строка, а у последней не было следующей.

Ограничения на ORDER BY

ORDER BY в аналитических функциях может использоваться только с одной колонкой, за исключением случаев, когда используется `RANGE` и окно задается одним из следующих способов:

- `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`
- `RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING`
- `RANGE BETWEEN CURRENT ROW AND CURRENT ROW`
- `RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`

Distinct. Удаление дубликатов

Оператор `DISTINCT` в `SELECT` запросах используется для удаления дублирующихся строк из выборки. В общем виде запрос выглядит следующим образом:

```
select distinct col_name1, col_name2, col_name3, ...  
from table_name
```

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');

insert into employees
values(5, 'Владимир', 'Иванов',
    to_date('01.10.1987', 'dd.mm.yyyy'), 'Сторож');

insert into employees
values(6, 'Ирина', 'Васина',
    to_date('20.03.1962', 'dd.mm.yyyy'), 'Специалист отдела кадров');

insert into employees
values(7, 'Ирина', 'Иванова',
    to_date('31.12.1990', 'dd.mm.yyyy'), 'Арт-директор');

insert into employees
values(8, 'Евгения', NULL,
    to_date('18.07.1978', 'dd.mm.yyyy'), 'Бухгалтер');

```

Итого, таблица `employees` выглядит следующим образом:

ID	FIRST_NAME	LAST_NAME	BD	JOB
1	Василий	Петров	07-OCT-90	Машинист
2	Александр	Сидоров	18-JUL-80	Бухгалтер
3	Евгения	Цветочкина	18-JUL-78	Бухгалтер
4	Владимир	Столяров	18-JUL-77	Слесарь
5	Владимир	Иванов	01-OCT-87	Сторож
6	Ирина	Васина	20-MAR-62	Специалист отдела кадров
7	Ирина	Иванова	31-DEC-90	Арт-директор
8	Евгения	-	18-JUL-78	Бухгалтер

Удаление дубликатов из одной колонки

Теперь получим список имён сотрудников:

```
select first_name
from employees
```

```
FIRST_NAME
-----
Василий
Александр
Евгения
Владимир
Владимир
Ирина
Ирина
Евгения
```

Можно заметить, что некоторые имена(Ирина, Владимир, Евгения) дублируются. Теперь посмотрим, какой мы получим результат, если применим оператор `DISTINCT` :

```
select distinct first_name
from employees
```

```
FIRST_NAME
-----
Евгения
Василий
Александр
Владимир
Ирина
```

Теперь каждое имя повторяется только один раз.

DISTINCT учитывает все колонки в строке

Теперь применим `DISTINCT` к выборке из нескольких строк:

```
select distinct id, first_name
from employees
```

ID	FIRST_NAME
7	Ирина
8	Евгения
5	Владимир
6	Ирина
2	Александр
1	Василий
3	Евгения
4	Владимир

Теперь дубликаты имен остались, и дело здесь в том, что `DISTINCT` удаляет дублирующиеся строки, а они в данном случае уникальны, так как, несмотря на повторяющиеся имена сотрудников, каждая строка имеет уникальное значение в колонке `ID`.

NULL учитывается

`distinct` учитывает NULL значения также, как и все остальные:

```
select distinct last_name
from employees
```

```
LAST_NAME
-----
Цветочкина
Столяров
Петров
Васина
Сидоров
Иванова
-
Иванов
```

Строка с пустой фамилией, как и все остальные, попала в выборку.

DISTINCT с агрегатными функциями

Когда `DISTINCT` используется с агрегатными функциями, дубликаты колонок не учитываются.

Для начала посчитаем количество не пустых имён в таблице:

```
select count(first_name) cd
from employees
```

```
CD
--
8
```

А теперь посчитаем количество **уникальных** имён в таблице:

```
select count(distinct first_name) cd
from employees
```

```
CD
--
5
```

Параллельно количеству уникальных имён мы можем получить и количество всех имён:

```
select count(distinct first_name) cd,  
       count(first_name) cnt  
from employees
```

```
CD | CNT  
-----  
5  |  8
```

Результат наглядно демонстрирует тот факт, что здесь `DISTINCT` применялся только к той функции, в которой был указан.

DISTINCT и GROUP BY

Получим количество должностей сотрудников и сгруппируем их по именам:

```
select first_name, count(job) job_cnt  
from employees  
group by first_name
```

FIRST_NAME	JOB_CNT
Евгения	2
Василий	1
Александр	1
Владимир	2
Ирина	2

Здесь должно быть всё понятно — у нас есть по два сотрудника с именем «Евгения», «Владимир» и «Ирина», у которых указаны значения в поле `JOB`.

А теперь посчитаем, сколько уникальных наименований должностей приходится на каждое имя:

```
select first_name, count(distinct job) job_cnt  
from employees  
group by first_name
```


FIRST_NAME	JOB_CNT
Евгения	1
Александр	1
Василий	1
Владимир	2
Ирина	2

Обратить следует внимание на строку с именем «Евгения» - теперь там указано число 1. Это потому, что в нашей таблице у сотрудников с таким именем одна и та же должность — бухгалтер, и количество уникальных значений здесь будет равно 1. Вообще, здесь следует помнить о [порядке выполнения запроса](#) — группировка выполняется первее, чем выборка значений, а значит и оператор `DISTINCT` будет применяться к уже сгруппированному набору данных.

DML. Изменение данных и структуры БД

Как уже говорилось ранее, `INSERT` предназначен для вставки данных в таблицу. Существует несколько вариантов его использования.

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)
```

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 необязательную колонку - написанный ранее запрос с перечислением колонок будет работать, а запрос, в котором колонки не указывались - нет.

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_chlds`. Тогда запрос вставки данных мог выглядеть следующим образом:

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

Данный оператор изменяет уже существующие данные в таблице.

Общий синтаксис выглядит следующим образом:

```
UPDATE table_1 t
SET t.column_1 = val_1,
    t.column_2 = val_2
WHERE
```

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.

Данный оператор предназначен для удаления строк из таблицы.

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;
```

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;
```


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, что в некоторых случаях она позволяет сделать все эти операции в одном выражении SQL.

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 конференции TED.

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. 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.

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.conf_date, 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.

Уже созданные таблицы можно изменять. Для этого используется команда SQL `ALTER`. Данная команда относится к группе DDL.

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);
```

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

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?';
```

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.

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` .

Логическое удаление колонок

Удаление колонок в очень больших таблицах может занять достаточно большое количество времени. В таких случаях можно для начала пометить нужные колонки как неиспользуемые:

```
alter table employees  
set unused (emp_firstname, is_out)
```

После выполнения данной команды Oracle удалит эти колонки логически, попросту пометив их как неиспользуемые. При запросе из таблицы они не будут видны, и в таблицу можно даже добавлять колонки с такими же названиями.

Чтобы удалить неиспользуемые колонки физически, используется следующий запрос:

```
alter table employees  
drop unused columns
```

Конечно, выполнять его желательно во время наименьшей нагрузки на сервер.

Переименование колонки

Переименуем колонку `birthday` в `bd` :

```
alter table employees
rename column birthday to bd
```

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 `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

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.

Переименование таблицы

Следующий запрос переименует таблицу `employees` в `emps` :

```
rename employees to emps
```

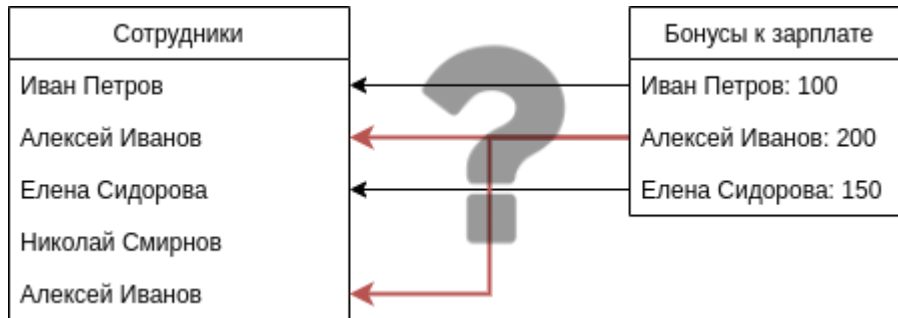
Стоит отметить, что переименование таблицы не приведет к ошибке при наличии ссылок на нее. В нашем примере таблица успешно переименуется, несмотря на дочернюю таблицу `emp_bonuses` . Внешний ключ при этом никуда не девается, в таблицу `emp_bonuses` по-прежнему нельзя добавить значения, нарушающие условия внешнего ключа.

Объекты БД

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 абсолютно 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

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_PXTWBEIMXNBUXOWCVTDQXEQKK.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
)
```

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`.

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 сделать первичным ключом, а он может быть только один. В итоге мы получим ошибку `ORA-02260: table can have only one primary key` :

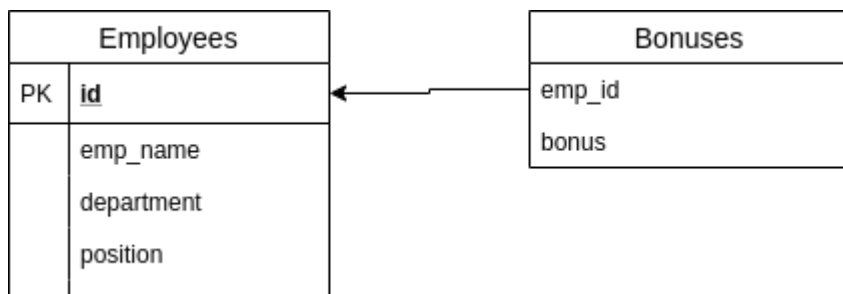
```
-- Получим ошибку при создании таблицы!  
create table month_bonuses(  
    emp_id number not null primary key,  
    month_bonus date not null primary key,  
    bonus_value number not null  
)
```

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.

id	emp_name	department	position		emp_id	bonus
1	Иван Петров	IT	QA	←	1	100
2	Алексей Иванов	SALARY	CLERK	←	2	400
3	Евгений Сидоров	SALARY	MANAGER	←	3	700
4	Екатерина Петрова	SECURITY	MANAGER		5	150
5	Алексей Иванов	SECURITY	MANAGER	←	18	300



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.

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,  
    position 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);
```

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.

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 в начале 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.

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);
```

Что такое 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.

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.

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
```

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

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 содержит 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);
```

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`.

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)`).

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`.

Что такое индексы

Индексы - это специальные объекты в БД, которые хранят в себе записи для каждого встречающегося в индексированной колонке значения. Внутреннее устройство индексов таково, что они позволяют быстро находить строки в таблице, содержащие определенные значения.

Чтобы лучше понять как работает индекс и для чего он нужен, представьте телефонный справочник, в котором абоненты расположены в случайном порядке. Чтобы найти нужного абонента, придется просматривать все записи в справочнике по порядку до тех пор, пока не встретится нужный. Но если расположить всех абонентов в алфавитном порядке, по адресам и т.д, то времени на поиски будет затрачено гораздо меньше. Для того же и используются индексы в БД - чтобы уменьшить время на поиски нужных записей в таблице.

То есть индексы - это отдельные объекты в БД, которые используются для того, чтобы ускорить поиск данных. В Oracle существует несколько типов индексов, далее здесь будут рассматриваться так называемые "B-tree" индексы. Это "классический" тип индексов, который используется на практике, и часто, когда говорят слово "индекс", то подразумевают именно его.

Создание индекса

Создадим индекс на колонку `dept_id` в таблице `employees` :

```
create index employee_dept_id_idx on employees(dept_id);
```

Здесь `employee_dept_id_idx` - это имя индекса, оно могло быть любым.

На длину имен индексов также действует ограничение в 30 символов.

При создании первичного или уникального ключа в таблице Oracle создает индексы на эти колонки автоматически.

Удаление индекса

Индексы удаляются по своим именам. Удалим индекс `employee_dept_id_idx` :

```
drop index employee_dept_id_idx;
```

Составные индексы

Индексы могут создаваться на несколько колонок. Такие индексы называются составными.

```
create index employee_name_idx on employees(first_name, last_name);
```

Составные индексы можно добавлять тогда, когда в таблице ищут данные сразу по нескольким колонкам. Порядок колонок в составном индексе важен. Однозначного правила, которое бы работало всегда, нет, но чаще всего следует добавлять колонки в индекс в порядке:

- Частоты их использования в запросах
- Количества уникальных значений, содержащихся в колонке.

Рассмотрим это на примере. Если у нас есть таблица `employees`, из которой часто получают данные по фамилии и коду должности, т.е. из таблицы часто запрашивают данные в подобном виде:

```
select *
from employees e
where e.emp_name = 'Евгений'
and e.job_code = 21
```

То в таком случае можно попробовать добавить составной индекс, состоящий из колонок `emp_name` и `job_code`:

```
create index emp_name_job_id_idx on employees(emp_name, job_id);
```

Порядок колонок в индексе не обязательно должен совпадать в порядке встречи колонок в условии запроса; Исходим мы здесь из того предположения, что:

1. По имени будут искать гораздо чаще
2. Количество уникальных имен в таблице больше, чем количество уникальных должностей.

До версии 9i порядок колонок в индексе был более важен, т.к. обязательным условием использования индекса было обращение в запросе к колонке, которая является первой в индексе. В нашем случае, если бы мы написали запрос следующим образом:

```
select *
from employees
where job_id = 10
```

индекс не использовался бы, т.к. в запросе нет фильтрации по колонке `emp_name`, которая идет первой в составном индексе.

Index skip scan

Начиная с версии 9i в Oracle появилась возможность использовать составной индекс, даже если его лидирующая колонка не используется в запросе.

Index skip scan может использоваться тогда, когда количество уникальных значений лидирующей колонки индекса относительно невелико. Опять же, решение об использовании или неиспользовании принимает оптимизатор Oracle.

По-прежнему, лучше стараться создавать составные индексы таким образом, чтобы лидирующая колонка использовалась чаще всего, так как наличие возможности использования index skip scan вовсе не означает, что он будет использован всегда.

Зачем использовать составные индексы

- Могут сократить кол-во записей за счет комбинирования колонок
- Если выбираются только колонки из индекса, то это сократит количество операций чтения, т.к. в этом случае данные будут выбраны из индекса без обращения к таблице.

Уникальные индексы

Уникальные индексы гарантируют, что значения колонок, входящих в него, будут уникальны. В этом смысле они похожи на [уникальные ключи](#) и [первичные ключи](#). Отличие в том, что индекс - это объект в базе данных, задача которого - обеспечить быстрый поиск данных. Ключи же являются отражением бизнес-требований к приложению. Мы создаем уникальный ключ не для ускорения поиска, а для того, чтобы обеспечить уникальность данных, потому что этого требует задача.

Ключи могут использовать индексы для реализации своих задач, например, первичный ключ может создать (или использовать уже имеющийся) индекс.

Уникальные индексы создаются следующим образом:

```
-- Создать уникальный индекс на колонку
-- id таблицы employees
create unique index emp_uk_idx on
employees(id);
```

Когда нужно создавать индекс

Однозначно ответить на этот вопрос нельзя, т.к. наличие индекса не означает, что он будет использоваться. Решение о том, использовать индекс или нет, принимает оптимизатор Oracle.

В общем случае, можно придерживаться следующих рекомендаций:

- Если колонка часто используется в `WHERE`
- Если колонка содержит большое количество уникальных значений по отношению к общему количеству строк в таблице.

Виртуальные колонки

Создание виртуальных колонок

Виртуальные колонки были добавлены в 11 версии Oracle.

Виртуальные колонки - это колонки, которые вычисляются на основе других колонок таблицы. Например:

```
create table cars(  
    id number primary key,  
    model varchar2(100) not null,  
    engine_volume number,  
    max_speed_km number not null,  
    max_speed_ml generated always as  
        (max_speed_km / 1.609);  
  
comment on column cars.max_speed_km is  
    'Максимальная скорость, км/ч';  
comment on column cars.max_speed_ml is  
    'Максимальная скорость, миль/ч';
```

Здесь максимальная скорость в милях вычисляется на основании данных о максимальной скорости автомобиля в километрах.

Фраза `generated always` используется для улучшения читаемости - чтобы было понятнее, что колонка будет вычисляемой. Объявить колонку `max_speed_ml` можно было и без нее:

```
max_speed_ml as (max_speed_km / 1609.344)
```

Указывать тип данных в колонке необязательно - Oracle сможет определить его на основании выражения, которое ее описывает.

Когда мы добавляем данные в таблицу, виртуальные колонки не должны нигде указываться, т.к. они будут вычислены автоматически:

```
insert into cars(id, model, engine_volume, max_speed_km)  
values(1, 'Tesla', NULL, 250);
```

Теперь посмотрим на данные в таблице:

ID	MODEL	ENGINE_VOLUME	MAX_SPEED_KM	MAX_SPEED_ML
1	Tesla	-	250	155.37

На самом деле значение в вычисляемой колонке будет другим, т.к. мы не использовали округление при его вычислении. Для удобства отображения мы оставили лишь 2 знака после запятой.

Добавление виртуальной колонки к уже существующей таблице

```
alter table cars
add(
    max_speed_m generated always as
        (max_speed_km * 1000)
);
```

Теперь в таблице хранится и скорость автомобиля в метрах в час:

```
select id,
    max_speed_km,
    round(max_speed_ml, 2) msm,
    max_speed_m
from cars
```

ID	MAX_SPEED_KM	MSM	MAX_SPEED_M
1	250	155.38	250000

Когда использовать виртуальные колонки

Виртуальные колонки позволяют не использовать [представления](#) в тех случаях, когда колонки, которые часто требуются наравне с "чистыми" данными таблицы, можно вычислить на основании этих самых чистых данных.

В таком случае ваша схема БД будет проще - за всеми необходимыми данными обращаемся напрямую к таблице.

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](#).

Мы будем использовать таблицу `dishes`, которая создается в части про [операторы сравнения](#).

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
Свинная 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
Свинная 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`.

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).

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

LEVEL

Данный псевдостолбец доступен только в рекурсивных запросах. Подробнее про него можно почитать в части про [рекурсивные запросы](#).

Транзакции

Что такое транзакции

Транзакции - это набор изменений данных, хранящихся в БД, который должен быть либо выполнен, либо отменен. До сих пор все примеры учебника, которые изменяли данные, как правило состояли из одного запроса, и все наши транзакции состояли из одного запроса. Если вы пользовались сервисом LiveSql, как мы рекомендовали, то беспокоиться о транзакциях не следовало. На самом деле, о них можно было не беспокоиться и в том случае, если примеры запускались для локально установленной БД Oracle, просто все изменения после закрытия соединения не были применены. Но в реальности изменения в данных часто представляют собой набор из нескольких запросов, и их результаты чаще всего нужно фиксировать.

Чтобы было понятнее, рассмотрим такой процесс, как копирование файлов. Предположим, какое-то количество файлов мы копируем на флешку или жесткий диск. Когда половина файлов уже была скопирована, возникла ошибка(любого рода - например, не хватило места на диске), или пользователь передумал и отменил операцию копирования. В этот момент копирование файлов прекращается, но те файлы, которые уже были скопированы, останутся на флешке/диске. Наша операция не была завершена, но состояние нашей системы не такое же, каким оно было до начала копирования.

Так вот, если бы копирование файлов было транзакцией, то после ее отмены, флешка/диск, на который мы копировали файлы, не содержала бы ни одного из копируемых файлов. А в том случае, если бы копирование прошло без прерываний, то все файлы, наоборот, гарантированно бы присутствовали в устройстве-получателе.

Рассмотрим такой процесс, как продажа автомобиля. Распишем, какие действия должны производиться при продаже:

1. Количество доступных автомобилей должно уменьшиться на 1
2. Должен быть составлен договор купли/продажи
3. Мы должны сохранить информацию о том, кто продал автомобиль, чтобы учесть это при начислении премии сотрудникам

Очевидно, что данный процесс должен быть произведен одной транзакцией, то есть все пункты этого процесса должны быть произведены, либо, в случае ошибки, или нашего собственного решения, ни одно из этих действий не возымело эффект. В противном случае, мы бы могли столкнуться с ситуацией, когда количество проданных машин не совпадает с количеством договоров купли-продажи (при условии, что на один автомобиль оформляется по одному договору), или наоборот, количество договоров о продаже авто больше, чем количество отсутствующих автомобилей, и так далее.

COMMIT. ROLLBACK

В Oracle транзакция начинается с того момента, как БД получает первый DML запрос и заканчивается либо закреплением транзакции, либо откатом всех изменений, произведенных в ней.

Транзакция продажи автомобиля может выглядеть подобным образом:

```
-- Уменьшаем количество автомобилей
update cars c
set c.count = c.count - 1
where c.id = 132;

-- Создаем договор
insert into orders(car_id, customer_name)
values(132, 'Иван Петров');

-- Сохраняем информацию о консультанте, продавшем авто
insert into consult_wrk(employee_name, car_id)
values('Петя Иванов', 132);
```

Вместе с первым запросом начинается транзакция. Все последующие запросы будут относиться к этой транзакции. Теперь главное - чтобы завершить транзакцию применением всех изменений, мы должны вызвать команду `COMMIT`, а чтобы отменить все изменения - вызывать команду `ROLLBACK`.

Почему до сих пор нигде в учебнике не использовались commit/rollback?

Как было сказано, Oracle начинает транзакцию при получении первого запроса, а точнее, первого запроса после предыдущей транзакции. Во многих IDE для коммита или отката транзакции предусмотрены отдельные элементы интерфейса - это могут быть кнопки, пункты меню и т.д. Конечно, завершить или откатить транзакцию можно и непосредственно ВЫЗОВОМ `commit` или `rollback`.

Если примеры из учебника запускались в LiveSql, то там нет необходимости делать коммиты или роллбэки, т.к. мы работаем с временной базой, и если мы зайдём на этот сервис через час, нам заново придется создавать таблицы и добавлять туда данные.

В том случае, если Oracle ставился локально, нужно было делать коммит, чтобы изменения, произведенные при запуске dml запросов остались в базе.

Больше о транзакциях будет рассказано при рассмотрении языка программирования PL/SQL, где будет подробнее рассмотрено, когда и кто должен отправлять БД команды `COMMIT` и `ROLLBACK`.

Транзакции в многопользовательской среде

База данных редко работает только с одним клиентом. Чаще всего БД отвечает на запросы от большого числа клиентов, и многие из этих запросов приходят к ней в одно и то же время. Что будет, если во время выполнения нашей транзакции, описанной выше, другой клиент выполнит следующий запрос:

```
-- Получить количество договоров в системе  
select count(*)  
from orders
```

Ответ будет зависеть лишь от того, как была завершена транзакция. Если она была завершена вызовом `COMMIT`, то запрос вернет количество с учетом того договора, который был нами создан, а если транзакция завершилась вызовом `ROLLBACK`, либо вообще еще не завершилась (запросы еще выполняются, либо они выполнились, но ни `ROLLBACK`, ни `COMMIT` не вызывались), то новый договор не будет учтен в общем количестве, т.к. другие сессии еще знают о существовании нового договора.

Здесь нужно сделать уточнение. Даже если наша транзакция будет завершена вызовом `COMMIT`, запрос у другого клиента не будет брать в расчет изменения, произведенные нашей транзакцией, если наша транзакция была завершена позднее, чем запрос в другой сессии начал свое выполнение.

Итого

Все изменения в БД производятся в пределах транзакций. Транзакция может состоять как из одного запроса, так и из нескольких.

`commit` фиксирует изменения в базе данных, `rollback` - отменяет.

```

-- Обновляем признак активности пользователя
-- ADMIN
update users u
set u.is_active = 'N'
where u.user_name = 'ADMIN';

-- Фиксируем транзакцию. Изменения сохраняются в БД
commit;

-- Теперь опять меняем флаг is_active на значение
-- 'N'. Т.к. изменения, произведенные предыдущим
-- запросом, были зафиксированы, этот запрос
-- начинает новую транзакцию.
update users u
set u.is_active = 'Y'
where u.user_name = 'ADMIN';

-- У пользователя ADMIN флаг is_active
-- по-прежнему равен 'N', т.к. действие
-- транзакции было отменено вызовом
-- rollback
rollback;

```

Транзакции могут состоять из набора изменений:

```

-- Уменьшаем количество автомобилей
update cars c
set c.count = c.count - 1
where c.id = 132;

-- Создаем договор
insert into orders(car_id, customer_name)
values(132, 'Иван Петров');

-- Сохраняем информацию о консультанте, продавшем авто
insert into consult_wrk(employee_name, car_id)
values('Петя Иванов', 132);

-- Изменения, произведенные тремя запросами, будут
-- сохранены в базе данных
commit;

```

Введение

PL/SQL - императивный язык программирования, интегрированный в БД "Oracle". Он дает возможность:

1. Создавать хранимые процедуры и функции
2. Использовать конструкции, характерные для императивных языков программирования: ветвление, циклы, обработка исключений и др.

Особенности

Пожалуй, самой отличительной особенностью является его тесная интеграция с БД "Oracle" и языком SQL. PL/SQL может работать только внутри БД "Oracle" и не может быть использован за её пределами.

Язык поддерживает процедурный и объектно-ориентированный(в его современном понимании) подходы.

PL/SQL нужно использовать, когда:

- Нужно скрыть функционал за общим интерфейсом(инкапсуляция)
- Возможностей SQL недостаточно
- Реализация на PL/SQL будет более сопровождаемой

Коротко по каждому из пунктов.

Инкапсуляция

Мы можем создать функцию, либо хранимую процедуру, после чего различные клиенты смогут воспользоваться ими, не зная деталей реализации функционала.

Недостаточно возможностей SQL

Здесь лучше подойдет пример. Предположим, нам нужно реализовать логику:

1. Если значение некоего параметра, назовем его `quantity`, больше 100, то:
2. Удалить некоторые строки из таблицы A.
3. Иначе:
4. Изменить некоторые строки в таблице B

Пункты 2 и 3 легко реализуются на чистом SQL. Связки 1-2 и 3-4 также можно реализовать на SQL, но условное выполнение 2 либо 4 возможно только при добавлении возможностей ветвления PL/SQL.

Сопровождаемость

В один монстрообразный SQL-запрос можно засунуть много сложной логики, и в целом так и нужно поступать, но лишь до тех пор, пока ты уверен, что он:

- Понятен
- Будет понятен через полгода
- Тестируем

Если есть сомнения, лучше разделить функционал на более простые части(используя хранимые процедуры, например) и реализовать каждую из них отдельно.

ОСНОВЫ PL/SQL

Переменные, константы. Простые типы данных

Рассмотрим типы, которые позволяют хранить три сущности - числа, строки и даты.

Числа

Тип для хранения чисел в PL/SQL используется тот же тип данных, что и при создании таблиц. Почитать про него можно [здесь](#).

Существуют и другие типы для работы с числами, например `PLS_INTEGER`, `SIMPLE_INTEGER` и др. Они будут рассмотрены позже.

Даты

Для дат используются те же типы, что и в SQL - `Date` и `Timestamp`.

Строки

Для хранения строк используются типы `varchar2` и `clob`. Последний расшифровывается как *Char Large Object*, и предназначен для хранения очень больших объемов текстовой информации.

Стоит отметить разницу типов `Varchar2` в SQL и PL/SQL: в первом случае максимальная длина ограничена 32767 байтами, во втором - 4000 байтами.

Логический тип

Логический тип данных используется для хранения значений `True`(истина) или `False`(ложь). В PL/SQL такой тип данных называется `BOOLEAN`. Он еще будет рассмотрен подробнее далее в учебнике.

Переменные

Переменные в PL/SQL должны объявляться в секции объявления переменных. Так как мы пока рассмотрели только [анонимные блоки](#), работать будем с ними.

Напишем анонимный блок, в котором задействуем переменные:

```
declare
    age number;
    name varchar2(100 char);
begin
    age := 45;
    name := 'Alex';

    dbms_output.put_line(age);
    dbms_output.put_line(name);
end;
/
```

Вывод на экран:

```
45
Alex
```

Выше мы объявили две переменные, `age` и `name`, которым позднее присвоили значения.

Присваивание значений переменным допускается сразу после их определения:

```
declare
    age number := 45;
    name varchar2(100 char) := 'Alex';
begin
    null;
end;
/
```

Следует обратить внимание на то, что при объявлении строкового типа нужно указывать размер, как в примере выше.

Команда `null` ничего полезного не делает, в PL/SQL она используется для того, чтобы подставлять ее в те места, где компилятор требует наличия команды. Так и в нашем примере - исполняемый блок должен что-то иметь внутри себя.

После объявления переменных с ними можно производить различные операции

- арифметические, логические, булевы и так далее. Не будем все рассматривать детально, так как сложностей здесь возникать не должно.

declare

```
age number := 10;  
name varchar2(100 char) := 'Alex';
```

begin

```
age := age + 20;  
name := 'Hello, ' || name;  
  
dbms_output.put_line(age);  
dbms_output.put_line(name);
```

end; /

Вывод:

```
30  
Hello, Alex
```

Константы

Константа - это переменная, значение которой нельзя изменять.

Пример объявления константы:

```
declare  
    MIN_AGE constant number := 21;  
begin  
    dbms_output.put_line('Минимальный возраст для входа: ' || MIN_AGE);  
end;  
/
```

Вывод:

```
Минимальный возраст для входа: 21
```

Константе должно присваиваться значение сразу после ее объявления.

Значение констант нельзя изменять, но во всем остальном они работают точно также, как и переменные:

```

declare
    DEFAULT_NAME VARCHAR2(10) := 'Anonymous';
    MIN_AGE constant number := 21;
    user_name varchar2(50);
    user_age number;
begin
    -- Присваиваем значение константы переменной
    user_name := DEFAULT_NAME;
    user_age := MIN_AGE;

    dbms_output.put_line(user_name);
    dbms_output.put_line(user_age);

    -- Увеличиваем значение переменной на значение константы
    user_age := user_age + MIN_AGE;

    dbms_output.put_line(user_age);

end;
/

```

Вывод:

```

Anonymous
21
42

```

Какую проблему помогают решить функции

Общее описание функций уже было дано в предыдущей части, добавим лишь то, что функции - это один из способов сделать наш код более простым, понятным, компактным, а также избежать повторного написания однотипного кода.

Пример создания простой функции

Предположим, что мы работаем над системой, в которой часто приходится считать размер скидок в зависимости от стоимости купленного товара. Размер скидки интересует нас с точностью до двух знаков после запятой.

Вот так может выглядеть код для вычисления нашей скидки:

```
declare
    -- Сумма скидки
    l_discount number;
    -- Стоимость товара
    l_price number := 350;
begin
    l_discount := round(l_price * (10 / 100), 2);

    dbms_output.put_line(l_discount);
end;
/
```

В результате на экран выведется размер скидки - 35.

Мы посчитали размер скидки по одному товару. А что, если нам нужно посчитать скидку по трем товарам в отдельности? Пишем код:

```

declare
    l_discount_1 number;
    l_discount_2 number;
    l_discount_3 number;
    l_price_1 number := 350;
    l_price_2 number := 100;
    l_price_3 number := 25;
begin
    l_discount_1 := round(l_price_1 * (10 / 100), 2);
    l_discount_2 := round(l_price_2 * (10 / 100), 2);
    l_discount_3 := round(l_price_3 * (10 / 100), 2);

    dbms_output.put_line(l_discount_1);
    dbms_output.put_line(l_discount_2);
    dbms_output.put_line(l_discount_3);
end;
/

```

Вывод:

```

35
10
2.5

```

Может показаться, что все выглядит вполне себе хорошо - необходимая логика реализована, в чем проблема?

Представим, что в какой-то момент начальство говорит вам о том, что размер скидки должен округляться до одного знака после запятой, а не до двух? Нам придется изменить это в трех местах. То же относится и к размеру скидки - что, если в определенный момент времени скидка станет не 10, а 20 или 30 процентов?

Помимо всего прочего, код выше выглядит сложно, без необходимой на то причины. Мы могли бы создать функцию, которая будет считать размер этой скидки и возвращать её нам:

```

create function getDiscount(
    pprice number
) return number
is
begin
    return round(pprice * (10 / 100), 2);
end;
/

```

Если запустить код выше, то в текущей схеме БД станет доступна функция `getDiscount`. Используем ее, чтобы модифицировать предыдущий пример:

```
declare
    l_discount_1 number;
    l_discount_2 number;
    l_discount_3 number;
    l_price_1 number := 350;
    l_price_2 number := 100;
    l_price_3 number := 25;
begin
    l_discount_1 := getDiscount(l_price_1);
    l_discount_2 := getDiscount(l_price_2);
    l_discount_3 := getDiscount(l_price_3);

    dbms_output.put_line(l_discount_1);
    dbms_output.put_line(l_discount_2);
    dbms_output.put_line(l_discount_3);
end;
/
```

Вывод:

```
35
10
2.5
```

Теперь, чтобы сделать изменение в логике подсчета скидки, это нужно сделать всего в одном месте, и сразу же размер скидки будет считаться по новым правилам.

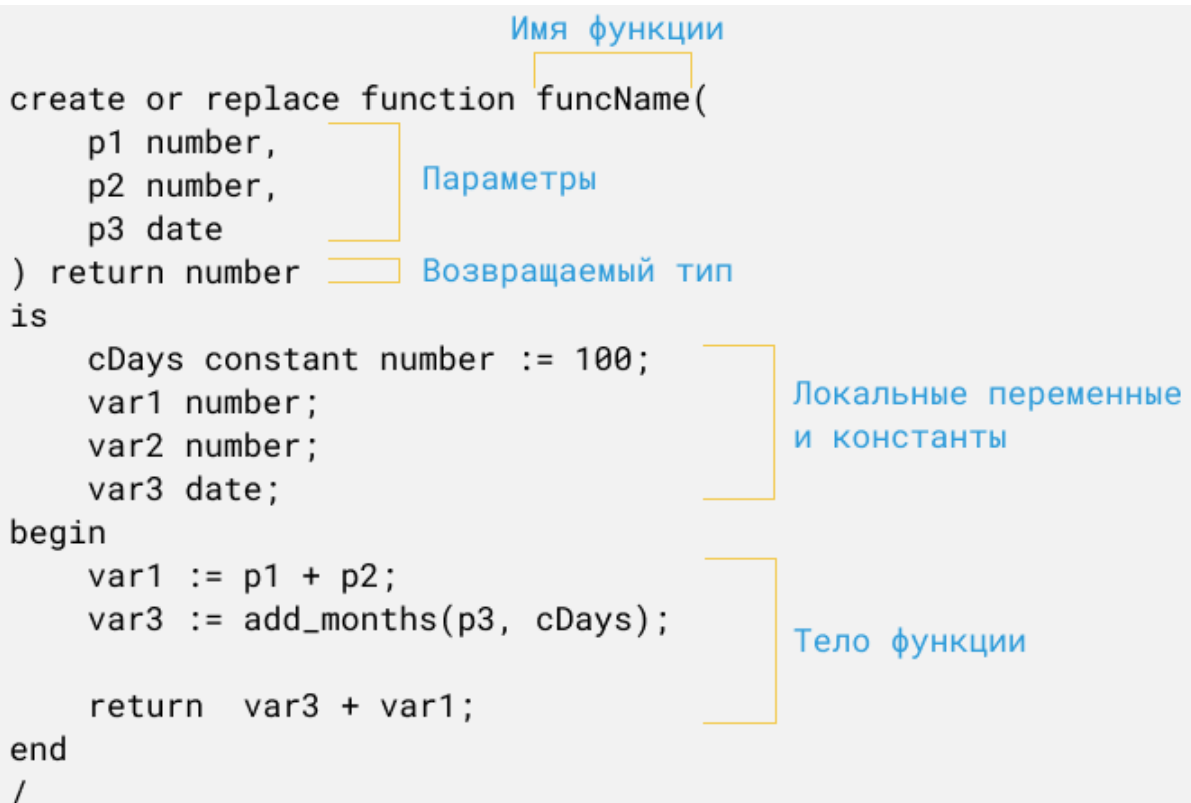
По-умолчанию, при попытке создать функцию с именем, которое уже существует в текущей схеме, Oracle выдаст ошибку. Для того, чтобы этого не возникало, при описании функции используется конструкция `create or replace`.

Рассмотрим подробнее синтаксис создания функции в PL/SQL.

```
-- Используем create or replace,
-- чтобы заменить предыдущую функцию
-- с таким же именем на новую
create or replace function funcName(
    param1 number
) return number
is
begin
    return 1;
end;
/
```

Основные элементы которые нужно указать при создании функции это:

- Имя функции
- Параметры(могут отсутствовать)
- Тип возвращаемого значения
- Тело функции
- Команда Return



The diagram illustrates the syntax of a SQL function with several annotations in Russian:

- Имя функции** (Function name): Points to `funcName`.
- Параметры** (Parameters): Points to the parameter list `p1 number, p2 number, p3 date`.
- Возвращаемый тип** (Return type): Points to `return number`.
- Локальные переменные и константы** (Local variables and constants): Points to the declarations `cDays constant number := 100;`, `var1 number;`, `var2 number;`, and `var3 date;`.
- Тело функции** (Function body): Points to the code block between `begin` and `end`, including `var1 := p1 + p2;`, `var3 := add_months(p3, cDays);`, and `return var3 + var1;`.

```
create or replace function funcName(
    p1 number,
    p2 number,
    p3 date
) return number
is
    cDays constant number := 100;
    var1 number;
    var2 number;
    var3 date;
begin
    var1 := p1 + p2;
    var3 := add_months(p3, cDays);

    return var3 + var1;
end
/
```

Вообще, существует большое количество дополнительных синтаксических конструкций при создании функций, как например указание того, что функция является табличной или детерминированной, но на данном этапе мы не будем их рассматривать, так как они используются лишь в специфичных случаях.

Выше мы создали функцию с именем funcName, которая принимает один параметр с типом number и возвращает значение типа number(в нашем примере это всегда число 1).

Если параметров несколько, они перечисляются через запятую, например:

```
create function funcName(  
    p1 number,  
    p2 varchar2  
) return boolean  
is  
begin  
    return false;  
end;  
/
```

Команда return используется для того, чтобы вернуть значение из функции. Сразу после этого работа функции завершается, то есть код, который указан после `return`, до которого дошло выполнение, никогда не будет вызван:

```
create function funcName() return bool  
is  
begin  
    return false;  
    -- Функция всегда будет возвращать False,  
    -- следующий return никогда не будет выполнен  
    return true;  
end;  
/
```

Следует обратить внимание, что при описании параметров функции мы не указываем размерность типов.

Следующее описание функции вызовет ошибку:

```
create function funcName(  
    p1 number(10,2),  
    p2 varchar2(100 char)  
) return boolean  
is  
begin  
    return false;  
end;  
/
```


Функции могут быть и без аргументов:

```
-- Возвращает процент скидки
create function discValue() return number is
begin
    return 10;
end;
/
```

Функции с ошибками все равно создаются

Если мы попытаемся создать функцию с ошибкой, то она все равно создастся в схеме БД, но использовать ее будет нельзя, т.к. она будет иметь статус invalid.

Создадим следующую функцию:

```
create function invalidFunc() return number is
begin
    return false;
end;
/
```

Здесь у нас ошибка, т.к. функция должна возвращать тип number, а мы возвращаем значение типа boolean. Oracle выдаст что-нибудь в этом духе:

```
Errors: FUNCTION INVALIDFUNC
Line/Col: 1/22 PLS-00103: Encountered the symbol ")" when
expecting one of the following:

    <an identifier> <a double-quoted delimited-identifier>
current delete exists prior
```

Ок, пробуем создать функцию без ошибок:

```
create function invalidFunc() return number is
begin
    return 1;
end;
/
```

И в итоге получаем сообщение `ORA-00955: name is already used by an existing object`, что означает что в схеме уже есть объект с таким именем. В нашем случае - это функция `invalidFunc`, которая была создана во время предыдущей попытки.

Поэтому, чтобы заменить предыдущий вариант функции на новый, нужно использовать конструкцию `create or replace`.

Удаление функции

Чтобы удалить функцию из схемы, используется команда `drop function`:

```
drop function invalidFunc;
```

Локальные переменные

Не всегда функции бывают такими простыми, что их логика помещается в одну команду `return`.

Для реализации более сложной бизнес-логики или для улучшения "читаемости" кода лучше можно использовать локальные переменные функции

- переменные, которые доступны для использования только внутри функции и нигде больше. Они объявляются между ключевым словом `IS` и `BEGIN`.

Попробуем переписать нашу функцию `getDiscount` с использованием локальных переменных:

```
create or replace function getDiscount(  
    pprice number  
) return number  
is  
    discPerc constant number := 10;  
    discount number;  
begin  
    discount := round(pprice * (discPerc / 100), 2);  
  
    return discount;  
end;  
/
```

Использование функций в SQL запросах

Функции PL/SQL можно использовать в SQL запросах. Для демонстрации этого создадим таблицу с товарами и ценами:

```
create table products(  
    id number primary key,  
    name varchar2(300 char),  
    price number  
);  
  
insert into products  
values(1, 'Фотоаппарат', 1340);  
  
insert into products  
values(2, 'Клавиатура', 55);  
  
insert into products  
values(3, 'Планшет', 800);
```

Теперь получим список товаров и размер скидки на них:

```
select name, price, getDiscount(price) disc  
from products
```

Результат:

```
NAME          PRICE  DISC  
=====
```

Фотоаппарат	1340	134
Клавиатура	55	5.5
Планшет	800	80

```
=====
```

Понятие схемы БД

Схема - это совокупность всех объектов некоего пользователя, называемого владельцем схемы. Сюда входят таблицы, индексы, представления, триггеры, всё-всё-всё. То есть мы должны понимать, что когда мы говорим про функции или процедуры в PL/SQL, они находятся внутри схемы.

Мы не будем перечислять все возможные типы объектов, которые могут храниться в схеме, перечислим лишь несколько. Все нижеописанные объекты будут рассмотрены по отдельности далее в учебнике, здесь будет дана информация для первичного ознакомления с ними.

Функции

Функции представляют собой именованные блоки кода, которые можно вызывать повторно. Отличительной особенностью функций является то, что по своей натуре они должны возвращать некое значение.

Простой пример - функция сложения двух чисел `mysum`. Вот пример того, как она могла бы быть описана:

```
function Mysum(a number,
               b number
) return number
is
begin
    return a + b;
end;
```

И далее, везде, где мы хотим получить сумму двух чисел, мы можем использовать уже созданную функцию:

```
begin
    a := sum(5, 10);
    b := sum(10, a);
    -- Функции могут использоваться в качестве
    -- параметров других функций
    c := sum(sum(a, b), 20);
end;
```

Также, функции pl/sql можно использовать в обычных SQL запросах, например вот так:

```
select Mysum(l.age, 2) new_age
from users l
```

Процедуры

Процедуры, так же как и функции, представляют собой именованные блоки кода, которые можно(и нужно) использовать повторно. Отличаются от функций тем, что вызов процедуры не возвращает значение, как функция, но несмотря на это, через процедуры можно возвращать значения - через так называемые `out` параметры.

Процедуры нельзя использовать в SQL запросах.

Процедура или функция

Процедуры очень похожи на функции, и не всегда можно понять, что же использовать. Тем не менее, в большинстве случаев, рекомендуется использовать функции лишь для того, чтобы что-то вычислять и возвращать это значение, но не изменять состояние данных в БД. Процедуры же, наоборот, рекомендуется использовать в тех случаях, когда они изменяют данные в БД.

Вышеупомянутая функция `mysum`, например, ничего не изменяет, она лишь считает сумму двух чисел и возвращает ее.

Какой код можно было бы использовать в процедуре? Ну, например, процедура добавления нового пользователя в систему. Добавление нового пользователя приводит к тому, что в таблицу(или даже несколько) БД будет добавлена запись, некоторые, возможно будут обновлены и т.д.

Итого, **лучше создавать функцию**, если:

- Она только производит вычисления
- Она только возвращает данные, хранящиеся в БД

Лучше создавать процедуру, если:

- Данные в БД будут изменяться
- Данные не будут изменяться, но мы не нуждаемся в возвращении значения после ее вызова

Конечно, эти советы не являются обязательными, и от них вполне можно отступаться, более того, достаточно часто в коде PL/SQL можно встретить функции, которые изменяют данные, например, подобного рода:

```
-- добавить пользователя и вернуть его id
function addUser(login varchar2) return number;
```

Здесь функция `addUser` добавляет нового пользователя в систему и сразу возвращает его `id`.

Главное, пожалуй, здесь то, чтобы действия, производимые подпрограммами, которые вы пишете, должны быть понятными; достичь этого во многом помогают правильные наименования, как в примере выше - вот если бы функция называлась бы просто `user`, было бы гораздо сложнее понять, что она делает.

Процедуры и функции, которые хранятся в БД, еще часто называют хранимыми процедурами и хранимыми функциями. Тем не менее, чаще всего используют термин "хранимая процедура" для обозначения как процедур, так и функций.

Пакеты

Пакеты - это такие объекты, которые позволяют сгруппировать набор функций, процедур, констант, переменных и типов в одном модуле. Они не только помогают улучшить организацию кода, но и позволяют реализовать инкапсуляцию - т.е. скрыть внутреннюю реализацию своего функционала. Достигается это благодаря тому, что пакет имеет *спецификацию* и *тело*. Для использования внутри схемы доступны всё, что описано в спецификации пакета, но все, что описано в теле пакета и не описано в спецификации, нельзя использовать вне тела пакета.

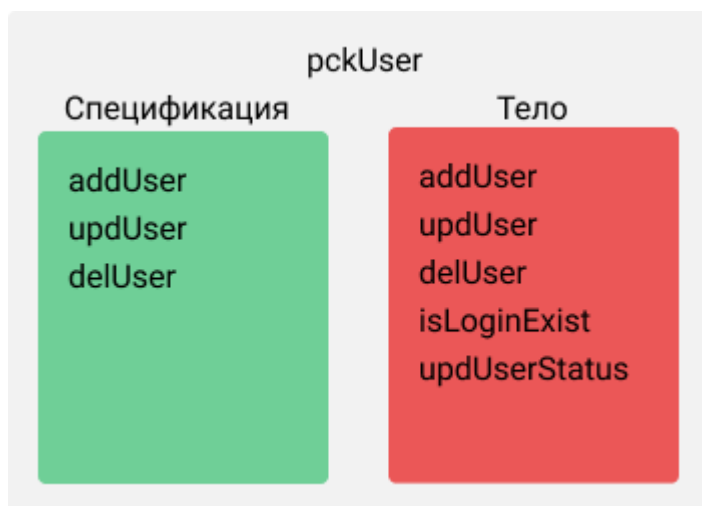
Приведем простой пример: предположим, у нас есть пакет под названием `pckUser`. В спецификации пакета даны описания функций добавления, удаления и редактирования пользователя:

```
procedure insUser(login varchar2);
procedure updUser(id number, login varchar2);
procedure delUser(id number);
```

Эти три процедуры будут доступны внутри схемы, и вызвать их можно будет подобным образом:

```
begin
    pckUser.insUser('username');
end;
```

Схематично структура пакета может выглядеть подобным образом:



"Снаружи" из пакета доступны только три вышеупомянутые процедуры. Но внутри пакета, помимо реализации публичных процедур, также созданы еще две функции - `isLoginExist` и `updUserStatus`. Они могут использоваться как вспомогательные функции/процедуры внутри других подпрограмм пакета, но не будут доступны для вызова вне тела пакета.

Триггеры

Триггеры - это специальные процедуры, которые запускаются автоматически при определенных действиях с таблицами.

То есть, когда мы запускаем, к примеру, следующий запрос:

```
update users
set is_active = 0
where id = 10
```

В базе данных может *автоматически* запуститься триггер, который будет при деактивации пользователя заполнять значение в колонке с датой деактивации.

Триггеры могут выполняться до или после вставки, изменения или удаления данных в таблице. Существуют и более сложные виды триггеров, но они будут рассматриваться в отдельной теме, посвященной триггерам.

Циклы используются для повторения какого-либо действия несколько раз.

Loop

Простейший цикл в PL/SQL выглядит так:

```
loop
    -- какое-либо действие
end loop;
```

Данный цикл будет выполнять код между `loop` и `end loop` бесконечно. Для того, чтобы завершить цикл, можно использовать команду `exit` :

```
declare
    i number := 5;
begin
    loop
        if i = 0 then
            exit;
        end if;

        dbms_output.put_line(i);
        i := i - 1;
    end loop;
end;
```

Результат:

```
5
4
3
2
1
```

Переменная `i` отвечает за оставшееся количество повторений. Такие переменные в программировании ещё называют *счётчиками цикла*.

Для более удобного выхода из цикла можно использовать конструкцию `exit when` :


```

declare
    i number := 5;
begin
    -- Такой же цикл, как и в предыдущем примере, только
    -- читается гораздо лучше
    loop
        exit when i = 0;

        dbms_output.put_line(i);
        i := i - 1;
    end loop;
end;

```

`Exit` можно использовать только внутри цикла, внутри обычного блока его использование приведёт к ошибке:

```

-- Ошибка!
begin
    exit;
end;

```

For

Цикл `for` удобно использовать для автоматического изменения счётчика цикла:

```

begin
    for i in 1..5
    loop
        dbms_output.put_line(i);
    end loop;
end;

```

Результат:

```

1
2
3
4
5

```

Здесь не использовались ни проверки на необходимость выйти из цикла, ни изменение счётчика цикла - конструкция `for` всё сделала за нас. Мы обращаемся к переменной `i` - она видима только внутри тела цикла, и недоступна вне его.

В `for` указывается нижняя граница счётчика цикла, и верхняя. Изменение производится увеличением счётчика на единицу. Нижняя граница всегда должна быть меньше либо равной верхней границе:

```
-- Выведет:
-- -5
-- -4
-- -3
-- -2
-- -1
-- 0
-- 1
begin
  for i in -5..1
  loop
    dbms_output.put_line(i);
  end loop;
end;
```

Если верхняя граница равна нижней, цикл выполнится ровно один раз:

```
-- Выведет одну строку:
-- 1
begin
  for i in 1..1
  loop
    dbms_output.put_line(i);
  end loop;
end;
```

В сторону уменьшения работать не будет - тело цикла `for i in 3..1` не выполнится ни разу.

Стоит также обратить внимание, что мы не объявляли переменную `i` в секции `declare` - она создаётся автоматически. Более того, если мы объявим переменную вне цикла `for`, то после завершения цикла внешняя переменная останется без изменений:

```

declare
    i number := 10;
begin
    for i in 1..4
    loop
        null;
    end loop;

    -- Выведет 10, а не 4, как можно было бы подумать
    dbms_output.put_line(i);
end;

```

While

Цикл `while` будет работать до тех пор, пока условие, указанное после него, истинно:

```

declare
    i number := 5;
begin
    while i > 0
    loop
        dbms_output.put_line(i);
        i := i - 1;
    end loop;
end;

```

Вложенные циклы

Циклы могут быть вложенными. В целом, здесь нет ничего сложного, обратим внимание лишь на несколько моментов.

Вложенные циклы `for` могут иметь счётчики с одним и тем же именем:

```

begin
  for i in 1..2
    loop
      dbms_output.put_line('i1= ' || i);
      for i in 1..2
        loop
          dbms_output.put_line('i2= ' || i);
        end loop;
      end loop;
    end loop;
  end;
end;

```

Выведет:

```

i1= 1
i2= 1
i2= 2
i1= 2
i2= 1
i2= 2

```

Как видно, значения счётчика из внешнего цикла не перепуталось со значением счётчика из внутреннего. Это произошло потому, что мы обращались к счётчикам внутри тела цикла, к которому они относятся. Если мы перенесём вывод счётчика внешнего цикла внутрь тела внутреннего, работать не будет:

```

begin
  for i in 1..2
    loop
      for i in 1..2
        loop
          dbms_output.put_line('i1= ' || i);
          dbms_output.put_line('i2= ' || i);
        end loop;
      end loop;
    end loop;
  end;
end;

```

```
i1= 1
i2= 1
i1= 2 -- А должно быть 1
i2= 2
i1= 1
i2= 1
i1= 2
i2= 2
```

Решить эту проблему помогают метки блоков:

```
begin
  <<main>>
  for i in 1..2
    loop
      <<child>>
      for i in 1..2
        loop

          dbms_output.put_line('i1= ' || main.i);
          dbms_output.put_line('i2= ' || child.i);

        end loop;
      end loop;
    end loop;
  end;
```

Результат:

```
i1= 1
i2= 1
i1= 1
i2= 2
i1= 2
i2= 1
i1= 2
i2= 2
```

Синтаксис для обращения к переменным с использованием меток выглядит так: `имя метки.переменная`. При использовании вложенных циклов часто используют отдельный вариант синтаксиса завершения цикла, чтобы было понятнее, какой именно цикл заканчивает свою работу:

```

begin
  <<main>>
  for i in 1..2
    loop
      <<child>>
      for i in 1..2
        loop
          dbms_output.put_line('i1= ' || main.i);
          dbms_output.put_line('i2= ' || child.i);
          -- указываем имя идентификатора блока
        end loop child;
      end loop main;
    end;
  end;

```

Continue. Переход на следующую итерацию цикла

Команда `continue` используется для перехода к следующей итерации цикла. Часть тела цикла после данной команды не будет выполнена.

```

-- Выведет пять строк с числами от 1 до 5.
-- Числа с приплюсованной двойкой не будут выведены,
-- так как их вывод находится после continue.
begin
  for i in 1..5
    loop
      dbms_output.put_line(i);
      continue;
      dbms_output.put_line(i + 2);
    end loop;
  end;

```

Как и с командой `exit`, есть и более удобный вариант этой команды - `continue when`:

```

begin
  for i in 1..10
    loop
      continue when i mod 2 = 0;
      dbms_output.put_line(i);
    end loop;
  end;

```

Результат:

```
1
3
5
7
9
```

В начале каждой итерации мы проверяем остаток от деления на два, и если он равен нулю, сразу переходим к следующей итерации, таким образом выводим только нечётные числа.

Вложенные и именованные блоки

В [прошлый раз](#) был рассмотрен самый простой вариант блоков в PL/SQL - анонимные блоки.

В этой части мы рассмотрим возможность вкладывать блоки внутрь других блоков, а также давать блокам имена.

Вложенные блоки

```
declare
    l_a number := 10;
begin
    -- Начало вложенного блока
    declare
        l_b number;
    begin
        l_b := 20;
        dbms_output.put_line(l_a);
        dbms_output.put_line(l_b);
    end;
end;
/
```

Вложенные блоки могут обращаться ко всем объектам, которые были объявлены во внешних блоках. Именно поэтому в примере выше мы свободно можем обращаться из к переменной `l_a`, которая была объявлена во внешнем блоке.

Обратное неверно - мы не можем обращаться из внешнего блока к объектам, которые были объявлены во внутреннем блоке:


```

declare
    l_a number := 10;
begin
    declare
        l_b number;
    begin
        l_b := 20;
        dbms_output.put_line(l_b);
        dbms_output.put_line(l_a);
    end;

    -- Мы не можем обращаться к переменной l_b
    if l_b = 10 then
        dbms_output.put_line('l_b равно 10')
    else
        dbms_output.put_line('l_b не равно 10');
    end if;
end;
/

```

Метки блоков

Мы можем именовать блоки метками, чтобы можно было отличить несколько объектов с одним и тем же именем во внешних блоках:

```

<&lt;main>>
declare
    l_a number := 10;
begin
    <&lt;child>>
    declare
        l_a number := 20;
    begin
        dbms_output.put_line(main.l_a);
        dbms_output.put_line(child.l_a);
    end;
end;
/

```

В примере выше во внешнем и внутреннем блоках объявлены переменные `l_a`, и используя метки блоков `<<main>>` и `<<child>>`, мы можем указывать, какую же именно переменную использовать - из внешнего, либо из внутреннего блока.

По возможности, подобных случаев следует избегать, так как они усложняют чтение программы.

Первая программа на PL/SQL

Итак, напишем нашу первую программу на PL/SQL. Но перед этим создадим простую таблицу `users`, в которой будем хранить список пользователей некой системы:

```
create table users(  
    id number primary key,  
    login varchar2(60) not null,  
    sign_date date default sysdate not null,  
    is_active number(1) default 1 not null  
);  
  
insert into users  
values(1, 'UserA', to_date('21.01.2019', 'dd.mm.yyyy'), 1);  
  
insert into users  
values(2, 'UserB', to_date('15.07.2017', 'dd.mm.yyyy'), 1);  
  
insert into users  
values(3, 'UserC', to_date('02.10.2015', 'dd.mm.yyyy'), 1);
```

Теперь приведем текст программы:

```
begin  
    update users  
    set is_active = 0  
    where sign_date < to_date('01.01.2016', 'dd.mm.yyyy');  
end;  
/
```

Данный код делает неактивными всех пользователей, которые были зарегистрированы ранее, чем первое января 2016 года.

Программа очень простая, но зато она наглядно демонстрирует важнейшую особенность PL/SQL - **тесную интеграцию с SQL**. Любой SQL запрос может быть вызван из PL/SQL, и это абсолютно нормально и естественно. На самом деле, большую часть кода PL/SQL, как правило, составляют именно SQL-запросы.

Здесь важно понимать, почему это PL/SQL программа, а не простой SQL-запрос. Причиной является тот факт, что запрос заключен в [анонимный блок](#).

Внутри блока может находиться сколько угодно запросов. Например, следующая программа делает неактивными пользователей, зарегистрировавшихся раньше 2016 года, и добавляет нового пользователя:

```
begin
    update users
    set is_active = 0
    where sign_date < to_date('01.01.2016', 'dd.mm.yyyy');

    insert into users
    values(4, 'UserD', SYSDATE);
end;
/
```

Условное ветвление используется тогда, когда нужно выполнить разные действия в зависимости от условий. Для этого в PL/SQL используется конструкция `if`.

If

```
declare
    l_name varchar2(100) := 'Admin';
begin
    if l_name = 'Admin' then
        dbms_output.put_line('User is Admin');
    end if;
end;
/
```

В примере выше описан простейший вариант использования `if`. Если имя пользователя равно строке "Admin", мы выводим соответствующее сообщение.

В конструкции `if` может быть несколько условий:

```
declare
    l_name varchar2(100) := 'Admin';
begin
    if l_name = 'Admin' or l_name = 'TempAdmin' then
        dbms_output.put_line('User is Admin');
    end if;
end;
/
```

Результат:

```
User is Admin
```

В случае с логическими переменными, проверять на равенство true или false не обязательно:

```

declare
    is_admin boolean := true;
begin
    if is_admin then
        dbms_output.put_line('Admin');
    end if;
end;
/

```

Общий принцип работы конструкции if таков: Если условие между идентификаторами if и then принимает истинное значение, выполняется код, находящийся между идентификаторами then и end if;

При проверке условий PL/sql использует так называемое "ленивое вычисление" - когда части условия вычисляются по порядку, и если можно сделать вывод о значении всего условия, дальнейшее вычисление не производится. Например, в следующем условии будет вычислен только первый предикат(`2 > 3`):

```

begin
    if (2 > 3) and (3 < 4) and (5 > 4) then
        dbms_output.put_line('True');
    else
        dbms_output.put_line('False');
    end if;
end;

```

Выражение будет истинным только тогда, когда каждая его часть будет истинным. Первое выражение у нас ложное, а значит, результат проверки всего условия вернёт False.

If...else

```

declare
    l_name varchar2(100) := 'Max';
begin
    if l_name = 'Admin' then
        dbms_output.put_line('User is Admin');
    else
        dbms_output.put_line('User is not Admin');
    end if;
end;
/

```

Результат:

```
User is not Admin
```

Использование конструкции `else` позволяет выполнять некий код в том случае, если условие принимает ложное значение. В примере выше, значение переменной `l_name` не равно строке "Admin", поэтому выполняется код, который находится между `else` и `end if`;

if...elsif

```
declare
    l_name varchar2(20) := 'Max';
begin
    if l_name = 'Admin' then
        dbms_output.put_line('Пользователь админ');
    elsif l_name = 'Alex' then
        dbms_output.put_line('Пользователь Alex');
    elsif l_name = 'Kat' then
        dbms_output.put_line('Пользователь Kat');
    else
        dbms_output.put_line('Неизвестный пользователь');
    end if;
end;
/
```

Вывод на экран:

```
Неизвестный пользователь
```

Использование `elsif` позволяет выполнить еще одну проверку условия в том случае, если предыдущее условие было ложным. В самом конце добавлен `else` - код в этом блоке будет выполнен только в том случае, когда ни одно из предыдущих условий в `if` не было истинным, но он может и отсутствовать.

Взаимодействие PL/SQL и SQL.

Переключение контекста

Движки SQL и PL/SQL

Когда Oracle получает команду выполнить какой-либо SQL запрос, он передаёт эту работу SQL движку. Следует взять за правило, что SQL движок является более быстрым, чем PL/SQL движок - все его функции встроены в ядро БД и написаны на языке C. Это **основной** язык для работы с данными в БД, он лучше оптимизирован. PL/SQL код, в свою очередь, выполняется PL/SQL движком. Он добавляет возможность процедурного программирования, но, в свою очередь, он медленнее, чем SQL, несмотря на то, что в Oracle постоянно работают над его улучшением.

Мы знаем, что из SQL можно вызывать PL/SQL функции, а в PL/SQL свободно использовать SQL запросы, и большинство задач решаются с использованием как SQL, так и PL/SQL. Давайте разберёмся, как взаимодействуют между собой эти движки и как частое переключение между ними может повлиять на производительность приложений.

Итак, когда нужно выполнить SQL запрос, в работу включается SQL движок. Что будет, если в нашем запросе мы будем вызывать PL/SQL функцию, например вот так:

```
select get_discount(o.id) disc_value,
       o.num,
       o.order_date
from orders o
```

Помимо номера и даты заказа, мы получаем ещё и размер скидки по нему, который считается PL/SQL функцией `get_discount`. Так как это PL/SQL функция, Oracle придётся использовать PL/SQL движок для её выполнения. Если представить, что в таблице заказов 100 строк, то это означает, что выполнение в PL/SQL движок придётся передать 100 раз.

Подобная передача выполнения из одного движка в другой называется *переключением контекста* (англ. *Context switch*), и это то, чего следует по возможности избегать, так как переключение из одного движка в другой занимает дополнительное время. Но здесь есть одна особенность - вызов PL/SQL из SQL даёт большую нагрузку, чем вызов SQL из PL/SQL кода, так что первое, чего следует избегать - это вызова PL/SQL функций из SQL запросов.

Давайте посмотрим на обратный пример - вызов SQL запроса из PL/SQL процедуры:


```

create or replace procedure set_phone_notes(
    pstart date,
    pend date
) is
begin
    update app_users au
    set au.note = (select listagg(ui.phone, ';')
        from user_phones ui
        where ui.user_id = au.id)
    where au.reg_date between pstart and pend
    and au.notes is null;
end;

```

Данная процедура обновляет поле с примечанием (`note`) на номера телефонов пользователя, перечисляемые через точку с запятой. Данные обновляются для пользователей, зарегистрировавшихся за указанный период, и у которых поле с примечанием не пустое.

Как будет происходить переключение контекста, если мы вызовем эту процедуру?

```

declare
    l_start date := sysdate;
    l_end date := sysdate - 10;
begin
    set_phone_notes(l_start, l_end);
end;
/

```

В данном примере мы вызываем PL/SQL процедуру, которая внутри себя вызывает один SQL запрос, передавая его исполнение SQL движку. Таким образом, у нас производится одно переключение контекста.

С другой стороны, наш код мог бы выглядеть следующим образом:

```

create or replace procedure set_phone_notes(
    pstart date,
    pend date
) is
begin
    update app_users au
    set au.note = get_phones_agg(au.id)
    where au.reg_date between pstart and pend
    and au.notes is null;
end;

```

Здесь получение строки с номерами телефонов по заданному пользователю вынесено в отдельную функцию `get_phones_agg`. Кажется бы, код стал выглядеть лучше в плане читаемости программистом - теперь проще разобраться, что же за значение мы устанавливаем, но вызывая PL/SQL функцию, мы добавляем по одному переключению контекста для каждой из обновляемой строки, причём это переключение из SQL в PL/SQL.

Вообще, довольно сложно бывает выбрать правильный баланс между монстрообразным SQL запросом или последовательностью вызова PL/SQL функций, выполняющих более мелкие задачи по отдельности, но лучше всегда стараться использовать SQL. Здесь нельзя не процитировать известную мантру Тома Кайта:

1. Вы должны сделать это одним SQL выражением
2. Если вы не можете сделать это одним SQL выражением, используйте PL/SQL
3. Если вы не можете сделать это на PL/SQL, попробуйте хранимые процедуры на Java
4. Если вы не можете сделать это на хранимых процедурах Java, используйте хранимые процедуры на C
5. Если вы не можете сделать это с помощью хранимых процедур на C, вам следует серьёзно подумать о том, зачем вы вообще это делаете.

DRY

Известный принцип **DRY** (*Dont Repeat Yourself*) [\[Wikipedia\]](#) гласит, что мы должны стремиться к максимальному переиспользованию кода, и в разрезе темы переключения контекста слепое следование данному принципу может негативно сказаться на скорости выполнения кода.

Допустим что у нас уже определены функции для получения определённых частей информации по заказам. Пусть это будут следующий набор:

- `get_order_status` - получить статус заказа
- `get_discount_value` - получить размер скидки
- `get_order_owner` - получить владельца заказа

В какой-то момент времени перед нами возникает задача получения всех этих данных по определённому набору заказов. Зная, что повторное использование кода - это хорошо, мы пишем следующий запрос, который решает поставленную задачу:

```
select o.id,
       o.num,
       o.get_order_status(o.id) status,
       o.get_order_owner(o.id) owner,
       o.get_phones_agg(o.id) phones
from orders o
where o.order_date between :pstart and :pend
```

Но в результате получаем крайне неэффективный запрос, который для каждой из обрабатываемых строк делает лишние переключения контекста, хотя в данном случае их легко избежать, используя чистый SQL:

```
select o.id,
       o.num,
       o.status,
       o.owner,
       (select listagg(ui.phone, ';')
        from user_phones ui
        where ui.user_id = o.owner) phones
from orders o
where o.order_date between :pstart and :pend
```

Поля `status` и `owner` вообще являются колонками в таблице `orders`, список телефонов легко получается с использованием [коррелированного подзапроса](#). Всё, теперь никаких переключений контекста, задача решена с использованием чистого SQL. Конечно, повторного использования кода по возможности следует избегать. Но когда речь идёт об получении набора данных из большого количества строк, следует использовать SQL настолько, насколько *вы можете*.

Нужно отметить, что нет ничего плохого в том, чтобы иметь функции, которые возвращают какое-то одно значение из БД, ведь мы вполне можем столкнуться с ситуацией, когда нам и нужно только одно это значение:

```
declare
    l_order_owner number;
    l_order_num number := 10;
begin
    l_order_owner := get_owner(l_order_num);
    -- ... Код, который как-либо использует l_order_owner
end;
```

Как бороться со сложностью

Изучайте SQL

Иногда бывает так, что кажется, будто задача не может быть решена на чистом SQL, но на самом деле вполне себе может. Разработчик может попросту не знать о каких-то возможностях SQL, функциях или приёмах их использования. Некоторые возможности SQL могли быть отсутствовать в предыдущей версии БД, разработчик привык к ней, а после обновления на более новую версию не изучал новые возможности или ещё не научился их применять. Одним словом, нужно больше всего времени уделять изучению именно SQL.

Используйте views

Если вы чувствуете, что SQL запрос с которым вы работаете, или его часть, становятся сложными, можно вынести его в [представление](#), чтобы облечить понимание общей бизнес-логики.

Используйте subquery factoring

Это можно отнести к совету об изучении SQL, но мы упомянем эту возможность отдельно. Напомним, [subquery factoring](#) позволяет разбить запрос на именованные подзапросы, сильно облегчив понимание того, что же делает весь запрос в целом. В какой-то степени это как разбиение кода на функции в императивных языках, только в SQL запросе.

При изучении любого языка программирования очень важно иметь возможность выводить информацию на экран.

Несмотря на то, что PL/SQL - язык, интегрированный в БД Oracle, он также имеет возможность вывода информации на экран. Для этого используется процедура

```
dbms_output.put_line .
```

Пример вызова:

```
begin
    dbms_output.put_line('Hello, World');
    dbms_output.put_line(23);
    dbms_output.put_line('Hello, ' || 'World');
    dbms_output.put_line(sysdate);
end;
/
```

Вывод программы:

```
Hello, World
23
Hello, World
29-AUG-21
```

Как видно из примера, `dbms_output.put_line` позволяет выводить строки, числа и даты - основные типы в Oracle.

Практически все среды разработки поддерживают вывод через `dbms_output`, включая [Live SQL](#).

Любой PL/SQL код, состоит из блоков. Для начала рассмотрим анонимные блоки. Их структура следующая:

```
DECLARE  
  
BEGIN  
  
END;
```

Секция объявления переменных является необязательной и может отсутствовать. В таком случае анонимный блок представляет собой блок вида:

```
BEGIN  
  
END;
```

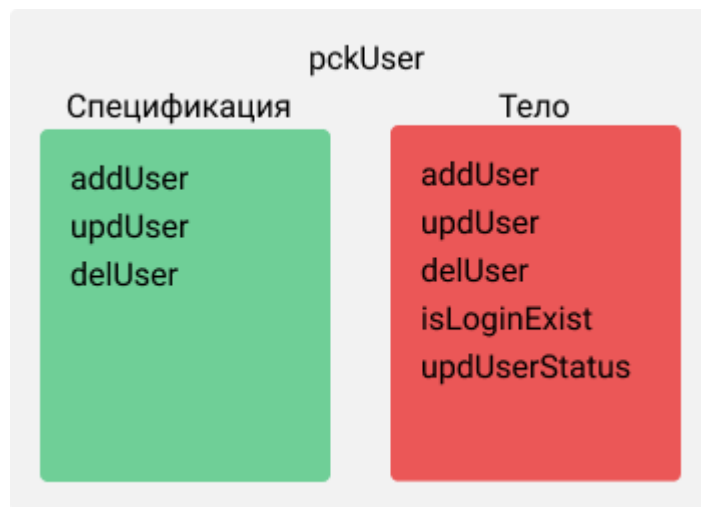
В данной части будут рассмотрены пакеты - основная сущность при разработке на PL/SQL. Пакеты используются для группировки функционала в именованные модули с возможностью разделения интерфейса и реализации. На самом деле, мы уже сталкивались с ними, когда рассматривали вывод на экран с использованием `dbms_output.put_line`.
`dbms_output` - это пакет, а `put_line` - процедура, объявленная в данном пакете.

Структура пакета

Пакеты как правило состоят из **спецификации** и **тела**. Можно создать пакет без тела, только со спецификацией, такой вариант использования тоже будет рассмотрен.

Спецификация пакета - это то, к чему можно обращаться при работе с пакетом. В спецификации могут быть объявлены типы, переменные, константы, сигнатуры процедур и функций.

Тело пакета содержит в себе код, необходимый для реализации спецификации. Также он может содержать всё то же, что спецификация - переменные, типы, константы и проч. Всё, что содержится в теле, но не описано в спецификации, недоступно для использования внешними модулями. Здесь можно провести аналогию с публичными и приватными модификаторами доступа в ООП языках(например, как `private` и `public` в Java).



Создание пакета

Общий синтаксис создания спецификации пакета выглядит так:

```
create package pck_utils as
-- Specification code
end pck_utils;
```

Команда выше создаст пакет с названием `pck_utils`. Если пакет с таким именем уже существует, будет выброшена ошибка, и новый пакет не создастся. Чтобы заменить уже существующий пакет, используется команда `create or replace`. На практике чаще всего используют именно её:

```
create or replace package pck_utils as
-- Specification code
end pck_utils;
```

Тело пакета создаётся следующим образом:

```
create or replace package body pck_utils as
-- Specification code
end pck_utils;
```

Давайте создадим пакет и наполним его каким-нибудь функционалом:

```
create or replace package pck_date_utils as

-- Возвращает максимальную дату,
-- используемую в системе
function maxdate return date;

-- Возвращает минимальную дату,
-- используемую в системе
function mindate return date;

-- Добавляет указанное количество недель к
-- указанной дате. Для того, чтобы отнять
-- недели, нужно передать отрицательное число
function add_weeks(
    pdate date,
    pweeks number
) return date;

end pck_date_utils;
```

Это была спецификация пакета. Теперь создадим тело:


```

create or replace package body pck_date_utils as

function maxdate return date is
begin
    return to_date('4000.01.01', 'yyyy.mm.dd');
end;

function mindate return date is
begin
    return to_date('1800.01.01', 'yyyy.mm.dd');
end;

function add_weeks(
    pdate date,
    pweeks number
) return date
is
begin
    return pdate + (7 * pweeks);
end;

end pck_date_utils;

```

Теперь мы можем обращаться ко всему, что объявлено в спецификации пакета в нашем коде. Обращение к содержимому пакета осуществляется в виде `имя_пакета.объект` (под объектом понимается всё, что объявлено в спецификации):

```

begin
    dbms_output.put_line(to_char(pck_date_utils.mindate, 'yyyy.mm.dd'));
    dbms_output.put_line(to_char(pck_date_utils.maxdate, 'yyyy.mm.dd'));
    dbms_output.put_line(to_char(pck_date_utils.add_weeks(sysdate, 3), 'yyyy.mm.dd'));
end;

```

Вывод:

```

1800.01.01
4000.01.01
2022.03.30

```

Удаление пакета

Удаление производится командой `drop package` :

```
drop package pck_date_utils
```

Компиляция пакета

В учебнике мы предполагаем, что текст пакета должен храниться в *файлах*, как SQL скрипты. Спецификацию и тело, как правило, хранят в разных файлах, с расширениями ".sql". Можно использовать любые другие расширения, например ".pks" для спецификации и ".pkb" для тела - подобные расширения также часто используются. Когда нужно изменить пакет, код в этих файлах меняется, после чего эти скрипты перезапускаются.

Если нужно произвести перекомпиляцию пакета, который уже создан, без его изменения, можно воспользоваться командой `alter package` :

```
alter package pck_utils compile package;
```

```
alter package pck_utils compile specification;
```

```
alter package pck_utils compile body;
```

Изменение пакета

Изменение пакета производится путём перекомпиляции его спецификации или тела(используя `create or replace`).

Если спецификация пакета не изменяется, а только её реализация, то её пересоздавать не нужно. Если же меняется и спецификация, то придётся перекомпилировать и спецификацию, и тело.

Пакеты без тела

Можно создать пакет, который не будет иметь тела. Как правило, это пакеты, которые содержат публичные константы, типы или переменные:

```

create or replace package pck_user_gl as

-- Статусы пользователей
active constant number := 0;
deleted constant number := 1;
paused constant number := 2;

end pck_user_gl;

```

В данном пакете содержатся константы для описания статусов пользователей. Теперь мы можем обращаться к статусам, объявленным в пакете:

```

-- Выведет три строки:
-- 0
-- 2
-- 1
begin
    dbms_output.put_line(pck_user_gl.active);
    dbms_output.put_line(pck_user_gl.paused);
    dbms_output.put_line(pck_user_gl.deleted);
end;

```

Напомним, что константы и переменные PL/SQL нельзя использовать в SQL запросах, но они могут быть использованы в другом PL/SQL коде.

Перегрузка процедур и функций

Внутри пакета можно объявить несколько функций или процедур с одним и тем же именем, но с разной сигнатурой. Подобная возможность в языках программирования называется перегрузкой (overloading). Простейший пример перегруженных функций (процедур) - это `dbms_output.put_line`. Мы можем вызывать данную процедуру как со строками, так и с датами или числами. Создадим свой пакет для вывода информации на экран, только с более коротким именем процедуры, для удобной работы:

```

create or replace package pck_output as

procedure print(v varchar2);
procedure print(v number);
procedure print(v date);

end pck_output;

```

В спецификации мы объявили три разных сигнатуры - несмотря на то, что имена у них одинаковые, они отличаются типами принимаемых аргументов. Теперь создадим тело пакета:

```
create or replace package body pck_output as

procedure print(v varchar2) is
begin
    dbms_output.put_line(v);
end;

procedure print(v number) is
begin
    dbms_output.put_line(v);
end;

procedure print(v date) is
begin
    dbms_output.put_line(v);
end;

end pck_output;
```

Скомпилируем пакет и проверим, как он работает:

```
-- Выведет строки:
-- 19-MAR-22
-- 34.23
-- Hello, World
declare
    v1 date := sysdate;
    v2 number := 34.23;
    v3 varchar2(20) := 'Hello, World';
begin
    pck_output.print(v1);
    pck_output.print(v2);
    pck_output.print(v3);
end;
```

Функции можно перегружать, если они отличаются

- Количеством аргументов
- Типами аргументов

Имена аргументов при этом не важны - не получится создать две процедуры вида:

```
procedure proc1(name varchar2);  
procedure proc1(username varchar2);
```

Но зато получится создать такие процедуры:

```
procedure proc1(name varchar2);  
  
-- Отличается от предыдущей количеством аргументов  
procedure proc1(name varchar2, trim: boolean);  
  
-- отличается от предыдущей типами аргументов  
procedure proc1(name varchar2, trim: number);
```

Сессии и состояния

Каждая сессия в БД работает со своей копией пакета в памяти, что означает, что состояние переменных пакета *локально для сессии, использующей его*. Рассмотрим простой пример:

```
create or replace package pck_test as  
  
    current_number number := 10;  
  
end pck_test;
```

Пакет содержит одну переменную `current_number` со значением по-умолчанию, равным 10. Предположим, с пакетом будут работать две сессии, А и Б:

Сессия А:

```
begin  
    dbms_output.put_line(pck_test.current_number);  
end;
```

Сессия Б:

```
begin  
    dbms_output.put_line(pck_test.current_number);  
end;
```

Результат будет одинаковым в двух сессиях:

```
10
```

Теперь изменим значение переменной в первой сессии:

```
-- Сессия А
begin
    pck_test.current_number := 20;
end;
```

После чего выведем содержимое переменной в двух сессиях:

```
-- Сессия А:
begin
    dbms_output.put_line(pck_test.current_number);
end;
```

```
-- Сессия Б:
begin
    dbms_output.put_line(pck_test.current_number);
end;
```

И получим следующий результат:

Сессия А:

```
20
```

Сессия Б:

```
10
```

Как видно, изменения переменной, произведённые в первой сессии, не повлияли на значение той же переменной пакета во второй сессии.

Если в пакете объявлена хотя бы одна переменная, константа, или курсор(не важно где, в теле пакета или в его спецификации), то пакет обладает *состоянием*. Когда Oracle создаёт экземпляр пакета, в сессии также хранится и состояние. Изменения в состоянии пакета сохраняются на всё время работы сессии(за исключением пакетов, объявленных как `SERIALLY_REUSABLE` , это будет рассмотрено в отдельной части). Но состояние может быть сброшено, если пакет был перекомпилирован.

Сброс состояния означает, что изменения, произведённые с переменными или курсорами, будут утеряны, и оракл выбросит ошибку с сообщением о том, что состояние пакета было сброшено. Способы уменьшения вероятности сброса состояния будут рассмотрены в отдельной части, посвящённой продвинутой работе с пакетами.

Порядок загрузки пакета в память

Помимо создания экземпляра пакета в памяти при первоначальном обращении к нему, Oracle производит его инициализацию, состоящую из следующих шагов:

- Присваивание первоначальных значений публичным константам
- Присваивание первоначальных значений публичным переменным
- Запуск блока инициализации

Блок инициализации

Блок инициализации добавляется в конце тела пакета между ключевым словом `begin` и конструкцией `end package_name`, и как правило используется для присваивания начальных значений переменным пакета. Сам блок является необязательным. Рассмотрим пример пакета с блоком инициализации:

```
-- Спецификация пакета
create or replace package pck_init as

    init_val number;

    procedure say_hello;

end pck_init;
/

-- Тело пакета
create or replace package body pck_init as

    procedure say_hello
    is
    begin
        dbms_output.put_line('Привет, Мир!');
    end;

--Секция инициализации
begin
    dbms_output.put_line('Инициализация пакета');
    init_val := 23;
end pck_init;
```

Теперь вызовем процедуру `say_hello` несколько раз подряд:

```
begin
    pck_init.say_hello();
    pck_init.say_hello();
    pck_init.say_hello();
end;
```

Результат:

```
Инициализация пакета
Привет, Мир!
Привет, Мир!
Привет, Мир!
```


Как видно, при первом обращении к пакету был вызван блок инициализации, причём до выполнения процедуры `say_hello`. При последующих обращениях к пакету инициализация не производится. Инициализация выполняется при любом первичном обращении к пакету, это не обязательно должна быть процедура или функция:

```
begin
    -- Выведет две строки:
    -- Инициализация пакета
    -- 23
    dbms_output.put_line(pck_init.init_val);
end;
```

Считается правилом хорошего тона производить инициализацию всех переменных именно в блоке инициализации, а не при объявлении переменных. Одна из причин - тот факт, что ошибки, которые могут возникнуть при инициализации переменных, можно отловить только здесь. Исключения будут рассмотрены позже, но для быстрого ознакомления приведём пример:

```
create or replace package pck_test is

    min_age number(2) := 123;
    default_name varchar2(3 char) := 'User';

end pck_test;
/
```

При создании пакета не будет выданы никаких сообщений об ошибке. Но если мы попробуем вывести на экран значение `default_name`:

```
begin
    dbms_output.put_line(pck_test.default_name);
end;
```

То получим ошибку(Во время выполнения!) `ORA-06502: PL/SQL: numeric or value error: number precision too large ORA-06512`. А при использовании блока инициализации ошибку можно отловить:

```

create or replace package pck_test is

min_age number(2);
default_name varchar2(3 char);

end pck_test;
/

create or replace package body pck_test is

begin
    min_age := 123;
    default_name := 'User';

exception
    when value_error then
        min_age := 18;
        default_name := 'NIL';
end pck_test;
/

```

В этот раз мы имеем возможность отловить все ошибки на стадии инициализации пакета и предпринять нужные меры. Посмотрим, как это работает:

```

-- Выведет две строки:
-- NIL
-- 18
begin
    dbms_output.put_line(pck_test.default_name);
end;

```

Ещё о функциях в пакетах

Forward declaration

Порядок приватных функций/процедур в теле пакета имеет значение. Если функция `A` использует функцию `B`, то функция `B` к тому моменту должна быть объявлена, то есть находится выше в коде тела:

```
create or replace package pck_test as
```

```
procedure print_hello;
```

```
end pck_test;
```

```
create or replace package body pck_test as
```

```
procedure print_hello is
```

```
begin
```

```
    dbms_output.put_line(get_hello_message);
```

```
end;
```

```
function get_hello_message return varchar2
```

```
is
```

```
begin
```

```
    return 'Hello, World!';
```

```
end;
```

```
end pck_test;
```

Здесь процедура `print_hello` выводит на экран текст сообщения, который возвращает функция `get_hello_message`, но сама процедура объявлена раньше, чем функция. При компиляции тела мы получим ошибку `PLS-00313: 'GET_HELLO_MESSAGE' not declared in this scope` - она не объявлена на момент своего вызова.

Решить эту проблему можно несколькими способами. Во-первых, можно поместить функцию `get_hello_message` выше процедуры:

```

create or replace package body pck_test as

function get_hello_message return varchar2
is
begin
    return 'Hello, World!';
end;

procedure print_hello is
begin
    dbms_output.put_line(get_hello_message);
end;

end pck_test;

```

Во-вторых, можно добавить функцию `get_hello_message` в спецификацию пакета, и тогда порядок внутри тела не будет ни на что влиять:

```

create or replace package pck_test as

procedure print_hello;

function get_hello_message return varchar2;

end pck_test;

```

Но объявлять всё в спецификации - тоже не выход; Некоторые функции не должны быть доступны для вызова всеми желающими. В таком случае можно использовать так называемую *Forward declaration* - разделить описание функций от их реализации:

```

create or replace package body pck_test as

-- Объявляем функцию, но не указываем её реализацию
function get_hello_message return varchar2;

procedure print_hello is
begin
    -- Ошибки не будет, функция get_hello_message
    -- уже объявлена
    dbms_output.put_line(get_hello_message);
end;

-- Реализация функции get_hello_message
function get_hello_message return varchar2
is
begin
    return 'Hello, World!';
end;

end pck_test;

```

Вызов функций в SQL запросах

Все функции, которые используются в SQL запросах, должны быть созданы на уровне схемы, то есть либо быть созданными как отдельные функции(`create function`), либо быть объявленными в спецификации пакета. Рассмотрим на примере:

```

create or replace package pck_test as

procedure print_hello;

end pck_test;

```

```

create or replace package body pck_test as

function get_hello_message return varchar2
is
begin
    return 'Hello, World!';
end;

procedure print_hello is
    l_msg varchar2(50);
begin
    select get_hello_message() into l_msg
    from dual;

    dbms_output.put_line(l_msg);
end;

end pck_test;

```

При компиляции тела пакета мы получим ошибку PLS-00231: function 'GET_HELLO_MESSAGE' may not be used in SQL . Это потому, что мы вызываем функцию из SQL, но данная функция не объявлена в спецификации пакета. Если мы добавим её сигнатуру в спецификацию, то все будет работать:

```

create or replace package pck_test as

procedure print_hello;

function get_hello_message return varchar2;

end pck_test;

```

```

begin
    -- Выведет "Hello, World!"
    pck_test.print_hello;
end;

```

Процедуры в PL/SQL

Пример создания простой процедуры

```
create or replace procedure validate_age(  
    page number  
)  
is  
begin  
    if page < 18 then  
        dbms_output.put_line('Вам должно быть 18 или больше');  
    else  
        dbms_output.put_line('Всё хорошо');  
    end if;  
end;
```

Вызовем процедуру с несколькими параметрами:

```
begin  
    validate_age(17);  
    validate_age(40);  
end;  
/
```

Вывод:

```
Вам должно быть 18 или больше  
Всё хорошо
```

Как видно, особых отличий от создания функций нет. Основное отличие - процедуры не возвращают значений в таком виде, как это делают функции (через вызов `return`).

Общепринятые различия между функциями и процедурами

См. [Различия между функциями и процедурами](#).

IN, OUT, IN OUT параметры

Каждый параметр функции или процедуры может иметь модификатор, отвечающий за характер данного параметра:

- `IN` - входной параметр
- `OUT` - выходной параметр
- `IN OUT` - входной и выходной параметр

По умолчанию все параметры являются входными, так что явно указывать `IN` необязательно. Такие параметры нельзя изменять в теле процедуры или функции.

`OUT` -параметры, наоборот, предназначены для того, чтобы быть измененными. Часто их используют в процедурах для того, чтобы вернуть некоторое значение(или даже несколько значений).

```
create or replace procedure get_const_values(  
    min_date out date,  
    max_date out date,  
    default_date out date  
)  
is  
begin  
    min_date := to_date('1800-01-01', 'yyyy-mm-dd');  
    max_date := to_date('4021-01-01', 'yyyy-mm-dd');  
    default_date := sysdate;  
end;  
/
```

После этого выполним следующий код:

```
declare  
    l_min_date date;  
    l_max_date date;  
    l_default_date date;  
begin  
    get_const_values(l_min_date, l_max_date, l_default_date);  
  
    dbms_output.put_line(l_min_date);  
    dbms_output.put_line(l_max_date);  
    dbms_output.put_line(l_default_date);  
end;  
/
```

Вывод:


```
01-JAN-00
01-JAN-21
05-DEC-21
```

OUT параметры не могут иметь значений по умолчанию:

```
create or replace procedure get_const_values(
    min_date out date := to_date('1800-01-01', 'yyyy-mm-dd'),
    max_date out date := to_date('4021-01-01', 'yyyy-mm-dd')
)
is
begin
    null;
end;
/
```

В результате функция будет создана с ошибкой `OUT and IN OUT formal parameters may not have default expressions`.

Как определять ошибки при создании хранимых процедур, будет рассказано в отдельной части.

IN OUT параметры доступны для чтения внутри хранимой процедуры, но в то же время они доступны и для изменения.

```
create or replace procedure get_const_values(
    min_date in out date,
    max_date in out date
)
is
begin
    -- Читаем значения переменных
    dbms_output.put_line(min_date);
    dbms_output.put_line(min_date);

    -- Изменяем значения переменных
    min_date := to_date('3000-02-02', 'yyyy-mm-dd');
    max_date := to_date('3001-02-02', 'yyyy-mm-dd');

end;
/
```

Запустим эту процедуру и выведем на экран значение переменных после её выполнения:

```
declare
    l_min date := to_date('1900.01.01', 'yyyy-mm-dd');
    l_max date := to_date('1900.01.01', 'yyyy-mm-dd');
begin
    get_const_values(l_min, l_max);

    dbms_output.put_line(l_min);
    dbms_output.put_line(l_max);
end;
/
```

Вывод:

```
01-JAN-00
01-JAN-00
02-FEB-00
02-FEB-01
```

Как можно заметить, значения переменных были изменены после вызова процедуры.

Важной особенностью `OUT` и `IN OUT` параметров является то, что они должны быть переданы в виде переменных, задать их значения литералом нельзя:

```
create or replace procedure myproc(
    page out number
)
is
begin
    dbms_output.put_line(page);
end;
/
```

И теперь попробуем вызвать процедуру, используя литерал, а не переменную:

```
begin
    myproc(12);
end;
/
```

В результате получим ошибку `expression '12' cannot be used as an assignment target`.

Удаление процедуры

Чтобы удалить процедуру из схемы, используется команда `drop procedure` :

```
drop procedure myproc;
```

Обработка ошибок

EXCEPTION блок

Обработка ошибок производится в блоке `exception` :

```
begin
    -- Код
exception
    -- Обработка ошибок
    when .... then .....;
    when .... then .....;
    when .... then .....;
end;
```

Ошибки отлавливаются в пределах блока `begin-end` . Работает это так:

1. Сначала выполняется код между `begin` и `exception`
2. Если ошибок не произошло, тогда секция между `exception` и `end` игнорируется
3. Если в процессе выполнения кода происходит ошибка, выполнение останавливается и переходит в блок `exception` .
4. Если в блоке находится обработчик для исключения, вызывается код после `then`
5. Если обработчик не найден, исключение выбрасывается за пределы блока `begin-end`

Пример блока с обработчиком исключений:

```
declare
    l_val number;
begin
    select 1 into l_var
    where 2 > 3;
exception
    when no_data_found then
        dbms_output.put_line('Нет данных');
    when dup_val_on_index then
        dbms_output.put_line('Такая строка уже есть');
end;
```

Предопределённые ошибки

Ошибки обрабатываются по их имени, поэтому часть наиболее частых ошибок в PL/SQL уже предопределена, как например вышеуказанные `no_data_found` и `dup_val_on_index` .

Ниже показан их список и в каких случаях ошибка может возникнуть.

Ошибка	Когда возникает
ACCESS_INTO_NULL	Попытка присвоить значение атрибуту неинициализированного объекта.
CASE_NOT_FOUND	В выражении <code>CASE</code> не нашлось подходящего условия <code>When</code> , и в нём отсутствует условие <code>Else</code> .
COLLECTION_IS_NULL	Попытка вызвать любой метод коллекции(за исключением <code>Exists</code>) в неинициализированной вложенной таблице или ассоциативном массиве, или попытка присвоить значения элементам неинициализированной вложенной таблицы или ассоциативного массива.
CURSOR_ALREADY_OPEN	Попытка открыть уже открытый курсор. Курсор должен быть закрыт до момента его открытия. Цикл <code>FOR</code> автоматически открывает курсор, который использует, поэтому его нельзя открывать внутри тела цикла.
DUP_VAL_ON_INDEX	Попытка вставить в таблицу значения, которые нарушают ограничения, созданные уникальным индексом. Иными словами, ошибка возникает, когда в колонки уникального индекса добавляются дублирующие записи.
INVALID_CURSOR	Попытка вызова недопустимой операции с курсором, например закрытие не открытого курсора.
INVALID_NUMBER	Ошибка приведения строки в число в SQL запросе, потому что строка не является числовым представлением (В PL/SQL коде в таких случаях выбрасывается <code>VALUE_ERROR</code>). Также может возникнуть, если значение параметра <code>LIMIT</code> в выражении <code>Bulk collect</code> не является положительным числом.
LOGIN_DENIED	Попытка подключиться к БД с неправильным логином или паролем.

Ошибка	Когда возникает
NO_DATA_FOUND	Выражение <code>SELECT INTO</code> не возвращает ни одной строки, или программа ссылается на удалённый элемент во вложенной таблице или неинициализированному объекту в ассоциативной таблице. Агрегатные функции в SQL, такие как AVG или SUM, всегда возвращают значение или null. Поэтому, <code>SELECT INTO</code> , которое вызывает только агрегатные функции, никогда не выбросит <code>NO_DATA_FOUND</code> . Выражение <code>FETCH</code> работает так, что ожидает отсутствия строк в определённый момент, поэтому ошибка также не выбрасывается.
NOT_LOGGED_ON	Обращение к БД будучи неподключенным к ней
PROGRAM_ERROR	Внутренняя проблема в PL/SQL.
ROWTYPE_MISMATCH	Курсорные переменные, задействованные в присваивании, имеют разные типы.
SELF_IS_NULL	Попытка вызвать метод неинициализированного объекта.
STORAGE_ERROR	Переполнение памяти или память повреждена.
SUBSCRIPT_BEYOND_COUNT	Попытка обратиться к элементу вложенной таблицы или ассоциативного массива по индексу, который больше, чем количество элементов в коллекции.
SUBSCRIPT_OUTSIDE_LIMIT	Попытка обратиться к элементу коллекции по индексу(например, -1) вне допустимого диапазона.
SYS_INVALID_ROWID	Ошибка конвертации строки в rowid.
TIMEOUT_ON_RESOURCE	Возникает при ожидании доступности ресурса.
TOO_MANY_ROWS	Выражение <code>SELECT INTO</code> возвращает более одной строки.
VALUE_ERROR	Арифметическая ошибка, ошибка конвертации, или превышение размерности типа. Может возникнуть, к примеру, если в переменную с типом <code>number(1)</code> попытаться присвоить значение <code>239</code> .
ZERO_DIVIDE	Попытка деления на ноль.

Объявление собственных ошибок

Можно объявлять собственные исключения, давая им названия, которые полнее раскрывают их суть.

```
declare
    -- Объявление собственного исключения,
    -- которое мы выбрасываем, если значение заработной
    -- платы ниже дозволенного минимума.
    exc_too_low_salary exception;
    l_salary number := 100;
begin
    if l_salary < 200 then
        -- Бросаем ошибку.
        raise exc_too_low_salary;
    end if;

exception
    when exc_too_low_salary then
        dbms_output.put_line('Обработчик исключения');
end;
```

Область видимости собственного исключения в данном случае - блок, в котором оно объявлено. Вне этого блока обработать исключение не получится.

Для более удобной работы с собственными исключениями их можно вынести в отдельный пакет:

```
create or replace pck_hr_errors is

    -- Объявляем исключения в спецификации пакета.
    -- Тела пакет не имеет, только спецификацию.

    exc_wrong_name      exception;
    exc_too_low_salary   exception;
    exc_incorrect_pass   exception;

end;
```

Далее работать с этими исключениями можно подобным образом:


```

begin
    -- Какой-то код
    ...
exception
    when pck_hr_errors.exc_too_low_salary then
        -- Обработка исключения
        ...
end;

```

Обработка непредопределённых ошибок

Не все ошибки в Oracle являются предопределёнными. Когда возникает необходимость их обрабатывать, нужно связать переменную типа `exception` с кодом ошибки, которую нужно обработать:

```

declare
    -- объявляем ошибку
    e_incorrect_date exception;

    -- связываем ошибку с кодом
    pragma exception_init(e_incorrect_date, -1830);
begin
    dbms_output.put_line(to_date('2022-02-01', 'dd.mm.yyyy'));
exception
    when e_incorrect_date then
        dbms_output.put_line ('Неправильный формат даты');
end;

```

Следует помнить, что коды ошибок являются *отрицательными* числами.

Ошибки и вложенные блоки

Если ошибка не обрабатывается в пределах блока `begin ..end`, она выбрасывается за его пределы. Далее эта ошибка может быть обработана блоком `exception` внешнего блока. Если и там ошибка не обрабатывается, она выбрасывается и за его пределы, и так далее.

```

declare
    a number;

-- Внешний блок
begin
    -- Вложенный блок
    begin
        a := 1 / 0;
        -- Важно помнить, что после возникновения ошибки
        -- выполнение кода в пределах блока прекращается.
        -- Следующий код не будет выполнен
        dbms_output.put_line('Этот код не будет выполнен');
    end;
exception
    when zero_divide:
        dbms_otuput.put_line('Ошибка обработана внешним блоком');
end;

```

raise_application_error

Если ошибка, брошенная Oracle, достигает клиентского приложения, то она имеет примерно такой текст: `ORA-01722 invalid number`.

Процедура `raise_application_error` позволяет вызвать исключение с заданным номером, связать его с сообщением и отправить его вызывающему приложению.

```

begin
    raise_application_error(-20134, 'Неправильный номер паспорта');
end;

```

Диапазон возможных кодов ошибок [-20999, 20000]. Сообщение должно помещаться в тип `varchar2(2000)`.

Можно указать третий boolean параметр, который в случае значения `true` добавит текущую ошибку в список предыдущих ошибок, возникших в приложении. По умолчанию значение равно `false`, что значит, про сообщение об ошибке заменяет все предыдущие ошибки собой.

Мы можем объявить собственное исключение, связать его с номером в диапазоне [-20999, 20000] и использовать для обработки исключений, брошенных с помощью

```
raise_application_error :
```

```
declare
    e_wrong_passport exception;

    -- связываем ошибку с кодом
    pragma exception_init(e_wrong_passport, -20999);
begin
    raise_application_error(-20999, 'Неправильный номер паспорта');
exception
    when e_wrong_password then
        dbms_output.put_line ('Неправильный номер паспорта');
end;
```

Взаимодействие с данными

Сам по себе PL/SQL бесполезен, если мы не сможем взаимодействовать с данными, которые хранятся в БД. В этой части мы рассмотрим, как сохранять результаты запросов в переменных, а также как легко можно использовать переменные PL/SQL в SQL-запросах.

Подготовка данных

```
create table users(  
    id number primary key,  
    username varchar2(50) not null,  
    create_date date default sysdate not null,  
    is_admin number default 0  
);  
  
insert into users(id, username, create_date)  
values(1, 'nagibator', to_date('2022-01-01', 'yyyy-mm-dd'));  
  
insert into users(id, username, create_date)  
values(2, 'ordinaryuser', to_date('2022-03-15', 'yyyy-mm-dd'));  
  
insert into users(id, username, create_date)  
values(3, 'reporter', to_date('2022-03-08', 'yyyy-mm-dd'));  
  
insert into users(id, username, create_date, is_admin)  
values(4, 'admin', to_date('2021-01-01', 'yyyy-mm-dd'), 1);
```

Сохранение единичных результатов в переменные

Сохранение результатов выборки в переменные PL/SQL осуществляется через конструкцию

```
select ... into .
```

Рассмотрим самый простой вариант - сохранить результат выполнения одного `select` запроса в переменную:

```

declare
    -- Объявление переменной
    l_username varchar2(50);
begin
    select username into l_username
    from users
    where id = 1;

    dbms_output.put_line(l_username);

end;

```

Результат:

```
nagibator
```

Выборка нескольких значений

Мы можем получать из одного SQL запроса несколько колонок и сразу сохранять их в переменных:

```

declare
    l_user_id number;
    l_username varchar2(50);
begin
    select id, username into l_user_id, l_username
    from users
    where id = 1;

    dbms_output.put_line(l_user_id);
    dbms_output.put_line(l_username);

end;

```

Результат:

```

1
nagibator

```

Количество колонок из запроса и количество переменных, в которые они сохраняются, должно быть одинаково. Следующий код не будет выполнен и выбросит ошибку `ORA-00947: not enough values :`

```
declare
    l_user_id number;
begin
    -- Будет выброшена ошибка, т.к. запрос возвращает
    -- две колонки, а принимающая переменная только одна
    select id, username into l_user_id
    from users
    where id = 1;
end;
```

Обработка ошибок

Все предыдущие варианты работают в тех случаях, когда запрос возвращает ровно одну строку. Сохранение нескольких строк в переменные будет рассмотрено в части про коллекции в PL/SQL. Тем не менее, нужно учитывать, что не всегда мы можем быть уверены в том, что запрос будет возвращать ровно одну строку, и будет ли вообще. Для того, чтобы программа не посыпалась, нужно обработать возможные ошибки; мы рассмотрим два варианта - когда строк больше одной и когда их нет вообще. Это одни из самых первых ошибок, которые следует отлавливать при сохранении данных в переменные PL/SQL.

Пример, когда запрос возвращает больше одной строки:

```
declare
    l_user_id number;
begin
    select id into l_user_id
    from users;
exception
    when too_many_rows then
        l_user_id := null;
end;
```

Пример, когда запрос не возвращает ни одной строки:

```

declare
    l_user_id number;
begin
    select id into l_user_id
    from users
    where id = -1;
exception
    when no_data_found then
        l_user_id := null;
end;

```

Обработаем обе возможные ошибки сразу:

```

declare
    l_user_id number;
begin
    select id into l_user_id
    from users
    where id = -1;
exception
    when no_data_found or too_many_rows then
        l_user_id := null;
end;

```

Использование переменных в SQL запросах

Переменные SQL использовать *очень просто*:

```

declare
    l_user_id number := 1;
    l_username varchar2(50);
begin
    select username into l_username
    from users
    where id = l_user_id;

    dbms_output.put_line(l_username);

end;

```


В примере выше мы сохранили в значение id нужной нам записи в переменной `l_user_id`, после чего использовали её в запросе. Как видно, нам не нужно делать абсолютно ничего, мы просто подставляем переменную PL/SQL в SQL запрос, и всё работает отлично, благодаря **тесной интеграции SQL и PL/SQL в Oracle**.

Ещё пример: получим количество записей в таблице и выведем его:

```
declare
    l_cnt number;
begin
    select count(*) into l_cnt
    from users;

    dbms_output.put_line(l_cnt);
end;
```

Результат:

4

Сохранять результаты запроса в константы нельзя:

```
declare
    l_cnt constant number := 20;
begin
    -- Возникнет исключение
    select count(*) into l_cnt
    from users;

    dbms_output.put_line(l_cnt);
end;
```

Код выше вызовет ошибку `PLS-00403: expression 'L_CNT' cannot be used as an INTO-target of a SELECT/FETCH statement`.

А вот использовать константы как параметры SQL запроса можно:

```
declare
    l_user_id constant number := 1;
    l_username varchar2(50);
begin
    select username into l_username
    from users
    where id = l_user_id;

    dbms_output.put_line(l_username);

end;
```