Введение в SQL

Данный конспект создан по данным взятым из разных открытых источников, а также включает мои собственные наработки по этой теме.

SQL (Structured Query Language - «язык структурированных запросов») - универсальный язык, применяемый для создания, модификации и управления данными в реляционных базах данных.

Цель данного курса: обзор основных конструкций оператора **SELECT**, используемого для выборки данных из реляционных СУБД.

Описание демонстрационной БД

В примерах будут использованы таблицы из схемы HR:

- COUNTRIES – справочник стран
- DEPARTMENTS – департаменты
- EMPLOYEES – сотрудники
- JOBS – должности
- JOB_HISTORY – история по должностям сотрудников
- LOCATIONS – местоположения департаментов
- REGIONS – регионы

Описание полей/столбцов таблиц

COUNTRIES

Столбец	Тип	Допустимо NULL	Описание
COUNTRY_ID	CHAR(2)	No	Primary key of countries table.
COUNTRY_NAME	VARCHAR2(40)	Yes	Country name
REGION_ID	NUMBER	Yes	Region ID for the country. Foreign key to region_id column in the departments table.

DEPARTMENTS

Столбец	Тип	Допустимо NULL	Описание
DEPARTMENT_ID	NUMBER(4,0)	No	Primary key column of departments table.
DEPARTMENT_NAME	VARCHAR2(30)	No	A not null column that shows name of a department. Administration, Marketing, Purchasing, Human Resources, Shipping, IT, Executive, Public Relations, Sales, Finance, and Accounting.
MANAGER_ID	NUMBER(6,0)	Yes	Manager_id of a department. Foreign key to employee_id column of employees table. The manager_id column of the employee table references this column.
LOCATION_ID	NUMBER(4,0)	Yes	Location id where a department is located. Foreign key to location id column of locations table.

EMPLOYEES

Столбец	Тип	Допустимо NULL	Описание
EMPLOYEE_ID	NUMBER(6,0)	No	Primary key of employees table.
FIRST_NAME	VARCHAR2(20)	Yes	First name of the employee. A not null column.
LAST_NAME	VARCHAR2(25)	No	Last name of the employee. A not null column.
EMAIL	VARCHAR2(25)	No	Email id of the employee
PHONE_NUMBER	VARCHAR2(20)	Yes	Phone number of the employee; includes country code and area code
HIRE_DATE	DATE	No	Date when the employee started on this job. A not null column.
JOB_ID	VARCHAR2(10)	No	Current job of the employee; foreign key to job_id column of the jobs table. A not null column."
SALARY	NUMBER(8,2)	Yes	Monthly salary of the employee. Must be greater than zero (enforced by constraint emp_salary_min)
COMMISSION_PCT	NUMBER(2,2)	Yes	Commission percentage of the employee; Only employees in sales department elgible for commission percentage
MANAGER_ID	NUMBER(6,0)	Yes	Manager id of the employee; has same domain as manager_id in departments table. Foreign key to employee_id column of employees table. (useful for reflexive joins and CONNECT BY query)
DEPARTMENT_ID	NUMBER(4,0)	Yes	Department id where employee works; foreign key to department_id column of the departments table

JOBS

Столбец	Тип	Допустимо NULL	Описание
JOB_ID	VARCHAR2(10)	No	Primary key of jobs table.
JOB_TITLE	VARCHAR2(35)	No	A not null column that shows job title, e.g. AD VP, FI ACCOUNTANT
MIN_SALARY	NUMBER(6,0)	Yes	Minimum salary for a job title.
MAX_SALARY	NUMBER(6,0)	Yes	Maximum salary for a job title

JOB_HISTORY

Столбец	Тип	Допустимо NULL	Описание
EMPLOYEE_ID	NUMBER(6,0)	No	A not null column in the complex primary key employee id+start date. Foreign key to employee id column of the employee table
START_DATE	DATE	No	A not null column in the complex primary key employee id+start date. Must be less than the end date of the job history table. (enforced by constraint jhist date interval)
END_DATE	DATE	No	Last day of the employee in this job role. A not null column. Must be greater than the start_date of the job_history table. (enforced by constraint jhist date interval)
JOB_ID	VARCHAR2(10)	No	Job role in which the employee worked in the past; foreign key to job_id column in the jobs table. A not null column.
DEPARTMENT_ID	NUMBER(4,0)	Yes	Department id in which the employee worked in the past; foreign key to deparment_id column in the departments table

LOCATIONS

Столбец	Тип	Допустимо NULL	Описание
LOCATION_ID	NUMBER(4,0)	No	Primary key of locations table
STREET_ADDRESS	VARCHAR2(40)	Yes	Street address of an office, warehouse, or production site of a company. Contains building number and street name
POSTAL_CODE	VARCHAR2(12)	Yes	Postal code of the location of an office, warehouse, or production site of a company.
CITY	VARCHAR2(30)	No	A not null column that shows city where an office, warehouse, or production site of a company is located.
STATE_PROVINCE	VARCHAR2(25)	Yes	State or Province where an office, warehouse, or production site of a company is located.
COUNTRY_ID	CHAR(2)	Yes	Country where an office, warehouse, or production site of a company is located. Foreign key to country_id column of the countries table.

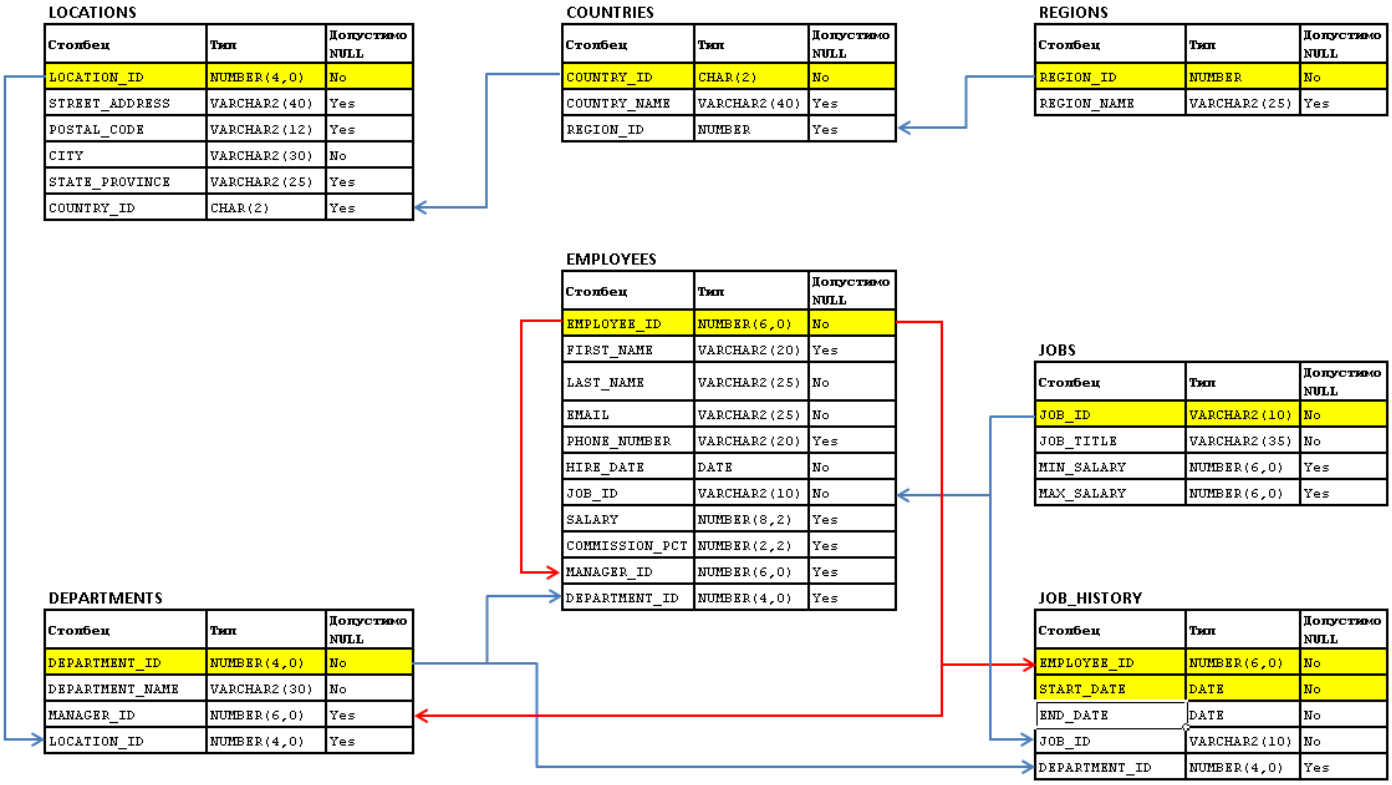
REGIONS

Столбец	Тип	Допустимо NULL	Описание
REGION_ID	NUMBER	No	
REGION_NAME	VARCHAR2(25)	Yes	

Желтым цветом выделены поля входящие в первичный ключ таблицы.

Описания полей можно так же посмотреть в комментариях базы данных.

Схема базы данных HR



LOCATIONS

Столбец	Тип	Допустимо NULL
LOCATION_ID	NUMBER(4,0)	No
STREET_ADDRESS	VARCHAR2(40)	Yes
POSTAL_CODE	VARCHAR2(12)	Yes
CITY	VARCHAR2(30)	No
STATE_PROVINCE	VARCHAR2(25)	Yes
COUNTRY_ID	CHAR(2)	Yes

COUNTRIES

Столбец	Тип	Допустимо NULL
COUNTRY_ID	CHAR(2)	No
COUNTRY_NAME	VARCHAR2(40)	Yes
REGION_ID	NUMBER	Yes

REGIONS

Столбец	Тип	Допустимо NULL
REGION_ID	NUMBER	No
REGION_NAME	VARCHAR2(25)	Yes

EMPLOYEES

Столбец	Тип	Допустимо NULL
EMPLOYEE_ID	NUMBER(6,0)	No
FIRST_NAME	VARCHAR2(20)	Yes
LAST_NAME	VARCHAR2(25)	No
EMAIL	VARCHAR2(25)	No
PHONE_NUMBER	VARCHAR2(20)	Yes
HIRE_DATE	DATE	No
JOB_ID	VARCHAR2(10)	No
SALARY	NUMBER(8,2)	Yes
COMMISSION_PCT	NUMBER(2,2)	Yes
MANAGER_ID	NUMBER(6,0)	Yes
DEPARTMENT_ID	NUMBER(4,0)	Yes

JOBS

Столбец	Тип	Допустимо NULL
JOB_ID	VARCHAR2(10)	No
JOB_TITLE	VARCHAR2(35)	No
MIN_SALARY	NUMBER(6,0)	Yes
MAX_SALARY	NUMBER(6,0)	Yes

DEPARTMENTS

Столбец	Тип	Допустимо NULL
DEPARTMENT_ID	NUMBER(4,0)	No
DEPARTMENT_NAME	VARCHAR2(30)	No
MANAGER_ID	NUMBER(6,0)	Yes
LOCATION_ID	NUMBER(4,0)	Yes

JOB_HISTORY

Столбец	Тип	Допустимо NULL
EMPLOYEE_ID	NUMBER(6,0)	No
START_DATE	DATE	No
END_DATE	DATE	No
JOB_ID	VARCHAR2(10)	No
DEPARTMENT_ID	NUMBER(4,0)	Yes

Простой оператор SELECT

Базовый синтаксис команды SELECT выглядит следующим образом:

```
SELECT [DISTINCT] список_столбцов FROM источник WHERE фильтр ORDER BY выражение_сортировки
```

SELECT *

В самом простом виде оператор SELECT выглядит следующим образом:

```
SELECT * FROM employees
```

«*» означает, что в выборку необходимо включить все столбцы таблицы EMPLOYEES. При этом строки результирующего набора будут не упорядочены.

Если вы работаете не под пользователем HR, т.е. делаете выборку из другой схемы, то имя таблицы необходимо предварить именем схемы:

```
SELECT * FROM hr.employees
```

Комментарии

В тексте запроса можно использовать комментарии 2-х видов – однострочные и многострочные:

```
-- текст комментария
/* текст комментария */
```

SELECT с перечислением полей

Если нам нужно получить из таблицы только определенные поля, мы их можем перечислить через запятую:

```
SELECT employee_id, first_name, last_name, hire_date, salary FROM employees
```

Имя поля можно предварять именем таблицы:

```
SELECT employees.employee_id, employees.first_name FROM employees
```

Для таблицы можно задать более удобный псевдоним:

```
SELECT emp.employee_id, emp.first_name FROM employees emp
```

В каких ситуациях нам могут пригодиться псевдонимы мы рассмотрим далее.

Арифметические операторы Oracle. Специальная таблица DUAL

В Oracle SQL предусмотрен стандартный набор арифметических операторов:

Оператор	Действие
+	сложение
-	вычитание
*	умножение
/	деление

К арифметическим также относятся унарные операторы (+) и (-), которые определяют знак числа. Приоритет арифметических операторов также привычен: наивысший приоритет у унарных операторов, затем выполняется умножение и деление, а после этого — сложение и вычитание. Порядок применения операторов регулируется круглыми скобками.

Применение унарного оператора к значению типа NULL даст в результате NULL.

Конечно, нельзя использовать два оператора - последовательно (как бинарный, а затем унарный оператор) — такая последовательность символов зарезервирована за комментариями SQL.

Арифметические операторы можно использовать даже в таких запросах, в которых вообще не используются столбцы:

```
SELECT 10*15 FROM hr.employees
```

Однако в этом случае результат будет повторен 107 раз, что, скорее всего, нам не нужно. Для выполнения запросов, в которых вообще не используются столбцы из базы данных, в Oracle зарезервирована специальная таблица DUAL (в SQL Server она не предусмотрена). В этой таблице — всего один столбец DUMMY с единственным значением x. Наш запрос с использованием этой служебной таблицы может выглядеть так:

```
SELECT 10*15 FROM DUAL
```

Оператор конкатенации

В реальной работе очень часто приходится производить слияние строковых значений. Такая операция называется конкатенацией.

В Oracle SQL предусмотрен несколько необычный (по крайней мере, с точки зрения специалистов, работавших с SQL Server) оператор конкатенации — две вертикальные черты (||). Отметим, что в некоторых версиях сервера Oracle, например, предназначенных для работы на мэйнфреймах IBM, используется другой оператор конкатенации. Поэтому на всякий случай в Oracle SQL предусмотрена еще и встроенная функция CONCAT, которая также производит конкатенацию и работает на любых платформах.

Пример применения оператора конкатенации || может выглядеть так:

```
SELECT first_name || ' ' || last_name FROM hr.employees
```

То же самое с использованием функции CONCAT:

```
SELECT CONCAT(CONCAT(first_name,' '),last_name) FROM hr.employees
```

В этом примере у нас сливается три значения: имя, пробел (в виде литерала) и фамилия.

Оператор конкатенации можно применять как для строковых значений и значений CLOB (больших строковых значений), так и для чисел и дат (эти значения будут автоматически конвертированы в строковые). При слиянии строкового значения со значением типа NULL Oracle вернет строковое значение (в отличие от поведения по умолчанию SQL Server).

SELECT DISTINCT – выборка только уникальных строк

Следует отметить, что вертикальная выборка может содержать дубликаты строк в том случае, если она не содержит потенциального ключа, однозначно определяющего запись. Например, следующий запрос вернет нам номер департамента и оклад по каждому сотруднику:

```
SELECT department_id, salary FROM employees
```

Понятно, что у некоторых сотрудников могут быть одинаковые оклады, и чтобы исключить повторения, т.е. если нам нужно получить все уникальные оклады в разрезе департаментов, то мы можем использовать ключевое слово **DISTINCT**:

```
SELECT DISTINCT department_id, salary FROM employees
```

Помимо **DISTINCT** может применяться также ключевое слово **ALL** (все строки), которое принимается по умолчанию.

Переименование столбцов в результирующем наборе

Имена столбцов, указанные в предложении SELECT, можно переименовать. Это делает результаты более читабельными, поскольку имена полей в таблицах часто сокращают с целью упрощения набора. Ключевое слово AS, используемое для переименования, согласно стандарту можно и опустить, так как оно неявно подразумевается. Особенно удобно использовать псевдонимы для вычисляемых столбцов, для которых изначально имя не предусмотрено. Согласно стандарту могут использоваться имена с ограничителями (delimited identifier), при этом в качестве ограничителя применяется символ двойной кавычки ("). Такой прием допускает присутствие в именах специальных символов и зарезервированных слов.

```
SELECT
  first_name || ' ' || last_name as ФИО, -- даем имя вычисляемому столбцу
  hire_date as "Дата приема", -- использование двойных кавычек
  salary zp -- ключевое слово AS является не обязательным
FROM employees
```

ORDER BY – сортировка результата запроса

Чтобы упорядочить строки результирующего набора, можно выполнить сортировку по любому количеству полей. Для этого используется предложение ORDER BY с перечнем полей, являющееся всегда последним предложением в операторе SELECT. При этом в списке полей могут указываться как имена полей, так и их порядковые позиции в списке предложения SELECT.

```
SELECT first_name, last_name, hire_date, salary FROM employees
ORDER BY first_name, last_name -- упорядочить список по Имени и Фамилии
```

Если требуется отсортировать по какому либо из полей в порядке убывания, то после поля необходимо добавить ключевое слово DESC:

```
SELECT first_name, last_name, hire_date, salary FROM employees
ORDER BY salary DESC, first_name, last_name /* упорядочить список в порядке убывания
размера Заработной Платы и после по Имени и Фамилии */
```

В **ORDER BY** можно так же указывать номера столбцов. Следующий пример выдаст аналогичный результат:

```
SELECT first_name, last_name, hire_date, salary FROM employees
ORDER BY 4 DESC, 1, 2
```

Сортировку можно проводить по возрастанию (параметр ASC принимается по умолчанию) или по убыванию (параметр DESC).

Не рекомендуется в приложениях использовать запросы с сортировкой по номерам столбцов. Это связано с тем, что со временем структура таблицы может измениться, например, в результате добавления/удаления столбцов. Как следствие, запрос типа:

```
SELECT * FROM employees
ORDER BY 2, 3
```

может давать совсем другую последовательность или вообще вызывать ошибку, ссылаясь на отсутствующий столбец.

Применим сортировку для запроса, который возвращал уникальные оклады в разрезе департаментов:

```
SELECT DISTINCT department_id, salary FROM employees
ORDER BY department_id, salary
```

Стоит отметить, что сортировка может производиться и по полям не входящих в выборку, а также по вычисляемым значениям:

```
SELECT first_name, last_name FROM employees
ORDER BY salary DESC, first_name || last_name


SELECT first_name || ' ' || last_name as fio FROM employees
ORDER BY salary DESC, fio -- используем для сортировки присвоенное имя
```

WHERE – условие для выборки строк

Горизонтальную выборку реализует предложение **WHERE**, которое записывается после предложения **FROM**. При этом в результирующий набор попадут только те строки из источника записей, для каждой из которых условие, указанное в предложении **WHERE** будет равно TRUE.

```
SELECT first_name, last_name, salary FROM employees
WHERE department_id = 50 -- выбрать записи по департаменту 50
ORDER BY salary DESC, first_name || last_name
```

Псевдостолбец ROWNUM

Oracle умеет нумеровать строки в возвращаемых результатах. Для этого используется специальный псевдостолбец ROWNUM.

```
SELECT ROWNUM, e.* FROM  hr.employees e
```

ROWNUM можно использовать для ограничения количества выводимых записей, например:

```
SELECT * FROM employees WHERE ROWNUM <= 10
```

В этом случае будет выведено всего 10 записей.

Такую возможность очень удобно использовать, когда запрос потенциально может возвращать очень большое количество значений и сильно загрузить сервер, а вам нужно просто убедиться в том, что он выполняется нормально.

Предикаты

Предикаты представляют собой выражения, принимающие истинностное значение. Они могут представлять собой как одно выражение, так и любую комбинацию из неограниченного количества выражений, построенную с помощью булевых операторов **AND**, **OR** или **NOT**. Кроме того, в этих комбинациях может использоваться SQL-оператор **IS**, а также круглые скобки для конкретизации порядка выполнения операций.

Предикат в языке SQL может принимать одно из трех значений **TRUE** (истина), **FALSE** (ложь) или **UNKNOWN** (неизвестно). Исключение составляют следующие предикаты: **IS NULL** (отсутствие значения), **EXISTS** (существование), которые не могут принимать значение **UNKNOWN**.

Простые условия (операторы сравнения) в запросах Oracle

Простые условия сравнения в Oracle:

Условие	Значение
=	Равно
<	Меньше
>	Больше
<=	Меньше или равно
>=	Больше или равно
<> != ^=	Не равно

«не равно» в Oracle может обозначаться разными способами. Однако рекомендуется по возможности использовать стандартное обозначение вида (<>).

Выражения в предикатах сравнения могут содержать константы и любые поля из таблиц, указанных в предложении FROM. Символьные строки и константы типа дата/время записываются в апострофах.

Логические условия в фильтрах WHERE в запросах Oracle SQL, ключевые слова AND, OR, NOT

AND — логическое И. Чтобы выражение вернуло True, нужно, чтобы истинными были оба условия:

```
SELECT * FROM hr.employees
WHERE department_id = 50 AND JOB_ID = 'SH_CLERK'
```

OR — логическое ИЛИ. Чтобы выражение вернуло True, достаточно, чтобы истинным было только одно условие:

```
SELECT * FROM hr.employees
WHERE department_id = 50 OR JOB_ID = 'SH_CLERK'
```

NOT — "обращает" условие. При добавлении NOT выражение будет возвращать True, когда без него оно возвращало бы False, и наоборот:

```
SELECT * FROM hr.employees
WHERE NOT department_id = 50
```

Таблица истинности для оператора AND:

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

Таблица истинности для оператора OR:

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

Таблица истинности для оператора NOT:

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

Приоритет: 1) Все операторы сравнения; 2) NOT; 3) AND; 4) OR.

Изменить стандартную последовательность вычислений можно при помощи круглых скобок.

Предикат BETWEEN

Синтаксис:

```
<Проверяемое выражение> [NOT] BETWEEN <Начальное выражение> AND <Конечное выражение>
```

Предикат BETWEEN проверяет, попадают ли значения проверяемого выражения в диапазон, задаваемый пограничными выражениями, соединяемыми служебным словом AND. Естественно, как и для предиката сравнения, выражения в предикате BETWEEN должны быть совместимы по типам.

Предикат `exp1 BETWEEN exp2 AND exp3` равносилен предикату `exp1 >= exp2 AND exp1 <= exp3`
А предикат `exp1 NOT BETWEEN exp2 AND exp3` равносилен предикату `NOT (exp1 BETWEEN exp2 AND exp3)`

```
SELECT * FROM employees
WHERE salary BETWEEN 10000 AND 15000 -- оклад в диапазоне
```

Предикат IN

Синтаксис:

```
<Проверяемое выражение> [NOT] IN (<подзапрос>) | (<выражение для вычисления значения>,...)
```

Предикат IN определяет, будет ли значение проверяемого выражения обнаружено в наборе значений, который либо явно определен, либо получен с помощью табличного подзапроса. Здесь табличный подзапрос это обычный оператор SELECT, который создает одну или несколько строк для одного столбца, совместимого по типу данных со значением проверяемого выражения. Если целевой объект эквивалентен хотя бы одному из указанных в предложении IN значений, истинностное значение предиката IN будет равно TRUE. Если для каждого значения X в предложении IN целевой объект <> X, истинностное значение будет равно FALSE. Если подзапрос выполняется, и результат не содержит ни одной строки (пустая таблица), предикат принимает значение FALSE. Когда не соблюдается ни одно из упомянутых выше условий, значение предиката равно UNKNOWN.

Пример с явным перечислением значений:

```
SELECT * FROM employees
WHERE department_id IN(10, 20) -- выбрать сотрудников только указанных департаментов
```

Синтаксические конструкции с IN можно заменить на обычные операторы сравнения с OR. Однако при сравнении с большим списком значений синтаксис с IN явно удобнее.

```
SELECT * FROM employees
WHERE department_id = 10 OR department_id = 20
```

Пример с использованием подзапроса возвращающим значения:

```
SELECT * FROM employees
WHERE department_id IN (SELECT department_id FROM departments WHERE location_id = 1700)
/* выбрать сотрудников департаментов расположенных в регионе 1700 */
```

Предикат LIKE

Синтаксис:

```
<Выражение для вычисления значения строки>
 [NOT] LIKE <Выражение для вычисления значения строки>
 [ESCAPE <символ>]
```

Предикат LIKE сравнивает строку, указанную в первом выражении, для вычисления значения строки, называемого проверяемым значением, с образцом, который определен во втором выражении для вычисления значения строки. В образце разрешается использовать два трафаретных символа:

- символ подчеркивания (_), который можно применять вместо любого единичного символа в проверяемом значении;
- символ процента (%) заменяет последовательность любых символов (число символов в последовательности может быть от 0 и более) в проверяемом значении.

Если проверяемое значение соответствует образцу с учетом трафаретных символов, то значение предиката равно TRUE.

```
SELECT * FROM employees
WHERE LOWER(first_name) LIKE 'alex%' -- все сотрудники чье имя начинается с alex
```

Если искомая строка содержит трафаретный символ, то следует задать управляющий символ в предложении **ESCAPE**. Этот управляющий символ должен использоваться в образце перед трафаретным символом, сообщая о том, что последний следует трактовать как обычный символ. Например, если в некотором поле следует отыскать все значения, содержащие символ «_», то шаблон '%_%' приведет к тому, что будут возвращены все записи из таблицы. В данном случае шаблон следует записать следующим образом:

```
'%#_%' ESCAPE '#'
```

Для проверки значения на соответствие строке «25%» можно воспользоваться таким предикатом:

```
LIKE '25|%' ESCAPE '|'
```

Истинностное значение предиката LIKE присваивается в соответствии со следующими правилами:

- если либо проверяемое значение, либо образец, либо управляющий символ есть NULL, истинностное значение равно UNKNOWN;
- в противном случае, если проверяемое значение и образец имеют нулевую длину, истинностное значение равно TRUE;
- в противном случае, если проверяемое значение соответствует шаблону, то предикат LIKE равен TRUE;
- если не соблюдается ни одно из перечисленных выше условий, предикат LIKE равен FALSE.

REGEXP_LIKE

Условие LIKE использовать совсем несложно, но и набор возможностей у него очень ограничен. Намного большие функциональные возможности предоставляет условие REGEXP_LIKE, которое позволяет определять условие поиска при помощи стандартных POSIX-совместимых регулярных выражений. Регулярные выражения — очень большая тема. Кратко о регулярных выражениях рассказано в главе «**Основы регулярных выражений**». Здесь приведем только простой пример:

```
SELECT first_name FROM employees
WHERE REGEXP_LIKE (first_name, '^Ste(v|ph)en$') -- Steven, Stephen
```

В этом примере мы возвращаем информацию о всех сотрудниках, у которых имя начинается на Ste, заканчивается на en, а между ними находится либо v, либо ph.

Использование значения NULL в условиях поиска

Предикат:

```
IS [NOT] NULL
```

позволяет проверить отсутствие (наличие) значения в полях таблицы. Использование в этих случаях обычных предикатов сравнения может привести к неверным результатам, так как сравнение со значением NULL дает результат UNKNOWN (неизвестно).

Пример использования предиката:
```
SELECT * FROM employees
WHERE commission_pct IS NOT NULL
```

Характерной ошибкой является написание предиката в виде:
```
SELECT * FROM employees
WHERE commission_pct <> NULL
```
Этому предикату не соответствует ни одной строки, поэтому результирующий набор записей будет пуст. Это происходит потому, что сравнение с NULL-значением согласно предикату сравнения оценивается как UNKNOWN. А строка попадает в результирующий набор только в том случае, если предикат в предложении WHERE есть TRUE. Это же справедливо и для предиката в предложении HAVING.

Групповые условия (операторы сравнения) - ключевые слова ALL, ANY, SOME

Для групповых условий сравнения в Oracle SQL используются три ключевых слова: ALL, SOME и ANY. Все эти ключевые слова применяются для сравнения указанного вами значения с набором значений, который возвращает подзапрос (или явно указанный набор значений):

ALL — сравнение будет производиться со всеми записями, которые возвращает подзапрос (или просто со всеми значениями в набор). True вернется только в том случае, если все записи, которые возвращает подзапрос, будут удовлетворять указанному вами условию. Кроме того, в Oracle значение True вернется в ситуации, когда подзапрос не вернет ни одной записи.

В качестве примера приведем такой запрос:
```
SELECT * FROM hr.employees
WHERE salary <= ALL(SELECT salary FROM hr.employees WHERE job_id = 'SH_CLERK')
```
Он вернет записи для всех сотрудников, для которых зарплата меньше или равна самой маленькой зарплате у сотрудников с должностью SH_CLERK.

ANY — сравнение вернет True, если условию будет удовлетворять любая запись из набора (или подзапроса). Например, такой запрос вернет всех пользователей, зарплата которых совпадает с зарплатой любого из клерков:
```
SELECT * FROM hr.employees
WHERE salary = ANY(SELECT salary FROM hr.employees WHERE job_id = 'SH_CLERK')
```

Пример сравнения со списком значений:
```
SELECT * FROM employees
WHERE department_id=ANY(10,20) /* обычно в таком виде не используется, т.к. для этой цели можно
воспользоваться оператором IN - department_id IN(10,20) */
```

SOME и **ANY** являются синонимами, то есть может использоваться любое из них.

Условие EXISTS и проверка существования набора значений

Условие EXISTS используется только в одной ситуации — когда вы используете в запросе и подзапрос и хотите проверить, возвращает ли подзапрос записи. Если подзапрос возвращает хотя бы одну запись, то условие EXISTS вернет True, если нет — то false. Пример применения этого условия может выглядеть так:
```
SELECT * FROM departments d -- задаем псевдоним "d" для таблицы departments
WHERE EXISTS(SELECT * FROM employees e WHERE d.department_id = e.department_id)
```

В этом примере мы возвращаем информацию о всех департаментах, для которых в таблице employees есть хотя бы один сотрудник.

Как обычно, это условие можно обращать при помощи NOT:
```
SELECT * FROM departments d
WHERE NOT EXISTS(SELECT * FROM employees e WHERE d.department_id = e.department_id)
```

Получение итоговых значений – Агрегатные функции

Стандартом предусмотрены следующие агрегатные функции:

название	**описание**
COUNT(*)	Возвращает количество строк источника записей
COUNT(имя столбца | выражение)	Возвращает количество значений в указанном столбце/выражении
COUNT(DISTINCT имя столбца | выражение)	Возвращает количество уникальных значений в указанном столбце/выражении
SUM	Возвращает сумму значений в указанном столбце
AVG	Возвращает среднее значение в указанном столбце
MIN	Возвращает минимальное значение в указанном столбце
MAX	Возвращает максимальное значение в указанном столбце

Все эти функции возвращают единственное значение. При этом функции COUNT, MIN и MAX применимы к данным любого типа, в то время как SUM и AVG используются только для данных числового типа. Разница между функцией COUNT(*) и COUNT(имя столбца | выражение) состоит в том, что вторая (как и остальные агрегатные функции) при подсчете не учитывает NULL-значения.

Для примера, получим некоторые итоговые значения по таблице employees:

```
SELECT
  COUNT(*), -- общее кол-во сотрудников
  COUNT(DISTINCT department_id), -- количество уникальных (задействованных) департаментов
  MIN(salary), -- минимальный оклад
  MAX(salary), -- максимальный оклад
  AVG(salary) -- средний оклад
FROM employees
```

Предложение GROUP BY – группировка данных

Предложение GROUP BY используется для определения групп выходных строк, к которым могут применяться агрегатные функции (COUNT, MIN, MAX, AVG и SUM). Если это предложение отсутствует, и используются агрегатные функции, то все столбцы с именами, упомянутыми в SELECT, должны быть включены в агрегатные функции, и эти функции будут применяться ко всему набору строк, которые удовлетворяют предикату запроса. В противном случае все столбцы списка SELECT, не вошедшие в агрегатные функции, должны быть указаны в предложении GROUP BY. В результате чего все выходные строки запроса разбиваются на группы, характеризуемые одинаковыми комбинациями значений в этих столбцах. После чего к каждой группе будут применены агрегатные функции. Следует иметь в виду, что для GROUP BY все значения NULL трактуются как равные, то есть при группировке по полю, содержащему NULL-значения, все такие строки попадут в одну группу.

Если при наличии предложения GROUP BY, в предложении SELECT отсутствуют агрегатные функции, то запрос просто вернет по одной строке из каждой группы. Эту возможность, наряду с ключевым словом DISTINCT, можно использовать для исключения дубликатов строк в результирующем наборе.

```
SELECT
  department_id,
  COUNT(*), -- кол-во сотрудников в департаменте
  MIN(salary), -- минимальный оклад по департаменту
  MAX(salary), -- максимальный оклад по департаменту
  AVG(salary) -- средний оклад по департаменту
FROM employees
GROUP BY department_id
```

Предложение HAVING

Если предложение WHERE определяет предикат для фильтрации строк, то предложение HAVING применяется после группировки для определения аналогичного предиката, фильтрующего группы по значениям агрегатных функций. Это предложение необходимо для проверки значений, которые получены с помощью агрегатной функции не из отдельных строк источника записей, определенного в предложении FROM, а из групп таких строк.

Например, выведем все департаменты, в которых минимальный оклад меньше 10000 и число сотрудников по департаменту больше 10:

```
SELECT
  department_id,
  COUNT(*), -- общее кол-во сотрудников
  MIN(salary) -- минимальный оклад
FROM employees
GROUP BY department_id
HAVING MIN(salary)<10000 AND COUNT(*)>10
ORDER BY department_id
```

Теперь мы можем сказать, что в общем виде оператор SELECT выглядит следующим образом:

```
SELECT [DISTINCT | ALL]{* | [<выражение для столбца> [[AS] <псевдоним>]] [,…]}
FROM <имя таблицы> [[AS] <псевдоним>] [,…]
[WHERE <предикат>]
[[GROUP BY <список столбцов>]
[HAVING <условие на агрегатные значения>]]
[ORDER BY <список столбцов>]
```

Следует отметить, что предложение HAVING может использоваться и без предложения GROUP BY. При отсутствии предложения GROUP BY агрегатные функции применяются ко всему выходному набору строк запроса, т.е. в результате мы получим всего одну строку, если выходной набор не пуст.

Таким образом, если условие на агрегатные значения в предложение HAVING будет истинным, то эта строка будет выводиться, в противном случае мы не получим ни одной строки.

Подзапросы

Как можно использовать подзапросы в блоке WHERE мы уже рассмотрели выше, когда рассматривали работу с групповыми условиями ALL и ANY.

Пример использования подзапроса в блоке SELECT:

```
SELECT
  department_name,
  (SELECT COUNT(*) FROM employees WHERE department_id=d.department_id) empl_count
FROM departments d
```

Здесь при помощи подзапроса, мы подсчитываем число сотрудников в каждом из департаментов.

Подзапросы могут также использоваться в блоке FROM:

```
SELECT d.department_name,e.empl_count
FROM
  departments d,
  (
    SELECT department_id,COUNT(*) empl_count FROM employees GROUP BY department_id
  ) e
WHERE d.department_id=e.department_id
```

Использование в запросе нескольких источников записей

Как видно из примера, приведенного в конце предыдущего раздела, в предложении FROM допускается указание нескольких таблиц. Простое перечисление таблиц через запятую практически не используется, поскольку оно соответствует реляционной операции, которая называется декартовым произведением. То есть в результирующем наборе каждая строка из одной таблицы будет сочетаться с каждой строкой из другой. Например, для таблиц:

A

A1	A2
1	2
2	1

B

B1	B2
2	4
3	3

Результат запроса:

```
SELECT * FROM A, B
```

будет выглядеть следующим образом:

A1	A2	B1	B2
1	2	2	4
1	2	3	3
2	1	2	4
2	1	3	3

Поэтому перечисление таблиц, как правило, используется совместно с условием соединения строк из разных таблиц, указываемым в предложении WHERE. Для приведенных выше таблиц таким условием может быть совпадение значений, скажем, в столбцах A1 и B1:

```
SELECT * FROM A, B WHERE A1=B1
```

Теперь результатом выполнения этого запроса будет следующая таблица:

A1	A2	B1	B2
2	1	2	4

то есть соединяются только те строки таблиц, у которых в указанных столбцах находятся равные значения (эквисоединение). Естественно, могут быть использованы любые условия, хотя эквисоединение используется чаще всего, поскольку эта операция воссоздает некую исходную сущность предметной области, декомпозированную на две других в результате процедуры нормализации в процессе построения логической модели.

Если разные таблицы имеют столбцы с одинаковыми именами, то для однозначности требуется использовать точечную нотацию, которая называется уточнением имени столбца:

```
<имя таблицы>.<имя столбца>
```

В тех случаях, когда это не вызывает неоднозначности, использование данной нотации не является обязательным.

Для примера получим для каждого сотрудника, название департамента и имя его менеджера (кому он подчиняется):

```
select
  e.first name || ' ' || e.last_name emp_full_name,
  d.department_name,
  (select first_name || ' ' || last_name from employees where employee_id=e.manager_id) manager_name
from employees e, departments d
where e.department_id = d.department_id
```

Операции соединения – JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN, CROSS JOIN

В предложении FROM может быть указана явная операция соединения двух и более таблиц. Среди ряда операций соединения, описанных в стандарте языка SQL, многими серверами баз данных поддерживается только операция соединения по предикату. Синтаксис соединения по предикату имеет вид:

```
FROM <таблица 1>
 [INNER]
 {{LEFT | RIGHT | FULL } [OUTER]} JOIN <таблица 2>
[ON <предикат>]
```

Соединение может быть либо внутренним (INNER), либо одним из внешних (OUTER). Служебные слова INNER и OUTER можно опускать, поскольку внешнее соединение однозначно определяется его типом — LEFT (левое), RIGHT (правое) или FULL (полное), а просто JOIN будет означать внутреннее соединение.

Предикат определяет условие соединения строк из разных таблиц. При этом INNER JOIN означает, что в результирующий набор попадут только те соединения строк двух таблиц, для которых значение предиката равно TRUE.

Перепишем пример, что был выше с использованием INNER JOIN (или просто JOIN):

```
SELECT
  e.first_name || ' ' || e.last_name emp_full_name,
  d.department_name
FROM employees e
JOIN departments d ON e.department_id = d.department_id
```

В результате, мы получили 106 записей, тогда как сотрудников в таблице employees 107.

Чтобы вывести всех сотрудников нам необходимо применить LEFT JOIN, что будет означать, что необходимо вывести все записи таблицы расположенной слева, т.е. employees:

```
SELECT
  e.first_name || ' ' || e.last_name emp_full_name,
  NVL(d.department_name,'не указан') dept_name
FROM employees e
LEFT JOIN departments d ON e.department_id = d.department_id
```

Также можем вывести имя менеджера, добавив еще одно соединение с таблицей employees:

```
SELECT
  e.first_name || ' ' || e.last_name emp_full_name,
  d.department_name,
  m.first_name || ' ' || m.last_name manager_name
FROM employees e
LEFT JOIN departments d ON d.department_id=e.department_id
LEFT JOIN employees m ON m.employee_id=e.manager_id
```

Здесь так же использовался LEFT JOIN, т.к. не у всех сотрудников может быть непосредственный руководитель.

Внешнее соединение LEFT JOIN означает, что помимо строк, для которых выполняется условие предиката, в результирующий набор попадут все остальные строки из первой таблицы (левой). При этом отсутствующие значения столбцов из правой таблицы будут заменены NULL-значениями.

К итоговой выборке мы можем применить дополнительное условие выборки, например:

```
SELECT
  e.first_name || ' ' || e.last_name emp_full_name,
  d.department_name,
  m.first_name || ' ' || m.last_name manager_name
FROM employees e
LEFT JOIN departments d ON d.department_id=e.department_id
LEFT JOIN employees m ON m.employee_id=e.manager_id
WHERE d.location_id=1700
```

Соединение RIGHT JOIN обратно соединению LEFT JOIN, то есть в результирующий набор попадут все строки из второй таблицы, которые будут соединяться только с теми строками из первой таблицы, для которых выполняется условие соединения.

Наконец, при полном соединении (FULL JOIN) в результирующую таблицу попадут не только те строки, которые имеют одинаковые значения в сопоставляемых столбцах, но и все остальные строки исходных таблиц, не имеющие соответствующих значений в другой таблице. В этих строках все столбцы той таблицы, в которой не было найдено соответствия, заполняются NULL-значениями. То есть полное соединение представляет собой комбинацию левого и правого внешних соединений. Так, запрос для таблиц A и B:

```
SELECT A.*, B.*
FROM A FULL JOIN B ON A1 = B1
```

даст следующий результат:

A1	A2	B1	B2
1	2	NULL	NULL
2	1	2	4
NULL	NULL	3	3

Заметим, что это соединение симметрично, то есть A FULL JOIN B эквивалентно B FULL JOIN A. Обратите также внимание на обозначение A.*, что означает вывести все столбцы таблицы A.

FULL JOIN можно съэмитировать при помощи 2-х запросов LEFT JOIN и RIGHT JOIN объединенных опирацией UNION (будет рассказано позже):

```
SELECT A.*, B.*
FROM A LEFT JOIN B ON A1 = B1

UNION

SELECT A.*, B.*
FROM A RIGHT JOIN B ON A1 = B1
```

Однако с точки зрения производительности второй запрос проигрывает первому вдвое по оценке стоимости плана. Это связано с тем, что операция UNION приводит к выполнению сортировки, которая отсутствует в плане первого запроса. Сортировка же необходима для процедуры исключения дубликатов, т.к. левое и правое соединения оба содержат строки, соответствующие внутреннему соединению.

В ORACLE можно использовать также старые конструкции (+), для реализации LEFT JOIN и RIGHT JOIN:

```
SELECT A.*, B.*
FROM A, B
WHERE A1 = B1(+) -- эквивалентно LEFT JOIN – дополнить таблицу B


SELECT A.*, B.*
FROM A, B
WHERE A1(+) = B1 -- эквивалентно RIGHT JOIN – дополнить таблицу A
```

По мне с использованием LEFT JOIN и RIGHT JOIN более наглядно, а операция (+) оправдана только в случае использования старых версий ORACLE, где еще не была реализована операции JOIN.

Ранее мы уже рассмотрели реализацию декартова произведения, состоящую в перечислении через запятую табличных выражений в предложении FROM (таблицы, представления, подзапросы) при отсутствии предложения WHERE, связывающего столбцы из перечисленных источников строк. Кроме того, можно использовать еще и явную операцию соединения – CROSS JOIN, например:

```
SELECT A.*, B.*
FROM A CROSS JOIN B
```

Напомним, что при декартовом произведении каждая строка из первой таблицы соединяется с каждой строкой второй таблицы. В результате количество строк результирующего набора равно произведению количества строк операндов декартова произведения. В чистом виде декартово произведение практически не используется, оно, как правило, является промежуточным результатом выполнения операции горизонтальной проекции (выборки) при наличии в операторе SELECT предложения WHERE.

Объединения – UNION [ALL], INTERSECT, MINUS

Операции объединения используются, в случае, когда необходимо объединить результаты работы нескольких запросов.

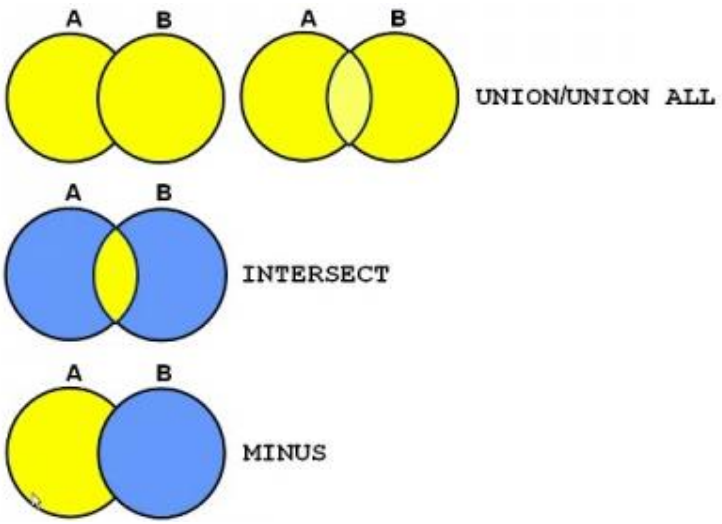
Для объединения запросов используется служебное слово UNION:

```
<запрос 1>
UNION [ALL]
<запрос 2>
```

Предложение UNION приводит к появлению в результирующем наборе всех строк каждого из запросов. При этом, если определен параметр ALL, то сохраняются все дубликаты выходных строк, в противном случае в результирующем наборе присутствуют только уникальные строки. Заметим, что можно связывать вместе любое число запросов. Кроме того, с помощью скобок можно задавать порядок объединения.

Операция объединения может быть выполнена только при выполнении следующих условий:

- количество выходных столбцов каждого из запросов должно быть одинаковым;
- выходные столбцы каждого из запросов должны быть совместимы между собой (в порядке их следования) по типам данных;
- в результирующем наборе используются имена столбцов, заданные в первом запросе;
- предложение ORDER BY применяется к результату соединения, поэтому оно может быть указано только в конце всего составного запроса.



Рассмотрим работу объединений на таблицах C и D.

C

C1	C2
1	Text 1
1	Text 1
2	Text 2
3	Text 3
4	Text 4
5	Text 5

D

D1	D2
2	Text 2
3	Text 3
6	Text 6

`SELECT c1 x, c2 y FROM c`
`UNION`
`SELECT d1, d2 FROM d`
`ORDER BY 1`
	X **Y**
	1 Text 1
	2 Text 2
	3 Text 3
	4 Text 4
	5 Text 5
	6 Text 6
		Объединение 2-х наборов с исключением дубликатов.

`SELECT c1 x, c2 y FROM c`
`UNION ALL`
`SELECT d1, d2 FROM d`
`ORDER BY 1`
	X **Y**
	1 Text 1
	1 Text 1
	2 Text 2
	2 Text 2
	3 Text 3
	3 Text 3
	4 Text 4
	5 Text 5
	6 Text 6
		Объединение 2-х наборов с включением всех строк.

`SELECT c1 x, c2 y FROM c`
`INTERSECT`
`SELECT d1, d2 FROM d`
`ORDER BY 1`
	X **Y**
	2 Text 2
	3 Text 3
		Пересечение 2-х множеств – строки, которые присутствуют в обоих множествах.

`SELECT c1 x, c2 y FROM c`
`MINUS`
`SELECT d1, d2 FROM d`
`ORDER BY 1`
	X **Y**
	1 Text 1
	4 Text 4
	5 Text 5
		Разница 2-х множеств – из первого множества убираются строки присутствующие во втором множестве.

Оператор CASE и функция DECODE

Оператор CASE может быть использован в одной из двух синтаксических форм записи:

1-я форма:	2-я форма:

`CASE <проверяемое выражение>`
` WHEN <сравниваемое выражение 1>`
` THEN <возвращаемое значение 1>`
` …`
` WHEN <сравниваемое выражение N>`
` THEN <возвращаемое значение N>`
` [ELSE <возвращаемое значение>]`
`END`
	`CASE`
	` WHEN <предикат 1>`
	` THEN <возвращаемое значение 1>`
	` …`
	` WHEN <предикат N>`
	` THEN <возвращаемое значение N>`
	` [ELSE <возвращаемое значение>]`
	`END`

Все предложения WHEN должны иметь одинаковую синтаксическую форму, то есть нельзя смешивать первую и вторую формы. При использовании первой синтаксической формы условие WHEN удовлетворяется, как только значение проверяемого выражения станет равным значению выражения, указанного в предложении WHEN. При использовании второй синтаксической формы условие WHEN удовлетворяется, как только предикат принимает значение TRUE. При удовлетворении условия оператор CASE возвращает значение, указанное в соответствующем предложении THEN. Если ни одно из условий WHEN не выполнилось, то будет использовано значение, указанное в предложении ELSE. При отсутствии ELSE, будет возвращено NULL-значение. Если удовлетворены несколько условий, то будет возвращено значение предложения THEN первого из них, так как остальные просто не будут проверяться.

Для примера разделим сотрудников на 4 группы по размеру их оклада:

```
SELECT
  CASE
    WHEN salary>20000 THEN 'A'
    WHEN salary>15000 THEN 'B'
    WHEN salary>10000 THEN 'C'
    ELSE 'D'
  END empl_group,
  salary,
  employee_id,
  first_name,
  last_name
FROM employees
```

Используя первую форму оператора, вычислим, сколько сотрудников было принято за 2006 год:

```
SELECT
  CASE EXTRACT(MONTH FROM hire_date)
    WHEN 1 THEN 'Январь'
    WHEN 2 THEN 'Февраль'
    WHEN 3 THEN 'Март'
    ELSE 'Апрель-Декабрь'
  END m,
  COUNT(*) emp_count
FROM employees
WHERE EXTRACT(YEAR FROM hire_date)=2006
GROUP BY
  CASE EXTRACT(MONTH FROM hire_date)
    WHEN 1 THEN 'Январь'
    WHEN 2 THEN 'Февраль'
    WHEN 3 THEN 'Март'
    ELSE 'Апрель-Декабрь'
  END
```

В старых версиях Oracle не было оператора CASE, и его моделировали при помощи функции DECODE:

```
SELECT
  DECODE(
     EXTRACT(MONTH FROM hire_date),
     1, 'Январь',
     2, 'Февраль',
     3, 'Март',
     'Апрель-Декабрь'
  ) m,
  COUNT(*) emp_count
FROM employees
WHERE EXTRACT(YEAR FROM hire_date)=2006
GROUP BY
  DECODE(
     EXTRACT(MONTH FROM hire_date),
     1, 'Январь',
     2, 'Февраль',
     3, 'Март',
     'Апрель-Декабрь'
  )
```

CASE и DECODE можно применять для вычисления сводных данных. Например, подсчитаем число принятых клерков, с разбивкой их по типу, с группировкой по годам:

```
SELECT
  EXTRACT(YEAR FROM hire_date) y,
  COUNT(*) qty_total,
  SUM(CASE WHEN job_id='PU_CLERK' THEN 1 END) qty_pu_clerk,
  SUM(CASE WHEN job_id='SH_CLERK' THEN 1 END) qty_sh_clerk,
  SUM(CASE WHEN job_id='ST_CLERK' THEN 1 END) qty_st_clerk
FROM employees
GROUP BY EXTRACT(YEAR FROM hire_date)
ORDER BY y
```

Далее будет показана как можно сделать сводную таблицу при помощи оператора PIVOT.

Новое в стандарте и реализациях языка SQL

Функции ранжирования - ROW_NUMBER, RANK и DENSE_RANK

Реляционная модель исходит из того факта, что строки в таблице не имеют порядка, являющегося прямым следствием теоретико-множественного подхода. Поэтому наивными выглядят вопросы новичков, спрашивающих: "А как мне получить последнюю добавленную в таблицу строку?" Ответом на вопрос будет "никак", если в таблице не предусмотрен столбец, содержащий дату вставки строки, или не используется последовательная нумерация строк, реализуемая во многих СУБД с помощью столбца с автоинкрементируемым значением. Тогда можно выбрать строку с максимальным значением даты или счетчика.

Функция ROW_NUMBER()

Функция ROW_NUMBER, как следует из ее названия, нумерует строки, возвращаемые запросом. С ее помощью можно выполнить более сложное упорядочивание строк в отчете, чем то, которое дает предложение ORDER BY в рамках Стандарта SQL-92.

Используя функцию ROW_NUMBER можно:

- задать нумерацию, которая будет отличаться от порядка сортировки строк результирующего набора;

- создать "несквозную" нумерацию, т.е. выделить группы из общего множества строк и пронумеровать их отдельно для каждой группы;
- использовать одновременно несколько способов нумерации, поскольку, фактически, нумерация не зависит от сортировки строк запроса.

Проще всего возможности функции ROW_NUMBER показать на простых примерах, к чему мы и переходим.

Например, пронумеруем список сотрудников в порядке Имени и Фамилии:

```
SELECT
  ROW NUMBER() OVER(ORDER BY e.first name,e.last name) n,
  e.employee id,e.first_name,e.last_name,e.department_id
FROM hr.employees e
```

Предложение OVER, с которым используется функция ROW_NUMBER задает порядок нумерации строк. При этом используется дополнительное предложение ORDER BY, которое не имеет отношения к порядку вывода строк запроса.

А если требуется пронумеровать сотрудников для каждого департамента отдельно? Для этого нам потребуется еще одна конструкция в предложении OVER - PARTITION BY.

Конструкция PARTITION BY задает группы строк, для которых выполняется независимая нумерация. Группа определяется равенством значений в списке столбцов, перечисленных в этой конструкции, у строк, составляющих группу.

```
SELECT
  ROW NUMBER() OVER(PARTITION BY e.department id ORDER BY e.first_name,e.last_name) n,
  e.employee id,e.first_name,e.last_name,e.department_id
FROM hr.employees e
ORDER BY e.department_id,n
```

Функции RANK() и DENSE_RANK()

Эти функции, как и функция ROW_NUMBER(), тоже нумеруют строки, но делают это несколько отличным способом. Это отличие проявляется в том, что строки, которые имеют одинаковые значения в столбцах, по которым выполняется упорядочивание, получают одинаковые номера (ранги).

Рассмотрим работу на примере таблицы E, содержащей следующие данные:

ID	GRP_ID	DESCR
1	1	Описание 1.1
2	1	Описание 1.2
3	1	Описание 1.3
4	1	Описание 1.4
5	2	Описание 2.1
6	2	Описание 2.2
7	2	Описание 2.3
8	2	Описание 2.4
9	3	Описание 3.1
10	3	Описание 3.2
11	4	Описание 4.1
12	4	Описание 4.2
13	6	Описание 6.1
14	6	Описание 6.2
15	6	Описание 6.3

Выполним запрос:

```
-- ранжирование (классификация) в разрезе группы неплотная и плотная
SELECT
  RANK() OVER(ORDER BY GRP ID) RANK NO,
  DENSE RANK() OVER(ORDER BY GRP ID) DRANK NO,
  ROW_NUMBER() OVER(ORDER BY GRP_ID) ROW_NO,
  e.*
FROM e
```

И посмотрим на результат:

RANK_NO	DRANK_NO	ROW_NO	ID	GRP_ID	DESCR
1	1	1	3	1	Описание 1.3
1	1	2	2	1	Описание 1.2
1	1	3	1	1	Описание 1.1
1	1	4	4	1	Описание 1.4
5	2	5	7	2	Описание 2.3
5	2	6	6	2	Описание 2.2
5	2	7	5	2	Описание 2.1
5	2	8	8	2	Описание 2.4
9	3	9	9	3	Описание 3.1
9	3	10	10	3	Описание 3.2
11	4	11	11	4	Описание 4.1
11	4	12	12	4	Описание 4.2
13	5	13	13	6	Описание 6.1
13	5	14	14	6	Описание 6.2
13	5	15	15	6	Описание 6.3

Из примера видно, что функция RANK в качестве начального номера группы берет значение номера строки, а DENSE_RANK использует последовательные номера для нумерации групп.

Функции специфичные для Oracle - LAG и LEAD, FIRST_VALUE и LAST_VALUE

Функции LAG и LEAD позволяют получить необходимое значение из предыдущей или следующей строки.

Функции FIRST_VALUE и LAST_VALUE первое и последнее значение.

Думаю, здесь будет достаточно примера:

```
select
  lag(descr) over(order by id) prev_descr,    -- предыдущее значение
  lead(descr) over(order by id) next descr,   -- следующее значение
  lag(descr,2) over(order by id) prev2_descr,
  lead(descr,2) over(order by id) next2_descr,
  lag(descr,1,'---') over(partition by grp_id order by id) prev1part_descr,
  lead(descr,1,'---') over(partition by grp id order by id) next1part_descr,
  FIRST VALUE(descr) over(order by id) first descr,
  LAST VALUE(descr) over(order by id) last descr,
  FIRST_VALUE(descr) over(partition by grp_id order by id) firstpart_descr,
  LAST_VALUE(descr) over(partition by grp_id order by id) lastpart_descr,
  e.*
from e
```

Для функции **LAST_VALUE**, для данного примера, нужно будет изменить стандартное поведение окна (`RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`) на `RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`:

- `LAST_VALUE(descr) over(order by id RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) last_descr`
- `LAST_VALUE(descr) over(partition by grp_id order by id RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)`

Оконные (Аналитические) функции

Фактически мы познакомились с этими функциями, когда рассматривали функции ранжирования. Только сейчас мы будем использовать агрегатные функции вместо функций, которые задают номер/ранг строки. Есть еще одно отличие (в реализации Майкрософт SQL Server 2005/2008) – предложение OVER() не содержит дополнительного предложения ORDER BY, поскольку значение агрегата не зависит от сортировки строк в «окне».

Как и ранее, предложение PARTITION BY определяет «окно», т.е. набор строк, характеризуемых равенством значений списка выражений, указанного в этом предложении. Если предложение PARTITION BY отсутствует, то агрегатные функции применяются ко всему результирующему набору строк запроса. В отличие от классической группировки, где мы получаем на каждую группу одну строку, которая может содержать агрегатные значения, подсчитанные для каждой такой группы, здесь мы можем добавить агрегат к детализированным (несгруппированным) строкам.

Например, посмотрим оклад сотрудника в сравнении минимальным и максимальным окладом в разрезе его департамента, а так же минимальным и максимальным окладом всех сотрудников:

```
select
  e.employee_id,e.first_name,e.last_name,
  e.salary,
  min(e.salary) over(partition by e.department_id) min_salary_in_dep,
  max(e.salary) over(partition by e.department_id) max_salary_in_dep,
  min(e.salary) over() min salary,
  max(e.salary) over() max_salary
from employees e
```

Теперь давайте получим из каждого департамента, сотрудников с минимальным окладом в этом департаменте, использование оконной функции позволяет сделать это достаточно просто и наглядно:

```
select *
from
  (
    select
      e.employee_id,e.department_id,e.first_name,e.last_name,e.salary,
      min(e.salary) over(partition by e.department_id) min_salary_in_dep
    from employees e
  )
where salary=min_salary_in_dep
```

В отличие от того же примера реализованного при помощи подзапроса:

```
select e.employee_id,e.department_id,e.first_name,e.last_name,e.salary
from employees e
where e.salary=(select min(salary) from employees where department_id=e.department_id)
```

В котором, к тому же, не учитывается сотрудник, у которого department_id не указано, т.к. подзапрос возвращает NULL значение. Чтобы это исправить сделаем следующее:

```
select e.employee_id,e.department_id,e.first_name,e.last_name,e.salary
from employees e
where e.salary=(select min(salary) from employees
                where department_id=e.department_id
                  or (e.department_id is null and department_id is null))
```

В Oracle предложение PARTITION BY и ORDER BY можно использовать в предложении OVER для агрегатных функций SUM и COUNT, и этим можно воспользоваться для вычисления нарастающего итога:

```
select
  e.employee_id,e.salary,
  sum(e.salary) over(order by e.employee_id) acc_sum,
  count(*) over(order by e.employee_id) acc_count
from employees e
```

В этом случае в предложении ORDER BY должна использоваться комбинация полей, которая будет уникально для каждой возвращаемой строки. Если это условие не выполняется, то можно воспользоваться дополнительным предложением **RANGE UNBOUNDED PRECEDING**:

```
select
  e.hire_date,e.salary,
  /* в этом случае для одинаковых значений даты будет выведено одно и тоже значение */
  sum(e.salary) over(order by e.hire_date) err_acc_sum,
  count(*) over(order by e.hire_date) err acc count,
  /* с предложением RANGE UNBOUNDED PRECEDING - все верно */
  sum(e.salary) over(order by e.hire_date rows unbounded preceding) acc_sum,
  count(*) over(order by e.hire_date rows unbounded preceding) acc_count
from employees e
order by hire_date
```

Нюансы реализации оконных функций в ORACLE и MS SQL Server 2012

```
ИМЯ_ФУНКЦИИ(<аргумент>,< аргумент >, . . . )
OVER(<конструкция_секционирования> <конструкция_упорядочения> <конструкция_окна>)
```

Конструкция секционирования – PARTITION BY

Конструкция PARTITION BY логически разбивает результирующее множество на группы по критериям, задаваемым выражениями секционирования. Аналитические функции применяются к каждой группе независимо — для каждой новой группы они сбрасываются. Если не указать конструкцию секционирования, все результирующее множество считается одной группой. **Каждая аналитическая функция в запросе может иметь уникальную конструкцию секционирования.**

Синтаксис конструкции секционирования аналогичен синтаксису конструкции **GROUP BY** в обычных SQL-запросах: **PARTITION BY выражение [, выражение] [, выражение]**

Конструкция упорядочения – ORDER BY

Конструкция ORDER BY задает критерий сортировки данных в каждой группе (в каждой секции). Это существенно влияет на результат выполнения любой аналитической функции: при наличии (или отсутствии) конструкции ORDER BY аналитические функции вычисляются по-другому. В отсутствие конструкции ORDER BY среднее значение вычисляется по всей группе, и одно и то же значение выдается для каждой строки (функция используется как итоговая). Когда функция используется с конструкцией ORDER BY, то она применяется по текущей и всем предыдущим строкам (функция используется как оконная).

Конструкция ORDER BY в аналитических функциях имеет следующий синтаксис:

```
ORDER BY выражение [ASC | DESC] [NULLS FIRST | NULLS LAST]
```

Необходимо учитывать, что строки будут упорядочены только в пределах секций (групп). Конструкции **NULLS FIRST** и **NULLS LAST** указывает, где при упорядочении должны быть значения **NULL** — в начале или в конце.

Конструкция окна

Синтаксис

```
{ROWS | RANGE} {{UNBOUNDED | выражение} PRECEDING | CURRENT ROW }

{ROWS | RANGE}
BETWEEN
{{UNBOUNDED PRECEDING | CURRENT ROW |
{UNBOUNDED | выражение 1}{PRECEDING | FOLLOWING}}
AND
{{UNBOUNDED FOLLOWING | CURRENT ROW |
{UNBOUNDED | выражение 2}{PRECEDING | FOLLOWING}}
```

Конструкция окна позволяет задать перемещающееся или жестко привязанное окно (набор, интервал) данных в пределах группы, с которым будет работать аналитическая функция.

Например, конструкция диапазона RANGE UNBOUNDED PRECEDING означает: "применять аналитическую функцию к каждой строке данной группы, с первой по текущую". Стандартным является жестко привязанное окно, начинающееся с первой строки группы и продолжающееся до текущей. Очень важно: для использования окон необходимо задавать конструкцию ORDER BY.

Фразы PRECEDING и FOLLOWING задают верхнюю и нижнюю границы агрегирования (то есть интервал строк, "окно" для агрегирования).

Возможны такие варианты окон:

1. Если нижняя граница окна фиксирована (совпадает с первой строкой упорядоченной некоторым образом группы строк), а верхняя граница ползет (совпадает с текущей строкой в этой группе), то получаем нарастающий итог (кумулятивный агрегат). Т.е. здесь и размер окна меняется (расширяется в одну сторону) и само окно движется (за счет расширения).
2. Если нижняя и верхняя границы фиксированы (относительно текущей строки в этой группе, например, 1 строка до текущей и 2 строки после текущей), то получаем скользящий агрегат. Т.е. здесь размеры окна фиксированы (оно никуда не расширяется), а само окно движется (скользит).
3. Частным случаем 2. является ситуация, когда окно симметрично относительно текущей строки (например, 2 строки до текущей и 2 строки после текущей). Это тоже скользящий агрегат.

Можно создавать окна по двум критериям:

- по диапазону (RANGE) значений данных
- по смещению (ROWS) относительно текущей строки.

Окна диапазона

Окна диапазона объединяют строки в соответствии с заданным порядком. Если в запросе сказано, например, "range 5 preceding", то будет сгенерировано перемещающееся окно, включающее предыдущие строки группы, отстоящие от текущей строки не более чем на 5 значений. Диапазон можно задавать в виде числового выражения или выражения, значением которого является дата. Применять конструкцию RANGE с другими типами данных нельзя.

Если имеется таблица EMP со столбцом HIREDATE типа даты и задана аналитическая функция count(*) over (order by hiredate asc range 100 preceding) она найдет все предыдущие строки фрагмента, значение которых в столбце HIREDATE лежит в пределах 100 дней от значения HIREDATE текущей строки. В этом случае, поскольку данные сортируются по возрастанию (ASC), значения в окне будут включать все строки текущей группы, у которых значение в столбце HIREDATE меньше значения HIREDATE текущей строки, но не более чем на 100 дней.

Если использовать функцию count(*) over (order by hiredate desc range 100 preceding) и сортировать фрагмент по убыванию (DESC), базовая логика работы останется той же, но, поскольку группа отсортирована иначе, в окно попадет другой набор строк. В рассматриваемом случае функция найдет все строки, предшествующие текущей, где значение в поле HIREDATE больше значения HIREDATE в текущей строке, но не более чем на 100 дней.

Для того чтобы понять, какие значения будут входить в диапазон, можно использовать функции FIRST_VALUE удобным методом, помогающим увидеть диапазоны окна и проверить, корректно ли установлены параметры.

Окна строк

Окна срок задаются в физических единицах, строках. Окно заданное конструкцией count (*) over (order by x ROWS 5 preceding) будет включать до 6 строк: текущую и пять предыдущих (порядок определяется конструкцией ORDER BY). Для окон по строкам нет ограничений, присущих окнам по диапазону; данные могут быть любого типа и упорядочивать можно по любому количеству столбцов.

Примеры

```
/*
Можно создавать окна по двум критериям:
  1) по диапазону (RANGE) значений данных
  2) по смещению (ROWS) относительно текущей строки.
*/
SELECT
  e.*,
  -- определяем окно используя смещения относительно текущией строки
  MIN(descr) OVER(ORDER BY id ROWS 2 PRECEDING) rows_1, -- значения 2-х предыдущих строк и текущей строки
  MAX(descr) OVER(ORDER BY id ROWS UNBOUNDED PRECEDING) rows_2, -- все предыдущие строки и текущая строка
  MIN(descr) OVER(ORDER BY id ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) rows_3, -- от текущей строки до последней
  MIN(descr) OVER(ORDER BY id ROWS BETWEEN 3 PRECEDING AND 1 PRECEDING) rows_4, -- 3 предыдущих строки
  MIN(descr) OVER(ORDER BY id ROWS BETWEEN 1 FOLLOWING AND 3 FOLLOWING) rows_5, -- 3 следующих строки
  -- определяем окно используя смещения по значению. В данном случае в ORDER BY допустимы значения только типа дата или число
  MIN(descr) OVER(ORDER BY grp_id RANGE 1 PRECEDING) range_1, -- значение диапазона от (grp_id-1) до grp_id
  MAX(descr) OVER(ORDER BY grp_id RANGE UNBOUNDED PRECEDING) range_2, -- строки у которых значения grp_id <= grp_id текущей строки
  MIN(descr) OVER(ORDER BY grp_id RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) range_3, -- gr_id >= gr_id текущей строки
  MIN(descr) OVER(ORDER BY grp_id RANGE BETWEEN 3 PRECEDING AND 1 PRECEDING) range_4, -- gr_id BETWEEN gr_id-3 AND gr_id-1
  MIN(descr) OVER(ORDER BY grp_id RANGE BETWEEN 1 FOLLOWING AND 3 FOLLOWING) range_5 -- gr_id BETWEEN gr_id+1 AND gr_id+3
FROM e
```

```
ORDER BY id;
```

Результат:

ID	GRP_ID	DESCR	ROWS_1	ROWS_2	ROWS_3	ROWS_4	ROWS_5	RANGE_1	RANGE_2	RANGE_3	RANGE_4	RANGE_5
1	1	Описание 1.1	Описание 1.1	Описание 1.1	Описание 1.1		Описание 1.2	Описание 1.1	Описание 1.4	Описание 1.1		Описание 2.1
2	1	Описание 1.2	Описание 1.1	Описание 1.2	Описание 1.2	Описание 1.1	Описание 1.3	Описание 1.1	Описание 1.4	Описание 1.1		Описание 2.1
3	1	Описание 1.3	Описание 1.1	Описание 1.3	Описание 1.3	Описание 1.1	Описание 1.4	Описание 1.1	Описание 1.4	Описание 1.1		Описание 2.1
4	1	Описание 1.4	Описание 1.2	Описание 1.4	Описание 1.4	Описание 1.1	Описание 2.1	Описание 1.1	Описание 1.4	Описание 1.1		Описание 2.1
5	2	Описание 2.1	Описание 1.3	Описание 2.1	Описание 2.1	Описание 1.2	Описание 2.2	Описание 1.1	Описание 2.4	Описание 2.1	Описание 1.1	Описание 3.1
6	2	Описание 2.2	Описание 1.4	Описание 2.2	Описание 2.2	Описание 1.3	Описание 2.3	Описание 1.1	Описание 2.4	Описание 2.1	Описание 1.1	Описание 3.1
7	2	Описание 2.3	Описание 2.1	Описание 2.3	Описание 2.3	Описание 1.4	Описание 2.4	Описание 1.1	Описание 2.4	Описание 2.1	Описание 1.1	Описание 3.1
8	2	Описание 2.4	Описание 2.2	Описание 2.4	Описание 2.4	Описание 2.1	Описание 3.1	Описание 1.1	Описание 2.4	Описание 2.1	Описание 1.1	Описание 3.1
9	3	Описание 3.1	Описание 2.3	Описание 3.1	Описание 3.1	Описание 2.2	Описание 3.2	Описание 2.1	Описание 3.2	Описание 3.1	Описание 1.1	Описание 4.1
10	3	Описание 3.2	Описание 2.4	Описание 3.2	Описание 3.2	Описание 2.3	Описание 4.1	Описание 2.1	Описание 3.2	Описание 3.1	Описание 1.1	Описание 4.1
11	4	Описание 4.1	Описание 3.1	Описание 4.1	Описание 4.1	Описание 2.4	Описание 4.2	Описание 3.1	Описание 4.2	Описание 4.1	Описание 1.1	Описание 6.1
12	4	Описание 4.2	Описание 3.2	Описание 4.2	Описание 4.2	Описание 3.1	Описание 6.1	Описание 3.1	Описание 4.2	Описание 4.1	Описание 1.1	Описание 6.1
13	6	Описание 6.1	Описание 4.1	Описание 6.1	Описание 6.1	Описание 3.2	Описание 6.2	Описание 6.1	Описание 6.3	Описание 6.1	Описание 3.1	
14	6	Описание 6.2	Описание 4.2	Описание 6.2	Описание 6.2	Описание 4.1	Описание 6.3	Описание 6.1	Описание 6.3	Описание 6.1	Описание 3.1	
15	6	Описание 6.3	Описание 6.1	Описание 6.3	Описание 6.3	Описание 4.2		Описание 6.1	Описание 6.3	Описание 6.1	Описание 3.1	

```
SELECT
  e.*,
  -- определяем окно используя смещения относительно текущией строки
  COUNT(descr) OVER(ORDER BY id ROWS 2 PRECEDING) rows_1, -- значения 2-х предыдущих строк и текущей строки
  COUNT(descr) OVER(ORDER BY id ROWS UNBOUNDED PRECEDING) rows_2, -- все предыдущие строки и текущая строка
  COUNT(descr) OVER(ORDER BY id ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) rows_3, -- от текущей строки до последней
  COUNT(descr) OVER(ORDER BY id ROWS BETWEEN 3 PRECEDING AND 1 PRECEDING) rows_4, -- 3 предыдущих строки
  COUNT(descr) OVER(ORDER BY id ROWS BETWEEN 1 FOLLOWING AND 3 FOLLOWING) rows_5, -- 3 следующих строки
  -- определяем окно используя смещения по значению. В данном случае в ORDER BY допустимы значения только типа дата или число
  COUNT(descr) OVER(ORDER BY grp_id RANGE 1 PRECEDING) range_1, -- значение диапазона от (grp_id-1) до grp_id
  COUNT(descr) OVER(ORDER BY grp_id RANGE UNBOUNDED PRECEDING) range_2, -- строки у которых значения grp_id <= grp_id текущей строки
  COUNT(descr) OVER(ORDER BY grp_id RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) range_3, -- gr_id >= gr_id текущей строки
  COUNT(descr) OVER(ORDER BY grp_id RANGE BETWEEN 3 PRECEDING AND 1 PRECEDING) range_4, -- gr_id BETWEEN gr_id-3 AND gr_id-1
  COUNT(descr) OVER(ORDER BY grp_id RANGE BETWEEN 1 FOLLOWING AND 3 FOLLOWING) range_5 -- gr_id BETWEEN gr_id+1 AND gr_id+3
FROM e
ORDER BY id;
```

Результат:

ID	GRP_ID	DESCR	ROWS_1	ROWS_2	ROWS_3	ROWS_4	ROWS_5	RANGE_1	RANGE_2	RANGE_3	RANGE_4	RANGE_5
1	1	Описание 1.1	1	1	15	0	3	4	4	15	0	8
2	1	Описание 1.2	2	2	14	1	3	4	4	15	0	8
3	1	Описание 1.3	3	3	13	2	3	4	4	15	0	8
4	1	Описание 1.4	3	4	12	3	3	4	4	15	0	8
5	2	Описание 2.1	3	5	11	3	3	8	8	11	4	4
6	2	Описание 2.2	3	6	10	3	3	8	8	11	4	4
7	2	Описание 2.3	3	7	9	3	3	8	8	11	4	4
8	2	Описание 2.4	3	8	8	3	3	8	8	11	4	4
9	3	Описание 3.1	3	9	7	3	3	6	10	7	8	5
10	3	Описание 3.2	3	10	6	3	3	6	10	7	8	5
11	4	Описание 4.1	3	11	5	3	3	4	12	5	10	3
12	4	Описание 4.2	3	12	4	3	3	4	12	5	10	3
13	6	Описание 6.1	3	13	3	3	2	3	15	3	4	0
14	6	Описание 6.2	3	14	2	3	1	3	15	3	4	0
15	6	Описание 6.3	3	15	1	3	0	3	15	3	4	0

Операторы PIVOT и UNPIVOT

Для объяснения оператора **PIVOT** снова рассмотрим пример, для подсчета числа принятых клерков, с разбивкой их по типу, с группировкой по годам:

```
SELECT
  EXTRACT(YEAR FROM hire_date) y,
  SUM(CASE WHEN job_id='PU_CLERK' THEN 1 END) pu_qty,
  SUM(CASE WHEN job_id='SH_CLERK' THEN 1 END) sh_qty,
  SUM(CASE WHEN job_id='ST_CLERK' THEN 1 END) st_qty
FROM employees
GROUP BY EXTRACT(YEAR FROM hire_date)
ORDER BY y
```

При помощи PIVOT запрос преобразуется:

```
SELECT *
FROM (SELECT EXTRACT(YEAR FROM hire_date) y,job_id FROM employees) -- получили выборку вида (год, должность)
PIVOT(COUNT(*) FOR job_id IN ('PU_CLERK' pu_qty, 'SH_CLERK' sh_qty, 'ST_CLERK' st_qty))
ORDER BY y
```

UNPIVOT выполняет обратную по отношению к **PIVOT** операцию, т.е. представляет данные, записанные в строке таблицы, в одном столбце.

Рассмотрим таблицу CLERK_INFO, содержащую сводные данные по приему клерков в разрезе годов и месяцев:

Y	M	PU_QTY	SH_QTY	ST_QTY
2001	1	0	0	0
2002	6	0	0	0
2002	8	0	0	0
2002	12	0	0	0
2003	5	1	0	0
2003	6	0	0	0
…				

При помощи операции **UNPIVOT** развернем данную таблицу:

```
SELECT *
FROM clerk_info
UNPIVOT(qty FOR clerk_type IN(pu_qty, sh_qty, st_qty))
WHERE qty<>0
ORDER BY y,m,clerk_type
```

В результате, мы получим развернутую таблицу со столбцами Y, M, CLERK_TYPE, QTY.

Данные для столбца QTY создаются на основании значений PU_QTY, SH_QTY, ST_QTY, а значения CLERK_TYPE будут содержать строки 'PU_QTY', 'SH_QTY', 'ST_QTY'.

Общие табличные выражения (CTE - Common Table Expressions) – оператор WITH

Общие табличные выражения позволяют существенно уменьшить объем кода, если многократно приходится обращаться к одним и тем же запросам. CTE играет роль представления, которое создается в рамках одного запроса и, не сохраняется как объект схемы.

Давайте снова получим из каждого департамента, сотрудников с минимальным окладом в этом департаменте и дополнительно подсчитаем итог по каждому департаменту и общий итог, в рамках одного запроса:

```
with emp as (
  select *
  from
    (
      select
        e.employee_id,e.department_id,e.first_name,e.last_name,
        e.salary,
        min(e.salary) over(partition by e.department_id) min_salary_in_dep
      from employees e
    )
  where salary=min_salary_in_dep
)
select *
from
  (
    select e.*
    from emp e -- обращение к cte 1
    union all
    select null,e.department id,'Total by department',to_char(count(*)) || ' employees',sum(salary),null
    from emp e -- обращение к cte 2
    group by e.department_id
    union all
    select null,null,'Grand total',to_char(count(*)) || ' employees',sum(salary),null
    from emp e -- обращение к cte 3
  )
order by department_id,employee_id
```

Без использования cte, один и тот же запрос пришлось бы дублировать 3 раза.

Можно использовать несколько cte, в этом случае они перечисляются через запятую:

```
with emp as ( -- CTE 1
  select *
  from employees
),
dep as ( -- CTE 2
  select *
  from departments
  where location_id = 1700
)
select *
from emp e, dep d
where e.department_id = d.department_id
```

Рекурсивные CTE

У CTE есть еще одно важное назначение. С его помощью можно написать рекурсивный запрос, т.е. запрос, который, написанный один раз, будет повторяться многократно пока истинно некоторое условие.

Рекурсивный CTE имеет следующий вид:

```
WITH <имя>[(<список столбцов>)]
AS(
< SELECT... > -- анкорная часть
UNION ALL -- рекурсивная часть
< SELECT...FROM <имя>… >
WHERE <условие продолжения итераций>
```

От обычного CTE-запроса рекурсивный отличается только рекурсивной частью, которая вводится предложением UNION ALL. Обратите внимание, что в рекурсивной части присутствует ссылка на имя CTE, т.е. внутри CTE ссылается само на себя. Это, собственно, и есть рекурсия. Естественно, анкорная и рекурсивная части должны иметь одинаковый набор столбцов.

До версии 11.2 в Oracle задача рекурсивных запросов к иерархически организованным данным решалась с помощью фразы CONNECT BY. В то же время в стандарте SQL:1999 была введена фраза WITH для вынесения подзапросов («факторизация» запроса), одна из двух вариантов которой решала задачу рекурсивных запросов более общим (и стандартным !) образом. В Oracle первый вариант фразы WITH (простое вынесение подзапроса из основного текста) был воплощен в версии 9, а второй, рекурсивный, хотя и с некоторыми вольностями, – в версии 11.2.

Простой пример употребления фразы WITH для построения рекурсивного запроса:

```
WITH numbers ( n ) AS (
   SELECT 1 AS n FROM dual  -- исходное множество -- одна строка
   UNION ALL                -- символическое «объединение» строк
   SELECT n + 1 AS n        -- рекурсия: добавок к предыдущему результату
   FROM   numbers           -- предыдущий результат в качестве источника данных
   WHERE  n < 5             -- если не ограничить, будет бесконечная рекурсия
)
SELECT n FROM numbers       -- основной запрос
```

Строка с n = 1 получена из опорного запроса, а остальные строки из рекурсивного. Из примера видна оборотная сторона рекурсивных формулировок: при неаккуратном планировании они допускают «бесконечное» выполнение (на деле – пока хватит ресурсов СУБД для сеанса или же пока администратор не прервет запрос или сеанс). С фразой CONNECT BY «бесконечное» выполнение в принципе невозможно. Программист обязан отнестись к построению рекурсивного запроса ответственно.

Еще один вывод из этого примера: подобно случаю с CONNECT BY, вынесенный рекурсивный подзапрос применим вовсе необязательно только к иерархически организованным данным.

Пример с дополнительным разъяснением способа выполнения:

```
WITH anchor1234 ( n ) AS (         -- обычный CTE
  SELECT 1 FROM dual UNION ALL
  SELECT 2 FROM dual UNION ALL
  SELECT 3 FROM dual UNION ALL
  SELECT 4 FROM dual
),
numbers ( n, descr, lev ) AS (    -- рекурсивный CTE
  SELECT n, 'обычный', 0
  FROM anchor1234
    UNION ALL
  SELECT n + 1, 'рекурсивный', lev+1
  FROM   numbers
  WHERE n < 5
)
SELECT n, descr, lev FROM numbers
```

Для примера выведем список сотрудников с учетом их подчинения.

Сначала решим эту задачу при помощи конструкции CONNECT BY PRIOR:

```
select rpad(' ',2*(level-1)) || e.employee_id id,e.first_name || ' ' || e.last_name ename,e.manager_id
from employees e
  connect by prior e.employee_id=e.manager_id
  start with manager id is null
order siblings by e.first_name,e.last_name
```

Для придания дополнительной сортировки строк результата в запросах с CONNECT BY используется особая конструкция **ORDER BY SIBLINGS**.

Теперь решим эту задачу при помощи рекурсивного WITH:

```
with empl cte (empno, ename, xlevel, mgr) as (
  -- определяем стартовую точку
  select e.employee_id,e.first_name || ' ' || e.last_name ename,0,e.manager_id
  from employees e
  where manager_id is null

  union all
  -- циклический запрос обращающийся к
  select e.employee id,e.first name || ' ' || e.last name ename,c.xlevel+1,e.manager_id
  from employees e join empl_cte c on e.manager_id=c.empno
)
search depth first by ename set ord /* задаем дополнительный порядок сортировки по ENAME и задаем имя столбца
для сортировки ORD  */
select rpad(' ',2*xlevel)||empno id,ename,mgr,ord
from empl_cte
order by ord
```

Аналогично в вынесенном рекурсивном запросе используется специальное указание **SEARCH**. В рамках последнего, в частности, задается вымышленное имя столбца, в котором СУБД автоматически проставит числовые значения, и который включит автоматически в порождаемый набор столбцов, допуская в последующей обработке его использование во фразе ORDER BY для осуществления упорядочения.

Рекурсивные запросы с фразой WITH, хоть и выглядят более сложно, но позволяют программисту сделать больше, нежели запросы с CONNECT BY (тоже рекурсивные). Например, они позволяют накапливать изменения, и не испытывают необходимости в функциях LEVEL или SYS_CONNECT_BY_PATH, имея возможность легко их моделировать.

Пример использования функции SYS_CONNECT_BY_PATH:

```
select
  rpad(' ',2*(level-1)) || e.employee_id id,e.first_name || ' ' || e.last_name ename,e.manager_id,
  SYS_CONNECT_BY_PATH(e.first_name || ' ' || e.last_name,'/') manager_list
from employees e
  connect by prior e.employee id=e.manager_id
  start with manager id is null
order siblings by e.first_name,e.last_name
```

Теперь сделаем похожее при помощи рекурсивного WITH:

```
with empl_cte (empno, ename, xlevel, mgr, manager_list) as (
  -- определяем стартовую точку
  select e.employee_id,e.first_name || ' ' || e.last_name ename,0,e.manager_id,null
  from employees e
  where manager_id is null

  union all
  -- циклический запрос обращающийся к empl_cte
  select e.employee id,e.first name || ' ' || e.last name ename,c.xlevel+1,e.manager_id,
         c.manager list||' -> '||c.ename -- формируем список менеджеров
  from employees e join empl_cte c on e.manager_id=c.empno
)
search depth first by ename set ord /* задаем дополнительный порядок сортировки по ENAME и задаем имя столбца
для сортировки ORD  */
select rpad(' ',2*xlevel)||empno id,ename,mgr,ord,manager_list
from empl cte
order by ord
```

Основные функции для работы с NULL значениями

NVL(expr1, expr2) и NVL2(expr1, expr2, expr3)

NVL(expr1, expr2) – если значение выражения expr1 равно NULL, возвращается значение expr2, иначе возвращается expr1.

NVL2(expr1, expr2, expr3) – если значение выражения expr1 равно NULL, возвращается значение expr3, иначе возвращается expr2.

```
SELECT
  last_name,
  NVL(TO_CHAR(commission_pct), 'Not Applicable') COMMISSION,
  NVL2(commission_pct, 'COMMISSION', 'Not Applicable') COMMISSION
FROM employees
WHERE last_name LIKE 'B%'
ORDER BY last_name
```

COALESCE (expr1, expr2, …, exprn)

Возвращает первое не NULL значение из списка значений.

```
SELECT COALESCE(f1, f1*f2, f2*f3) val -- в данном случае вернется третье значение
FROM (SELECT null f1, 2 f2, 3 f3 from dual)
```

NULLIF(expr1, expr2)

Если expr1 = expr2, то возвращается NULL, иначе expr1.

При помощи CASE реализуется так - CASE WHEN expr1 = expr2 THEN NULL ELSE expr1 END

```
SELECT c.*,nullif(c2,'Text 2') c3
FROM c
```

Символьные функции

Функция	Описание
LOWER(expr)	Преобразует алфавитные символы в нижний регистр.
UPPER(expr)	Преобразует алфавитные символы в верхний регистр.
INITCAP(expr)	Первая буква каждого слова становится заглавной, а остальные строчными.
CONCAT(expr1, expr2)	Конкатенация строк. Эквивалентно: expr1 || expr2.
SUBSTR(expr, start [, cnt])	Возвращает cnt символов из строки expr, начиная с позиции start. Если start<0, то отсчет начинается с конца строки. Если cnt не указано, то возвращаются все символы до конца строки.
LENGTH(expr)	Возвращает количество символов в строке.
INSTR(expr, str)	Ищет в expr подстроку str и возвращает номер позиции. Если подстрока не найдена возвращает (-1).
LPAD(expr, len, str)	Дополняет строку символами str слева, до длины len. Если длина строки expr превышает len, то возвращаются первые len символов этой строки.
RPAD(expr, len, str)	Дополняет строку символами str справа, до длины len. Если длина строки expr превышает len, то возвращаются первые len символов этой строки.
TRIM(LEADING|TRAILING|BOTH char FROM expr)	Удалить символ char слева | справа | с обоих концов, по умолчанию - **TRIM(expr)** будут усекаться пробелы, с обоих концов.
LTRIM(expr [, char])	Удалить символ char с начала строки (слева), по умолчанию будут усекаться пробелы.
RTRIM(expr [, char])	Удалить символ char с конца строки (справа), по умолчанию будут усекаться пробелы.

```
select
  LOWER('Проверка'),          -- проверка
  UPPER('Test'),              -- TEST
  INITCAP('иванов ИВАН'),     -- Иванов Иван
  CONCAT('Привет', ' мир!'),  -- Привет мир!
  SUBSTR('1234567890', 3),    -- 34567890
  SUBSTR('1234567890', 3, 2), -- 34
  SUBSTR('1234567890', -5),   -- 67890
  SUBSTR('1234567890', -5, 2),-- 67
  LENGTH('Длина строки'),     -- 12
  INSTR('Всем доброго утра!', 'утра'), -- 14
  LPAD('123', 10, '>'),       -- >>>>>>>123
  RPAD('123', 10, '<')        -- 123<<<<<<<
from dual
```

Полезные функции для работы с DATETIME

TO_CHAR (datetime, format) – преобразование даты в текст по маске

Данную функцию удобно применять для форматированного вывода или извлечения нужного значения из даты:

```
select
  to_char(sysdate, 'yyyy'), -- год
  to_char(sysdate, 'mm'),   -- месяц
  to_char(sysdate, 'yyyymm'), -- год и месяц
  to_char(sysdate, 'dd.mm.yyyy'), -- дата в формате dd.mm.yyyy
  to_char(sysdate, 'yyyymmdd hh24miss') -- дата в формате yyyymmdd hh24miss
from dual
```

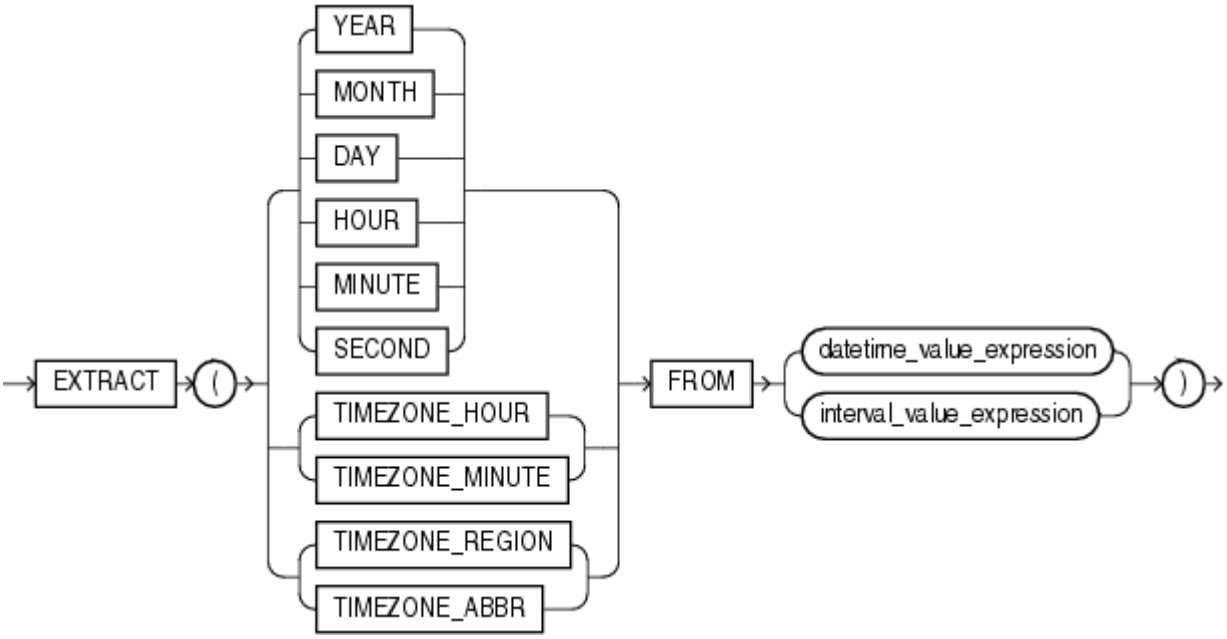
О прочих форматах можно почитать в официальной справочной документации Oracle.

TO_DATE (text, format) – преобразование текста в дату по маске

Эта функция получается обратной по отношению к TO_CHAR. Данную функцию удобно использовать для задания параметров в запросе, которые не будут зависеть от региональных настроек сервера:

```
select
  to_date('05.11.2012', 'dd.mm.yyyy'), -- дата в формате dd.mm.yyyy
  to_date('2011-03-25 05:01:18', 'yyyy-mm-dd hh24:mi:ss') -- дата в формате yyyy-mm-dd hh24:mi:ss
from dual
```

EXTRACT (datetime) – извлечение значения из даты



Пример:

```
select
  extract(year from sysdate),
  extract(month from sysdate),
  extract(day from sysdate)
from dual
```

Основы регулярных выражений

Начало и конец строки

^	используется если необходимо задать начало строки
$	используется если необходимо задать конец строки

Шаблоны

.	точка обозначает, любой символ
[0-9]	задать диапазон символов
[129]	один из символов
432	жестко заданная последовательность
(12|21)	одно из перечисленных значений

Повторения

*	любое число вхождений
?	ноль или одно вхождение (не более одного вхождения)
+	один или более вхождений (не менее одного вхождения)

Примеры

Выражение	**Описание**	**Например**
^1	значение должно начинаться с 1	12, 13, 123
^1.*8$	значение должно начинаться с 1 и заканчиваться на 8 (.* - любое число любых символов)	128, 18, 1238
^1.+8$	значение должно начинаться с 1 и заканчиваться на 8, причем между ними должен находиться хотя бы один символ	128, 1348
^1[23]$ ^(12|13)$	необходимо найти 12 или 13	12, 13
[6-9]$	значение должно заканчиваться цифрам от 6 до 9	129, 9, 19
^.?[6-9]$	значение должно заканчиваться цифрам от 6 до 9, причем в начале должно быть не более одного символа	9, 29, 19
^1[1-3]5$	Поиск 115, 125 или 135	115, 125, 135
^34$	жестко заданная последовательность значение должно быть равно 34	34