Работа с составными типами данных. Применение коллекций, записей и объектных типов

Рассматриваемые вопросы

- **Создание пользовательских записей PL/SQL**
- **Создание записи с атрибутом %ROWTYPE**
- **Создание таблицы PL/SQL INDEX BY**
- **Создание таблицы записей PL/SQL INDEX BY**
- **Различия между записями, таблицами и таблицами записей**
- **Расширения оператора GROUP BY**

Составные типы данных

Два типа данных:

- PL/SQL RECORDs (записи)
- PL/SQL коллекции
  - Nested table collections (Вложенные таблицы),
  - varray collections
  - Associative arrays,
  - string indexed collections

К составным типам данных, называемым также коллекциями (collections), относятся RECORD, TABLE, Nested TABLE и VARRAY. Тип данных RECORD ("запись") используется для обработки взаимосвязанных, но разных данных как одной логической единицы. Тип данных TABLE ("таблица'") используется для ссылок на коллекции данных как на единый объект.

Запись - это группа взаимосвязанных элементов данных, которые хранятся в полях; каждый из этих элементов данных имеет собственное имя и тип данных. Таблица имеет столбец и главный ключ, что позволяет вам обращаться к строкам как к массивам. После того, как таблицы или записи определены, они могут использоваться повторно.

```
DECLARE
 m_COMPANY VARCHAR2(30);
 m_CUST_REP INTEGER;
 m_CREDIT_LIMIT NUMBER;
```

Но иногда удобнее использовать, так называемый составной тип. Одним из таких типов, в языке **PL/SQL** является запись **RECORD**. Синтаксис объявления записи:

```
 TYPE тип записи IS RECORD   (
поле1 тип1 [NOT NULL] [:= выражение1]
поле2 тип2 [NOT NULL] [:= выражение2]
……
…..
поле-n тип-n [NOT NULL] [:= выражение-n] );
```

Можно указывать ограничение **NOT NULL**, переменной вне записи исходное значение и ограничение **NOT NULL** не обязательны!

Для создания записи необходимо:

1. Объявить тип данных RECORD;
2. Объявить переменную этого типа данных.

```
TYPE type_name IS RECORD
     (field_declaration[, field_declaration]…);
identifier  type_name;
```

Где *field_declaration*:

```
field_name {field_type | variable%TYPE
           | table.column%TYPE | table%ROWTYPE}
           [[NOT NULL] {:= | DEFAULT} expr]
```

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2  TYPE is_SmplRec IS RECORD
  3     (
  4     m_Fld1 VARCHAR2(10),
  5     m_Fld2 VARCHAR2(30)  := 'Buber',
  6     m_DtFld DATE,
  7     m_Fld3 INTEGER := 1000,
  8     m_Fld4 VARCHAR2(100) NOT NULL := 'System'
  9     );
 10  MY_SMPL is_SmplRec;
 11  BEGIN
 12  DBMS_OUTPUT.enable;
 13  DBMS_OUTPUT.put_line(MY_SMPL.m_Fld2||' '||MY_SMPL.m_Fld4);
 14  END;
 15  /
Buber System
```

Как видно процедура успешно выполнилась и вернула значения. Здесь хорошо видно, как мы проинициализировали переменные записи внутри объявления. Так же одно из полей объявленное как **NOT NULL** сразу получило значение.

```
SET SERVEROUTPUT ON
DECLARE
TYPE is_SmplRecOne IS RECORD
   ( m_Fld1 VARCHAR2(10),
   m_Fld2 VARCHAR2(30),
   m_DtFld DATE,
   m_Fld3 INTEGER,
   m_Fld4 VARCHAR2(100));
TYPE is_SmplRecTwo IS RECORD
   ( m_Fld1 VARCHAR2(10),
   m_Fld2 VARCHAR2(30),
   m_DtFld DATE,
   m_Fld3 INTEGER,
   m_Fld4 VARCHAR2(100));
 MY_SMPLONE is_SmplRecOne;
MY_SMPLTWO is_SmplRecTwo;
BEGIN
MY_SMPLONE.m_Fld3 := 100; MY_SMPLONE.m_Fld4 := 'Buber';
MY_SMPLTWO.m_Fld3 := MY_SMPLONE.m_Fld3;
MY_SMPLTWO.m_Fld4 := MY_SMPLONE.m_Fld4;
DBMS_OUTPUT.enable;
DBMS_OUTPUT.put_line(TO_CHAR(MY_SMPLTWO.m_Fld3));
DBMS_OUTPUT.put_line(MY_SMPLTWO.m_Fld4);
END;
```

```
DECLARE
TYPE is_Customers IS RECORD
 (
 m_COMPANY CUSTOMERS.COMPANY%TYPE,
 m_CUST_REP CUSTOMERS.CUST_REP%TYPE,
 m_CREDIT_LIMIT CUSTOMERS.CREDIT_LIMIT%TYPE
 );
MY_CUST is_Customers;
BEGIN
SELECT COMPANY, CUST_REP, CREDIT_LIMIT INTO MY_CUST
FROM CUSTOMERS WHERE CUST_NUM = 2108;
DBMS_OUTPUT.enable;
DBMS_OUTPUT.put_line(MY_CUST.m_COMPANY||' '||
TO_CHAR(MY_CUST.m_CUST_REP)||' ‘
|| TO_CHAR(MY_CUST.m_CREDIT_LIMIT));
END;
```

Здесь объявлена запись на основе типов таблицы **CUSTOMERS**, и выбрана одна запись из этой таблицы с помощью, оператора **INTO**, все это отправлено в переменные записи

Атрибут %ROWTYPE

- Используется для объявления переменной **типа "запись«** на основе совокупности столбцов в таблице или представлении базы данных.
- Перед %ROWTYPE указывается имя таблицы.
- Поля записи наследуют имена и типы данных от столбцов таблицы или представления. Последовательность полей также наследуется из структуры таблицы (по умолчанию).
- Такой тип значительно облегчает программирование операций со строками, позволяя выполнять выборку строки целиком в одну переменную типа "запись", а также предотвращает необходимость перепрограммирования блоков в случае изменения структуры таблицы.

Составной тип **TABLE**! В своей сути это одномерный массив скалярного типа. Он не может содержать тип **RECORD** или другой тип **TABLE**. Но может быть объявлен от любого другого стандартного типа. **TABLE** может быть объявлен от атрибута **%ROWTYPE**! Вот где скрывается, по истине огромная мощь, этого типа данных! Тип **TABLE**, можно представить как одну из разновидностей коллекции. Хотя в более глубоком понимании это не совсем так. При первом рассмотрении он похож на массив языка **C**. Массив **TABLE** индексируется типом **BINARY_INTEGER** и может содержать, что-то вроде - 2,147,483,647 - 0 - + 2,147,483,647 как в положительную, так и в отрицательную часть. Начинать индексацию можно с 0 или 1 или любого другого числа. Если массив будет иметь разрывы, то это не окажет какой-либо дополнительной нагрузки на память. Так же следует помнить, что для каждого элемента, например, типа **VARCHAR2** будет резервироваться столько памяти, сколько вы укажете, по этому определяйте размерность элементов по необходимости. Скажем не стоит объявлять **VARCHAR2(255)**, если хотите хранить строки менее 100 символов!

```
SET SERVEROUTPUT ON
DECLARE
  TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
  INDEX BY BINARY_INTEGER;
  TYPE m_SmplTblData IS TABLE OF DATE
  INDEX BY BINARY_INTEGER;
  MY_TBL m_SmplTable;
  MY_TBL_DT m_SmplTblData;
BEGIN
MY_TBL(1) := 'Buber';
MY_TBL_DT(2) := SYSDATE - 1;
DBMS_OUTPUT.enable;
DBMS_OUTPUT.put_line(TO_CHAR(MY_TBL(1)));
DBMS_OUTPUT.put_line(TO_CHAR(MY_TBL_DT(2)));
END;
```

Мы объявили две одномерные коллекции **m_SmplTable**, **m_SmplTblData**, одна из них содержит элементы размерностью **VARCHAR2(128)**, другая **DATE**. Затем объявили две переменные данного типа - **MY_TBL**, **MY_TBL_DT** и присвоили первому элементу строкового массива значение "Buber", а второму элементу массива дат, значение текущей даты минус 1. Результат вывели на консоль. При этом хорошо видно, что тип **TABLE** очень схож с таблицей БД и содержит обычно два ключа **KEY** и **VALUE**, ключ и значение. Ключ имеет тип **BINARY_INTEGER**:

Преимущества использования %ROWTYPE

- Количество базовых столбцов базы данных и типы данных в них могут быть неизвестны.
- Количество и типы базовых столбцов могут меняться во время выполнения.
- Полезен при выборке строки с помощью команды SELECT *.

Атрибут %ROWTYPE

В первом объявлении на слайде создается запись с такими же именами полей и типами данных, как и у строки таблицы DEPARTMENTS. Поля называются DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID и LOCATION_ID. Во втором объявлении создается запись с такими же именами полей и типами данных полей, как у строки таблицы EMPLOYEES. Поля называются EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, PHONE_NUMBER, HIRE_DATE, JOB_ID, SALARY, COMMISSION_PCT, MANAGER_ID, DEPARTMENT_ID.

В следующем примере, сотрудник выходит на пенсию. Информация о вышедших на пенсию сотрудниках добавляется в таблицу, которая хранит эту. Пользователю известен номер сотрудника. Запись о сотруднике, заданная пользователем выбирается из EMPLOYEES и хранится в переменной emp_rec, которая объявлена с помощью атрибута %ROWTYPE.

```
DEFINE employee_number = 124
DECLARE
        emp_rec     employees%ROWTYPE;
BEGIN
        SELECT * INTO emp_rec FROM employees
        WHERE employee_id = &employee_number;
        INSERT INTO retired_emps(empno, ename, job, mgr,     hiredate,leavedate, sal, comm,
deptno)
        VALUES (emp_rec.employee_id, emp_rec.last_name, emp_rec.job_id,emp_rec.manager_id,
emp_rec.hire_date, SYSDATE, emp_rec.salary,emp_rec.commission_pct, emp_rec.department_id);
        COMMIT;
END;
```

```
DECLARE
TYPE is_Customers IS TABLE OF CUSTOMERS%ROWTYPE
 INDEX BY BINARY_INTEGER;
MY_CUST is_Customers;
BEGIN
SELECT * INTO MY_CUST(100)
FROM CUSTOMERS WHERE CUST_NUM = 2108;
DBMS_OUTPUT.enable;
DBMS_OUTPUT.put_line(TO_CHAR(MY_CUST(100).CUST_NUM)||'
'||MY_CUST(100).COMPANY||' '||
TO_CHAR(MY_CUST(100).CUST_REP)||'
'||TO_CHAR(MY_CUST(100).CREDIT_LIMIT));
END;
```

Здесь мы объявили коллекцию, с типом запись из таблицы **CUSTOMERS**! Итак, объявляем тип:

```
TYPE is_Customers IS TABLE OF CUSTOMERS%ROWTYPE
INDEX BY BINARY_INTEGER;
```

Это значит, что каждая строка коллекции содержит полную запись из таблицы БД **CUSTOMERS**. Далее 100 элементу массива присваиваем значение записи таблицы с индексом 2108.

The %ROWTYPE Attribute

```
   ...
DEFINE employee_number = 124
 DECLARE
  emp_rec   employees%ROWTYPE;
 BEGIN
  SELECT * INTO emp_rec FROM employees
  WHERE  employee_id = &employee_number;
  INSERT INTO retired_emps(empno, ename, job, mgr,
  hiredate, leavedate, sal, comm, deptno)
  VALUES (emp_rec.employee_id, emp_rec.last_name,
  emp_rec.job_id,emp_rec.manager_id,
  emp_rec.hire_date, SYSDATE, emp_rec.salary,
  emp_rec.commission_pct, emp_rec.department_id);
END;
```

Если служащий удаляется, информация о служащем добавлена к таблице, которая содержит информацию об уволенных служащих *retired_emps*. Пользователь поставляет код (id) служащего. Запись о служащем, определенным пользователем с id=124, elfktyf из таблицы служащих и сохранена в emp_rec переменную, которая объявлена, используя аттрибут %ROWTYPE. Запись, которая вставлена в таблицу *retired_emps*

```
  ...
DEFINE employee_number = 124
  DECLARE
     emp_rec  retired_emps%ROWTYPE;
  BEGIN
   SELECT employee_id, last_name, job_id,
manager_id, hire_date, hire_date, salary,
commission_pct,department_id INTO emp_rec
FROM employees
   WHERE  employee_id = &employee_number;
   INSERT INTO retired_emps VALUES emp_rec;
  END;
/
  SELECT * FROM retired_emps;
```

```
SET SERVEROUTPUT ON
SET VERIFY OFF
DEFINE employee_number = 124
DECLARE
   emp_rec retired_emps%ROWTYPE;
BEGIN
 SELECT * INTO emp_rec FROM retired_emps;
 emp_rec.leavedate:=SYSDATE;
 UPDATE retired_emps SET ROW = emp_rec
WHERE empno=&employee_number;
END;
/
SELECT * FROM retired_emps;
```

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
 TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
 INDEX BY BINARY_INTEGER;
 MY_TBL m_SmplTable;
        BEGIN
        FOR i IN 1..10 LOOP
        MY_TBL(i) := TO_CHAR(i+5);
        END LOOP;
DBMS_OUTPUT.enable;
DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
END;
```

С помощью цикла **FOR** мы ввели в таблицу 10 значений и применив атрибут - **COUNT** получили количество записей в таблице. Следующий атрибут **DELETE**. Тут чуть сложнее. Во-первых, он имеет аргументы:

DELETE - удаляет все строки таблицы.

DELETE(n) - удаляет **n**-ю строку коллекции.

DELETE(n,m) - удаляет строки коллекции с **n** по **m**.

```
SET SERVEROUTPUT ON
-- DELETE
DECLARE
  TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
  INDEX BY BINARY_INTEGER;
  MY_TBL m_SmplTable;
BEGIN
MY_TBL(1) := 'One'; MY_TBL(3) := 'Three';
MY_TBL(-2) := 'Minus Two'; MY_TBL(0) := 'Zero';
MY_TBL(100) := 'Hundred';
DBMS_OUTPUT.enable;
DBMS_OUTPUT.put_line(MY_TBL(1)); DBMS_OUTPUT.put_line(MY_TBL(3));
DBMS_OUTPUT.put_line(MY_TBL(-2)); DBMS_OUTPUT.put_line(MY_TBL(0));
DBMS_OUTPUT.put_line(MY_TBL(100));
DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
MY_TBL.DELETE(100);
DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
MY_TBL.DELETE(1,3);
DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
MY_TBL.DELETE;
DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
END;
```

В данном случае сначала мы удалили 100-ю запись коллекции, затем с 1 по 3 и затем очистили всю коллекцию.

- SET SERVEROUTPUT ON
  -- *EXISTS*
  DECLARE
    TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
    INDEX BY BINARY_INTEGER;
    MY_TBL m_SmplTable;
  BEGIN
  MY_TBL(1) := 'Miller';
  MY_TBL(3) := 'Kolobok';
  DBMS_OUTPUT.enable;
  DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
  IF (MY_TBL.EXISTS(1)) THEN
  DBMS_OUTPUT.put_line(MY_TBL(1));
  ELSE DBMS_OUTPUT.put_line('MY_TBL(1) is not exist!');
  END IF;
  IF (MY_TBL.EXISTS(3)) THEN
  DBMS_OUTPUT.put_line(MY_TBL(3));
  ELSE DBMS_OUTPUT.put_line('MY_TBL(3) is not exist!');
  END IF;
  IF (MY_TBL.EXISTS(2)) THEN
  DBMS_OUTPUT.put_line(MY_TBL(2));
  ELSE DBMS_OUTPUT.put_line('MY_TBL(2) is not exist!');
  END IF; END;
  Count table is: 2
  Miller
  Kolobok
  MY_TBL(2) is not exist!

Объявленной коллекции существуют только две строки с номером 1 и 3! А строка с номером 2 не существует.

Расширения оператора GROUP BY

Операторы ROLLUP, CUBE и GROUPING SETS являются расширениями предложения GROUP BY. Операторы ROLLUP, CUBE и GROUPING SETS могут сформировать такой же результирующий набор, который получится в результате использования оператора UNION ALL для объединения одиночных запросов группирования, однако использование одного из операторов предложения GROUP BY обычно является более эффективным.

Группировка ROLLUP приводит к созданию **промежуточных итогов** по столбцу назначения. Промежуточные итоги вычисляются от самого низкого уровня до общей суммы. То есть, включая ROLLUP во фразе GROUP BY, мы указываем Oracle, чтобы он просуммировал данные по уровнях указанных столбцов и подвел общий итог.

```
SELECT    [column,] group_function(column). . .
FROM      table
[WHERE    condition]
[GROUP BY    [ROLLUP] group_by_expression]
[HAVING    having_expression];
    [ORDER BY    column];
```

Пример. Запрос просуммировал данные по зарплатам на уровнях по отделам, должностям и подвел общий итог

```
SELECT   department_id, job_id, SUM(salary)
FROM     employees
WHERE    department_id < 60
GROUP BY ROLLUP(department_id, job_id);
```

DEPARTMENT_ID	JOB_ID	SUM(SALARY)	
10	AD_ASST	4400	1
10		4400	
20	MK_MAN	13000	
20	MK_REP	6000	
20		19000	
30	PU_MAN	11000	
30	PU_CLERK	13900	
30		24900	
40	HR_REP	6500	
40		6500	2
50	ST_MAN	36400	
50	SH_CLERK	64300	
50	ST_CLERK	55700	
50		156400	
		211200	3

15 rows selected.

Функция CUBE

Оператор CUBE формирует результирующий набор, представляющий собой многомерный куб. То есть, результирующий **набор содержит все возможные комбинации значений указанных столбцов вместе со значениями статистических вычислений** соответствующих строк, которые совпадают с комбинацией значений указанных столбцов.

Если запрос с CUBE возвращает больше значений, чем вам нужно, то лишнее можно спрятать в представление или вложенный запрос.

В дополнение к групповым под итогам и общим итогам, созданным ROLLUP, CUBE автоматически вычисляет все возможные комбинации возможных под итогов. Это предоставляет агрегированную, просуммированную информацию для каждой категории.

```
SELECT [column,] group_function(column)...
FROM table [WHERE condition]
[GROUP BY [CUBE] group_by_expression]
[HAVING having_expression]
[ORDER BY column];
```

Пример ниже показывает запрос, ранее уже использованный, но с **функцией CUBE** для получения дополнительной агрегированной информации:

```
SELECT   department_id, job_id, SUM(salary)
FROM     employees
WHERE    department_id < 60
GROUP BY CUBE (department_id, job_id) ;
```

DEPARTMENT_ID	JOB_ID	SUM(SALARY)	
		211200	1
	HR_REP	6500	
	MK_MAN	13000	
	MK_REP	6000	
	PU_MAN	11000	
	ST_MAN	36400	2
	AD_ASST	4400	
	PU_CLERK	13900	
	SH_CLERK	64300	
	ST_CLERK	55700	
10		4400	3
10	AD_ASST	4400	
20		19000	
20	MK_MAN	13000	
20	MK_REP	6000	4
30		24900	
30	PU_MAN	11000	

15 rows selected.

Различия между CUBE и ROLLUP: CUBE создает результирующий набор, содержащий статистические выражения для всех комбинаций значений заданных столбцов. ROLLUP создает результирующий набор, содержащий статистические выражения иерархии значений в заданных столбцах.

Функция GROUPING

Для того, чтобы **однозначно определить промежуточные итоги** и чтобы улучшить обработку NULL-значений в строках, созданных ROLLUP и CUBE, корпорация Oracle представила новую функцию GROUPING, которая возвращает значение 1, если строка - это под итог, созданная ROLLUP или CUBE, и 0 в противном случае. Пример в следующим слайде показывает запрос, уже использованный ранее, с функциями DECODE и GROUPING, применение которых позволяет более элегантно обрабатывать null-значения, созданные ROLLUP и CUBE.

```
SELECT    [column,] group_function(column) .. ,
          GROUPING(expr)
FROM      table [WHERE   condition]
[GROUP BY [ROLLUP][CUBE] group_by_expression]
[HAVING   having_expression]
[ORDER BY column];
```

Пример:

```
SELECT    department_id DEPTID, job_id JOB,
          SUM(salary),
          GROUPING(department_id) GRP_DEPT,
          GROUPING(job_id) GRP_JOB
FROM      employees
WHERE     department_id < 50
GROUP BY ROLLUP(department_id, job_id);
```

DEPTID	JOB	SUM(SALARY)	GRP_DEPT	GRP_JOB
10	AD_ASST	4400	0	0
10		4400	0	1
20	MK_MAN	13000	0	0
20	MK_REP	6000	0	0
20		19000	0	1
30	PU_MAN	11000	0	0
30	PU_CLERK	13900	0	0
30		24900	0	1
40	HR_REP	6500	0	0
40		6500	0	1
		54800	1	1

11 rows selected.

1

2

3

Функция GROUPING SETS

GROUPING SETS является дальнейшее расширение предложения GROUP BY, которые можно использовать для указания нескольких групп данных. Это способствует эффективной агрегации и, следовательно, облегчает анализ данных по нескольким измерениям. Простой оператор SELECT теперь может быть написан с использованием GROUPING SETS для задания различных группировок (которые также могут включать ROLLUP или CUBE операторы), а не множественного выбора заявления объединены UNION ALL операторов.

```
SELECT department_id, job_id, manager_id, AVG(salary)
  FROM employees
   GROUP BY
    GROUPING SETS
    ((department_id, job_id, manager_id),
        (department_id, manager_id),(job_id, manager_id));
```

Сравните предыдущий пример со следующей альтернативой:

```
SELECT department_id, job_id, manager_id, AVG(salary)
     FROM employees
     GROUP BY CUBE(department_id, job_id, manager_id);
```

Следующий запрос вычисляет все 8 (2 * 2 * 2) группировки, но только по группам:

```
SELECT department_id, job_id, manager_id, AVG(salary)
  FROM employees
    GROUP BY department_id, job_id, manager_id
UNION ALL
SELECT department_id, NULL, manager_id, AVG(salary)
  FROM employees
    GROUP BY department_id, manager_id
UNION ALL
SELECT NULL, job_id, manager_id, AVG(salary)
   FROM employees
     GROUP BY job_id, manager_id;
```

Выполните все примеры и сравните.

Объединение группировок

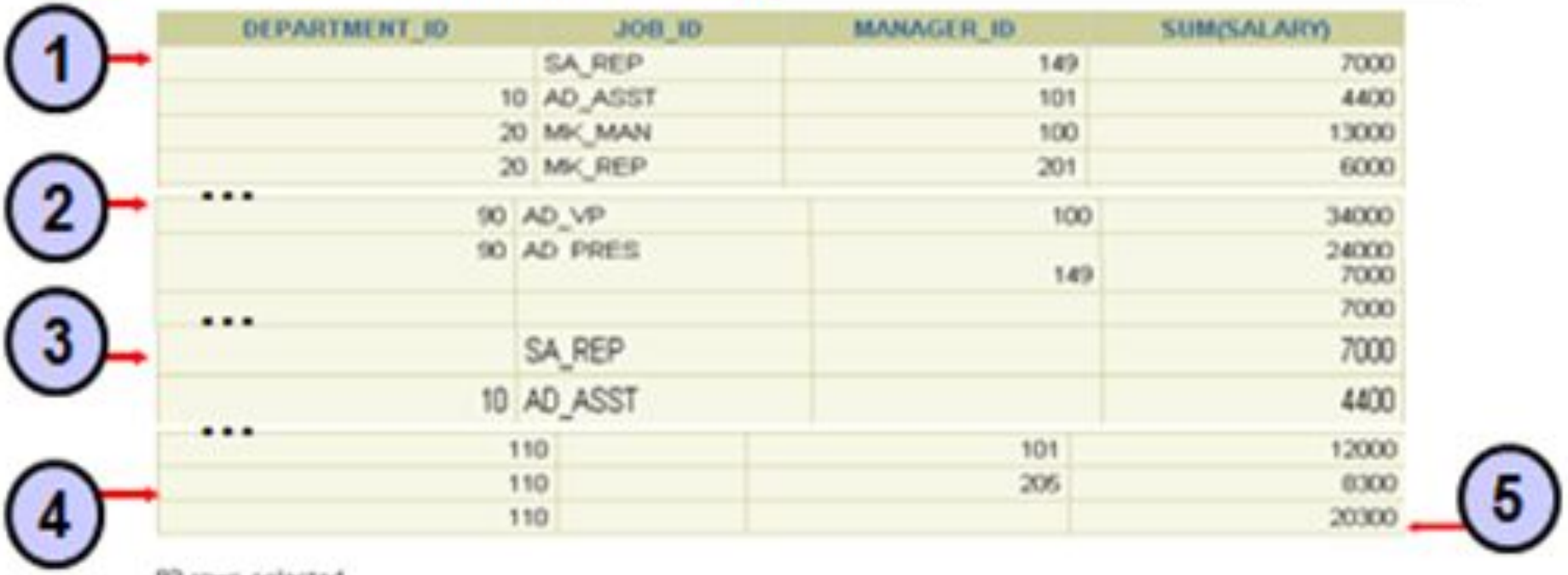
Объединение групп - краткий путь для создания полезной комбинации группировок. Каскадной группировки указано, перечислив несколько наборов группировки, кубы и свертки и разделяя их запятой.

Пример. Рассмотрим пример объединенной группировки:

- (job_id, manager_id) (1)
- (department_id,job_id, manager_id) (2)
- (job_id)(3)
- (department_id,manager_id)(4)
- (department_id) (5)

Рассчитывается общая зарплата для каждой из этих групп. Запрос и результат представлены на следующем слайде

```
SELECT    department_id, job_id, manager_id,
          SUM(salary)
FROM      employees
GROUP BY  department_id,
          ROLLUP(job_id),
          CUBE(manager_id);
```



DEPARTMENT_ID	JOB_ID	MANAGER_ID	SUM(SALARY)
	SA_REP	149	7000
10	AD_ASST	101	4400
20	MK_MAN	100	13000
20	MK_REP	201	6000
...			
90	AD_VP	100	34000
90	AD_PRES		24000
		149	7000
...			7000
	SA_REP		7000
10	AD_ASST		4400
...			
110		101	12000
110		205	8300
110			20300

93 rows selected

Интерактивный запрос значений переменных в командах Oracle

При рассмотрении команды INSERT мы использовали переменную подстановки. Достаточно часто при выполнении отчетов пользователи задают критерии выборки данных динамически. Возможны следующие варианты использования переменных подстановки SQL*Plus для временного хранения значений:

- одиночный амперсанд (&);
- двойной амперсанд (&&);
- команда DEFINE.

В SQL*Plus можно использовать переменные подстановки с одним амперсандом (&) для временного хранения значений.

```
SELECTemployee_id, last_name, salary, department_id
FROM  employees
WHERE    employee_id = &employee_num ;
```

Переменные в SQL*Plus можно предопределить с помощью команд ACCEPT или DEFINE. Команда ACCEPT считывает строку пользовательских вводных данных и запоминает их в переменной. Команда DEFINE создает переменную и присваивает ей значение. Переменные подстановки могут замещать:

- условие WHERE;
- предложение ORDER BY;
- выражение столбца;
- имя таблицы.

В предложении WHERE даты и символьные значения должны быть заключены в апострофы. Это правило относится и к переменным подстановки. Амперсанд можно использовать также с функциями типа UPPER и LOWER. Чтобы пользователю не нужно было вводить должность заглавными буквами, используйте функцию UPPER (‘&job_title’).

```
SELECT employee_id, last_name, job_id, &column_name
FROM        employees
WHERE  &condition
ORDER BY    &order_column ;
```

Пользовательские переменные можно определять до выполнения команды SELECT. Для определения и установки значений пользовательских переменных SQL*Plus предлагает команду DEFINE. Если в команде DEFINE требуется одиночный пробел, этот пробел должен быть заключен в апострофы.

DEFINE переменная = значение (создает пользовательскую переменную с типом данных CHAR).

```
DEFINE job_title = IT_PROG
DEFINE job_title
DEFINE JOB_TITLE = "IT_PROG" (CHAR)
```

Если для удаления переменных используется команда UNDEFINE, проверить изменения можно командой DEFINE. При выходе из SQL*Plus все переменные, заданные во время сеанса, теряются.

Использование двойного знака «&&» для переменной подстановки. Для многократного использования значения переменной без повторных приглашений пользователю ввести значение можно использовать переменную подстановки с двойным амперсандом (&&). Пользователь получает приглашение ввести значение только один раз.

```
SELECT   employee_id, last_name, job_id, &&column_name
FROM     employees
ORDER BY &column_name;
```

Значение введенной после сдвоенного знака «&» переменной подстановки («table_name» в данном случае) запоминается в SQL*Plus, и в следующий раз при обращении к этой переменной подстановки ее значение вводится автоматически:

```
SELECT * FROM &&table_name;
```

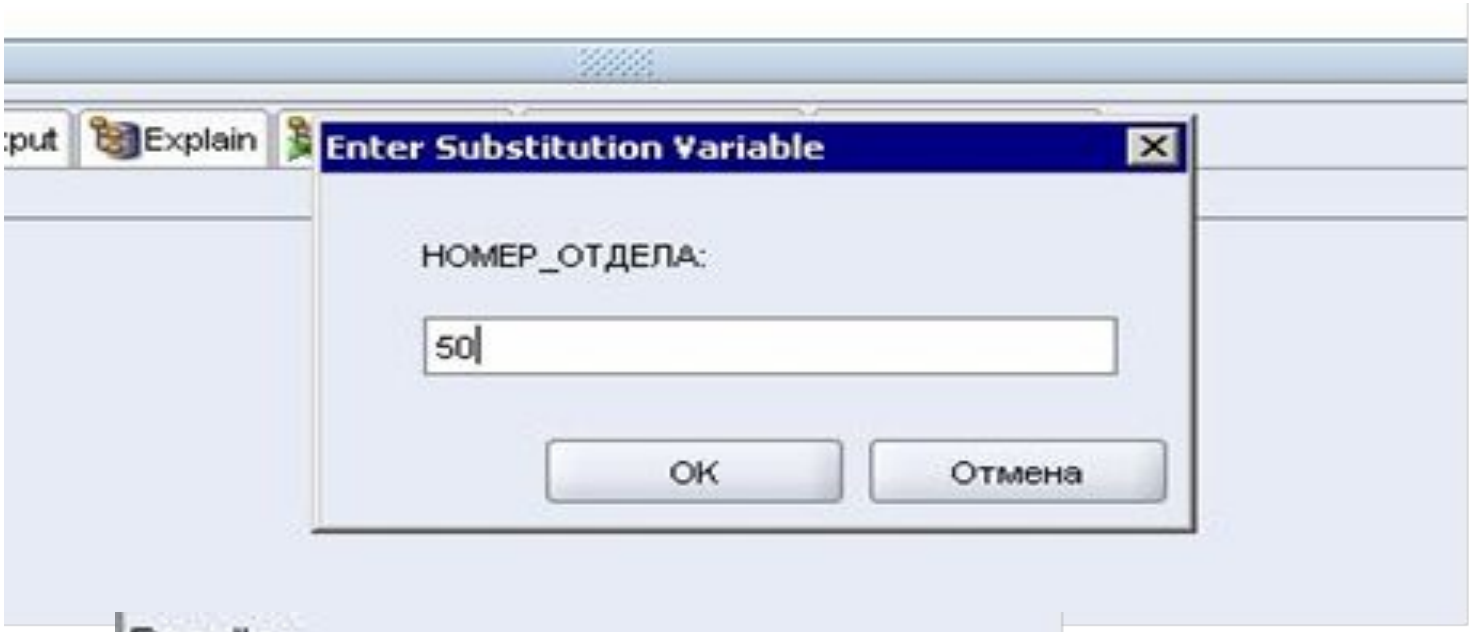
Пример: Напишите запрос, который вы выводил информацию об имени, фамилии и заработной плате сотрудников из таблицы hr.employees. При этом запрос:

- должен запрашивать у пользователя информацию о номере отдела и выводить информацию только о пользователях соответствующего отдела;
- должен показывать пользователю не только результаты запроса, но и текст запроса со значением, введенным пользователем;

Примечание: Если вы используете Oracle SQL Developer, то для вывода результатов запроса в текстовом виде нужно использовать клавишу F5.

Код соответствующих команд может выглядеть так:

```
SET VERIFY ON;
SELECT first_name As "Имя", last_name As "Фамилия",
      salary As "Оклад"
 FROM employees
WHERE department_id = &Номер_Отдела;
```



Results:

	Имя	Фамилия	Оклад
1	Donald	OConnell	2600
2	Douglas	Grant	2600
3	Matthew	Weiss	8000
4	Adam	Fripp	8200
5	Payam	Kaufling	7900
6	Shanta	Vollman	6500
7	Kevin	Mourgos	5800

Конструкции MERGE и WITH

Оператор MERGE

MERGE — DML-оператор вставки (INSERT)/обновления (UPDATE)/удаления (DELETE, начиная с Oracle Database 10g) данных при слиянии таблиц.

В сервере Oracle Database 10g будут срабатывать триггеры BEFORE UPDATE, INSERT и/или DELETE – в зависимости от указанных в операторе MERGE операций, так как в этом операторе предложения **WHEN MATCHED THEN UPDATE** (когда совпадают, то обновить) и **WHEN NOT MATCHED THEN INSERT** (когда не совпадают, то вставить) являются обязательными.

Оператор MERGE иногда называют еще UPSERT = UPdate + inSERT, так как он как бы объединяет в себе два этих оператора. Данный оператор позволяет обновить основную таблицу с помощью вспомогательной таблицы. При совпадении ключевых полей в строках обеих таблиц строки основной таблицы обновляются данными из совпавших строк вспомогательное таблицы; все не совпавшие строки из вспомогательной таблицы добавляются в основную таблицу.

Пример. Следующий пример использует таблицу **BONUSES** в примере схемы OE со значением бонуса по умолчанию в размере 100. Затем он вставляет в таблицу **BONUSES** всех сотрудников, объем продаж которых, основан на столбце SALES_REP_ID таблицы OE.ORDERS. Наконец, менеджер по персоналу решает, что работники с зарплатой в $ 8000 или менее должны получать бонус. Те же, кто не сделал продаж, получают бонус в размере 1% от их зарплаты. Те, кто уже сделал продажи, получают увеличение их бонуса в размере 1% от их заработной платы. Заявление MERGE реализует эти изменения в один шаг:

Предварительные действия для получения доступа к схеме OE:

1)

```
ALTER USER oe ACCOUNT UNLOCK;
ALTER USER oe IDENTIFIED BY OE;
GRANT SELECT ON oe.orders TO hr;)
```

2)

```
CREATE TABLE bonuses(employee_id NUMBER,
bonus NUMBER DEFAULT 100);
```

3)

```
    INSERT INTO hr.bonuses(employee_id)
     (SELECT e.employee_id  FROM hr.employees e, oe.orders o
      WHERE e.employee_id = o.sales_rep_id
  GROUP BY e.employee_id);
```

4)

```
SELECT *
 FROM bonuses
 ORDER BY employee_id;
```

EMPLOYEE_ID	BONUS
153	100
154	100
155	100
156	100
158	100
159	100
160	100
161	100

5)

```
MERGE INTO bonuses1 D
  USING (SELECT employee_id, salary, department_id
        FROM employees   WHERE department_id = 80) S
  ON (D.employee_id = S.employee_id)
  WHEN MATCHED THEN
    UPDATE
      SET D.bonus = D.bonus +    S.salary*.01
    DELETE
      WHERE (S.salary > 8000)
  WHEN NOT MATCHED THEN
    INSERT (D.employee_id, D.bonus)
    VALUES (S.employee_id, S.salary*.01)
      WHERE (S.salary <= 8000);
```

6)

```
SELECT *
 FROM bonuses
 ORDER BY employee_id;
```

EMPLOYEE_ID	BONUS
153	180
154	175
155	170
159	180
160	175
161	170
164	72
165	68
166	64
167	62
171	74
172	73
173	61
179	62

Пример : Слияние строк - пример показывает соответствия EMPLOYEE_ID в таблице COPY_EMP к EMPLOYEE_ID в таблице EMPLOYEES. Если соответствие найдено, строка таблицы COPY_EMP модифицируется в соответствии со строкой таблицы EMPLOYEES. Если строка не найдена, она вставляется в таблицу COPY_EMP.

```
MERGE INTO copy_emp AS c
   USING employees e
   ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
   UPDATE SET
      c.first_name = e.first_name,
      c.last_name = e.last_name,
      c.email = e.email,
      c.phone_number = e.phone_number,
      c.hire_date = e.hire_date,
      c.job_id = e.job_id,
      c.salary = e.salary,
      c.commission_pct = e.commission_pct,
      c.manager_id = e.manager_id,
      c.department_id = e.department_id
WHEN NOT MATCHED THEN
   INSERT VALUES(e.employee_id, e.first_name, e.last_name,
      e.email, e.phone_number, e.hire_date, e.job_id,
      e.salary, e.commission_pct, e.manager_id,
      e.department_id);
```

Использование конструкции WITH

Oracle допускает вынесение определений подзапросов из тела основного запроса с помощью особой фразы **WITH. WITH** (формально известный как subquery_factoring_clause) позволяет многократно использовать один и тот же блок запроса в инструкции SELECT, когда она встречается более одного раза в сложном запросе.

Фраза WITH используется в двух целях:

- для придания запросу формулировки, более понятной программисту (просто subquery factoring) и
- для записи рекурсивных запросов (recursive subquery factoring).

Используя оператор **WITH**, можно определить блок запроса, прежде чем использовать его в запросе. Это особенно удобно, если запрос имеет несколько ссылок на один и тот же блок запросов и в нем есть объединения и агрегатные функции.

Использование **WITH**, сервер Oracle получает результаты запроса блока и сохраняет его во временном табличном пользователя. Это может улучшить производительность.

Фраза WITH предшествует фразе SELECT и позволяет привести сразу несколько предварительных формулировок подзапросов для ссылки на них в ниже формулируемом основном запросе. Общая схема употребления демонстрируется следующей схемой:

```
WITH
  A AS ( SELECT ... )
, B AS ( SELECT ... FROM x )
, C AS ( SELECT ... FROM x, y )
SELECT ... FROM A, B, C
;
```

Пример. Необходимо выдать список департаментов, имеющих фонд заработной платы больший, чем 1/8 фонда заработной платы всего предприятия.

```
WITH
summary AS (
  SELECT d.department_name AS department,
        SUM(e.salary) AS dept_total
  FROM employees e, departments d
  WHERE e.department_id = d.department_id
  GROUP BY d.department_name)
SELECT department, dept_total
FROM summary
WHERE dept_total > (
                SELECT SUM(dept_total) * 1/8
                FROM summary  )
ORDER BY dept_total DESC;
```

Пример . Запрос создает запросы **DEPT_COSTS** и **AVG_COST,** а затем использует их в теле основного запроса.

```
WITH
dept_costs  AS (
 SELECT d.department_name, SUM(e.salary) AS dept_total
  FROM   employees e JOIN departments d
  ON     e.department_id = d.department_id   GROUP BY d.department_name),
avg_cost    AS ( SELECT SUM(dept_total)/COUNT(*) AS dept_avg FROM   dept_costs)
SELECT * FROM   dept_costs WHERE  dept_total >(SELECT dept_avg FROM avg_cost)
ORDER BY department_name;
```

Пример . Выдать фамилию сотрудников, у которых зарплата больше средней по компании, их зарплату и департамент (два метода):

```
SELECT e1.Last_name,e1.salary,e1.department_id, e2.average
FROM employees e1,
(SELECT ROUND(AVG(salary)) as average
   FROM employees  ) e2 WHERE e1.salary>e2.average ORDER BY salary;
```

-----с оператором----------WITH----------

```
WITH t1 AS
(SELECT last_name,salary FROM employees),
t2 AS (SELECT ROUND(AVG(salary)) AS average FROM employees)
seleCT last_name,salary,average
FROM t1,t2 WHERE salary >  average ORDER BY salary;
```

Формулирование рекурсивных запросов

С версии Oracle 11.2 фраза WITH может использоваться для формулирования рекурсивных запросов, в соответствии (неполном) со стандартом SQL:1999. В этом качестве она способна решать ту же задачу, что и CONNECT BY, однако (а) делает это похожим с СУБД других типов образом, (б) обладает более широкими возможностями, (в) применима не только к запросам по иерархии и (г) записывается значительно более замысловато.

Общий алгоритм вычисления фразой WITH таков:

Результат := пусто;
Добавок := исходный SELECT ...;
Пока Добавок не пуст выполнять:
  Результат := Результат
{UNION ALL | UNION | INTERSECT | EXCEPT}
Добавок;
  Добавок := рекурсивный SELECT ... FROM Добавок …;
конец цикла;

Предложение SELECT для исходного множества строк Oracle называет опорным (anchor) членом фразы WITH. Предложение SELECT для получения добавочного множества строк Oracle называют рекурсивным членом. Обратите внимание, что для вычитания множеств строк Oracle использует здесь не собственное обозначение MINUS, а стандартное EXCEPT.

Приведем 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
;
```

Операция UNION ALL здесь используется символически, в рамках определенного контекста, для указания способа рекурсивного накопления результата.

Результат:

```
         N
----------
         1
         2
         3
         4
         5
```

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 не обязательно только к иерархически организованным данным.

Пример с дополнительным разъяснением способа выполнения:

```
SQL> WITH
  2   anchor1234 ( n ) AS (          -- обычный
  3     SELECT 1 FROM dual UNION ALL
  4     SELECT 2 FROM dual UNION ALL
  5     SELECT 3 FROM dual UNION ALL
  6     SELECT 4 FROM dual
  7   )
  8  , numbers ( n ) AS (           -- рекурсивный
  9     SELECT n FROM anchor1234
 10       UNION ALL
 11     SELECT n + 1 AS n
 12     FROM   numbers
 13     WHERE  n < 5
 14   )
 15  SELECT n FROM numbers
 16  ;
```

Разбор решения:

```
       N
----------
       1   опорный запрос
        ……..
       4   опорный запрос
       2   рекурсия 1
        ……..
       5   рекурсия 1
       3   рекурсия 2
       4   рекурсия 2
       5   рекурсия 2
       4   рекурсия 3
       5   рекурсия 3
       5   рекурсия 4
```

Пример. Вычислить какое количество сотрудников ежегодно поступает на работу с 2002 по 2010 гг.

```
WITH     period ( year ) AS ( SELECT 2002 AS year FROM dual
                              UNION ALL
                              SELECT year + 1 AS year  FROM   period WHERE  year < 2010 )
SELECT   p.year, COUNT ( e.employee_id )
FROM     employees e RIGHT OUTER JOIN period p
         ON p.year = EXTRACT ( YEAR FROM e.hire_date )
GROUP BY p.year
ORDER BY p.year;
```

Древовидные запросы

В Oracle реализованы так называемые запросы, предназначенные для работы с данными, организованными в виде дерева. Для реализации дерева в виде таблицы в ней должно быть дополнительных два поля: id узла и id родительского узла. Также должен быть корень (корни).

```
SELECT [LEVEL], column, expr...  FROM   table
[WHERE condition(s)]
[START WITH condition(s)]
    [CONNECT BY PRIOR condition(s)] ;
```

Для реализации древовидных запросов имеются два дополнительных предложения:

START WITH - для идентификации коренных строк

CONNECT BY - для связи строк-потомков и строк-предков

Необязательный оператор **START WITH** (начать с) - если задано, то позволяет указать, с какой вершины (строки) или вершин (строк) начать построение дерева. Если в заданной иерархии обнаруживается петля, то Oracle возвращает сообщение об ошибке. Условие может быть практически любым, можно даже использовать функции или внутренние запросы: pid is null, или id = 1, или даже substr(title, 1, 1) = ‘P’.

Обязательный оператор **CONNECT BY.** Условие после CONNECT BY (соединить по) - задает зависимость между родительскими и дочерними вершинами (строками) иерархии. Тут надо сказать Ораклу, как долго продолжать цикл. Что-то в духе while в обычных языках программирования. Например, мы можем попросить достать нам 10 строк: **ROWNUM** <=10 – он и создаст нам в цикле ровно 10 одинаковых строк. **ROWNUM** это "псевдостолбец", в котором нумеруются строки, начиная от 1 в порядке их выдачи. Его можно использовать не только в иерархических запросах. Но это уже другая история.

Фигурирующие в документации по Oracle "псевдостолбцы" подобны "системным переменным", но в отличие от них способны давать в запросах на разных строках разные значения, которые вычисляются по мере выполнения определенных фаз обработки запроса и доступны для использования на последующих фазах обработки, образуя как бы дополнительный "столбец".

Псевдостолбец	Тип	Описание
ROWNUM	NUMBER	Последовательный номер строки в результате SELECT
LEVEL	NUMBER	Номер уровня выдаваемой строки в предложении SELECT с использованием CONNECT BY
CONNECT_BY_ISCYCLE	NUMBER	В предложении SELECT с использованием CONNECT BY: 1, если потомок узла является одновременно его предком, иначе 0
CONNECT_BY_ISLEAF	NUMBER	В предложении SELECT с использованием CONNECT BY:1, если узел не имеет потомков
ROWID	VARCHAR2(256)	Физический адрес строки или хранимого объекта
XMLDATA	CLOB	Текст документа объекта типа XMLTYPE
OBJECT_ID	RAW (16)	Идентификатор объекта в таблице первичных или виртуальных объектов (представлений)
OBJECT_VALUE	тип объекта	Системное имя для столбца в таблице первичных или виртуальных объектов (в том числе типа XMLTYPE)
ORA_ROWSCN	NUMBER	Порядковый номер изменения в БД (SCN), соответствующий строке таблицы или же блоку данных с этой строкой

Чтобы получить нормальную иерархию нужно использовать специальный оператор, который называется **PRIOR**. Это обычный унарный оператор, точно такой же как + или -. “Позвоните родителям” – говорит он, заставляя Оракл обратиться к предыдущей записи. В примере задано направление от родителя к потомку (прочитать эту часть запроса следует так: предшестающим столбцу EMPLOYEE_id является столбец MANAGER_ID). Направлению от потомка к родителю соответствовало бы условие EMPLOYEE_id = PRIOR manager_id или PRIOR manager_id =EMPLOYEE_id. (как говорится, от перестановки мест…).

Используя информацию, заданную в этих предложениях, Oracle формирует иерархию следующим образом:

1. Выбирает корневую строку (строки) иерархии в соответствии с условиями заданными во фразе START WITH. Если эта фраза опущена, то Oracle будет использовать все строки таблицы в качестве корневых (попробуйте). Условие в START WITH может содержать подзапрос.

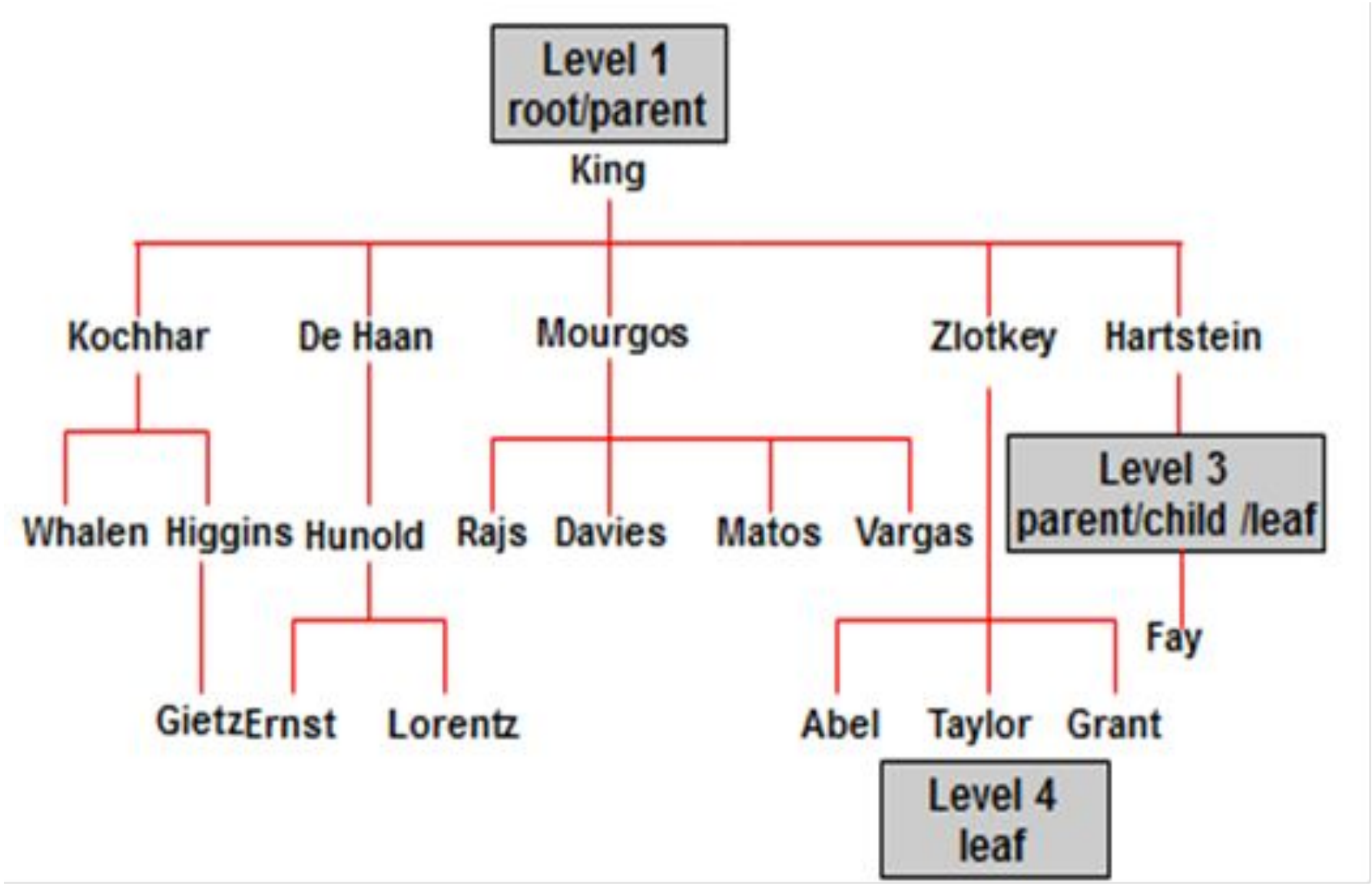
2. Для каждой корневой строки выбираются дочерние строки, каждая из которых должна удовлетворять условию фразы CONNECT BY по отношению к одной из корневых строк.

3. Oracle выбирает следующие одно за другим поколения дочерних строк. Сначала выбираются потомки строк, полученных на этапе 2, затем потомки этих потомков и так далее. Oracle всегда выбирает дочерние строки на основании условия CONNECT BY по отношению к текущей родительской строке.

4. Если запрос содержит фразу WHERE, то Oracle исключает из иерархии все строки, которые не удовлетворяют условию, заданному в этой фразе.

5. Oracle возвращает результат запроса, в котором выбранные строки расположены в иерархическом порядке, то есть потомки выводятся после своих родителей.

Порядок строк — это хорошо, но нам было бы трудно понять, две строки рядом это родитель и его потомок или два брата-потомка одного родителя. Пришлось бы сверять id и pid., Oracle предлагает в помощь дополнительный псевдостолбец **LEVEL**. Как легко догадаться, в нем записывается уровень записи по отношению к корневой. Так, 1-ая запись будет иметь уровень 1, ее потомки уровень 2, потомки потомков — 3 и т.д.


Level 1
root/parent
King
Kochhar
De Haan
Mourgos
Zlotkey
Hartstein
Whalen
Higgins
Hunold
Rajs
Davies
Matos
Vargas
Level 3
parent/child /leaf
Fay
Gietz
Ernst
Lorentz
Abel
Taylor
Grant
Level 4
leaf

Пример . Показать менеджеров и сотрудников пока **employee_id**=101.

```
SELECT employee_id, last_name, job_id, manager_id FROM employees
START  WITH  employee_id = 101
    CONNECT BY PRIOR manager_id = employee_id ;
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	MANAGER_ID
101	Kochhar	AD_VP	100
100	King	AD_PRES	

Оператор SELECT, осуществляющий древовидный запрос, может использовать псевдостолбец LEVEL, содержащий уровень вложенности для каждой строки. Для коренных записей LEVEL=1, для потомков коренных записей LEVEL=2 и и.д.

Пример .

```
START WITH employee_id = 100;
SELECT LPAD(' ',2*(LEVEL-1)) || last_name
org_chart,  department_id,  manager_id,  job_id
FROM employees
 START WITH job_id = 'AD_PRES'
CONNECT BY PRIOR employee_id = manager_id;
```

	ORG_CHART	DEPARTMENT_ID	MANAGER_ID	JOB_ID
1	King	90	(null)	AD_PRES
2	Kochhar	90	100	AD_VP
3	Greenberg	100	101	FI_MGR
4	Faviet	100	108	FI_ACCOUNT
5	Chen	100	108	FI_ACCOUNT
6	Sciarra	100	108	FI_ACCOUNT
7	Urman	100	108	FI_ACCOUNT
8	Popp	100	108	FI_ACCOUNT
9	Whalen	10	101	AD_ASST
10	Mavris	40	101	HR_REP
11	Baer	70	101	PR_REP
12	Higgins	110	101	AC_MGR
13	Gietz	110	205	AC_ACCOUNT

Используя функцию LPAD и псевдостолбец LEVEL можно украсить выводимый результат, предварительно отформатировав столбец «tree» :

```
SELECT LPAD ('  ', (LEVEL-1)*3) || LEVEL || ' ' || last_name AS tree, EMPLOYEE_ID,job_id,manager_id
 FROM EMPLOYEES
  CONNECT BY PRIOR EMPLOYEE_ID = manager_id;
```

	TREE	EMPLOYEE_ID	JOB_ID	MANAGER_ID
1	1 King	100	AD_PRES	(null)
2	2 Kochhar	101	AD_VP	100
3	3 Greenberg	108	FI_MGR	101
4	4 Faviet	109	FI_ACCOUNT	108
5	4 Chen	110	FI_ACCOUNT	108
6	4 Sciarra	111	FI_ACCOUNT	108
7	4 Urman	112	FI_ACCOUNT	108
8	4 Popp	113	FI_ACCOUNT	108
9	3 Whalen	200	AD_ASST	101
10	3 Mavris	203	HR_REP	101
11	3 Baer	204	PR_REP	101
12	3 Higgins	205	AC_MGR	101
13	4 Gietz	206	AC_ACCOUNT	205
14	2 De Haan	102	AD_VP	100
15	3 Hunold	103	IT_PROG	102
16	4 Ernst	104	IT_PROG	103
17	4 Austin	105	IT_PROG	103
18	4 Pataballa	106	IT_PROG	103
19	4 Lorentz	107	IT_PROG	103

Напишем запрос, который возвращал бы из таблицы hr.employees информацию:

- имя сотрудника;
- фамилию сотрудника;
- уровень подчиненности (самый высокий уровень – главный начальник, который никому не подчиняется — уровень 0);
- путь подотчетности в формате /руководитель1/руководитель2/ сотрудник.

```
SELECT first_name "NAME", last_name "LAST_NAM",
LEVEL - 1 As "Level podchin", SYS_CONNECT_BY_PATH
(first_name     || ' ' || last_name, '/') As "Podotchetn"
FROM employees
START WITH employee_id = 100
CONNECT BY PRIOR employee_id = manager_id
ORDER By LEVEL;
```

	Имя	Фамилия	Уровень подчиненности	Подотчетность
1	Steven	King	0	/Steven King
2	Neena	Kochhar	1	/Steven King/Neena Kochhar
3	Lex	De Haan	1	/Steven King/Lex De Haan
4	Den	Raphaely	1	/Steven King/Den Raphaely
5	Matthew	Weiss	1	/Steven King/Matthew Weiss
6	Adam	Fripp	1	/Steven King/Adam Fripp
7	Payam	Kaufling	1	/Steven King/Payam Kaufling
8	Shanta	Vollman	1	/Steven King/Shanta Vollman
9	Kevin	Mourgos	1	/Steven King/Kevin Mourgos
10	John	Russell	1	/Steven King/John Russell

Чтобы указать БД в Oracle, что сортировать надо только в пределах одного уровня иерархии, нам поможет маленькая добавка в виде оператора **SIBLINGS**. Достаточно изменить условие сортировки на **ORDER SIBLINGS BY** title – и все встанет на свои места.

```
SELECT first_name "NAME", last_name "LAST_NAM",
LEVEL - 1 As "Level podchin", SYS_CONNECT_BY_PATH (first_name  || ' ' || last_name, '/') As "Podotchetn"
FROM employees
START WITH employee_id = 100
CONNECT BY PRIOR employee_id = manager_id
ORDER SIBLINGS By first_name ;
```

Пример. Запрос вниз по иерархии должностей

```
 SELECT LEVEL - 1 As "Level podchin",
SYS_CONNECT_BY_PATH ( job_id, '/' ) rrrrr, last_name "Fam"
FROM   employees
 START WITH job_id = 'AD_PRES'
CONNECT BY PRIOR employee_id = manager_id ;
```

Запрос «в строку»

```
 SELECT SYS_CONNECT_BY_PATH(last_name,' --> ') TREE
   FROM employees
   CONNECT BY PRIOR manager_id=employee_id
```

Транзакции в Oracle SQL.

В Oracle нет явного оператора, чтобы начать транзакцию, но и нет автоматического завершения транзакции. ТРАНЗАКЦИЯ - это ряд операций манипулирования данными SQL, которые выполняют логическую единицу работы. Например, две операции UPDATE кредитуют один банковский счет и дебетируют другой. В один момент времени Oracle либо делает постоянными, либо отменяет все изменения в базе данных, осуществленные транзакцией. Если ваша программа сбивается в середине транзакции, Oracle обнаруживает ошибку и выполняет отмену (откат) транзакции. Следовательно, база данных автоматически возвращается в свое прошлое состояние. Первое предложение SQL в вашей программе начинает транзакцию. Когда одна транзакция заканчивается, очередное предложение SQL автоматически начинает следующую транзакцию. Таким образом, каждое предложение SQL является частью некоторой транзакции.

Все изменения со стороны индивидуальных команд DML заносятся Oracle в БД только группами, в рамках транзакции, по завершению транзакции.

Завершение транзакции с фиксацией изменений, внесенных операторами DML, происходит только по выдаче (а) команды COMMIT или (б) оператора DDL (скрытым образом завершающего свои действия по изменению таблиц словаря-справочника той же командой COMMIT).

Операторы управления транзакциями:

COMMIT

ROLLBACK

SAVEPOINT

ROLLBACK TO

SET TRANSACTION

- **COMMIT** - завершает транзакцию и делает любые выполненные в ней изменения постоянными. Освобождаются блокировки.
- **ROLLBACK** - Оператор отката завершает транзакцию и отменяет все выполненные в ней и незафиксированные изменения. Для этого он читает информацию из сегментов отката и восстанавливает блоки данных в состояние, в котором они находились до начала транзакции. Освобождаются блокировки.

По завершении транзакции необходимо явно указывать одну из команд завершения транзакции иначе за вас это сделает среда, в которой вы работаете (а среда не всегда это делает так, как вы предполагаете).

- **SAVEPOINT -** Позволяет создать в транзакции точку сохранения. В одной транзакции можно выполнять оператор SAVEPOINT несколько раз, устанавливая несколько точек сохранения. Точки сохранения позволяют устанавливать маркеры внутри транзакции таким образом, чтобы была возможность отмены только части работы, проделанной в транзакции. Оправдано использование точек сохранения в продолжительных и сложных транзакциях. ORACLE освобождает блокировки, которые были установлены отменённым оператором.
- **ROLLBACK TO <точка сохранения> -** Этот оператор используется совместно с представленным выше оператором SAVEPOINT. Транзакцию можно откатить до указанной точки сохранения, не отменяя все сделанные до нее изменения. Таким образом, можно выполнить два оператора UPDATE, затем — оператор SAVEPOINT, а после него — два оператора DELETE. При возникновении ошибки или исключительной ситуации в ходе выполнения операторов DELETE транзакция будет откатываться до указанной оператором SAVEPOINT точки сохранения; при этом будут отменяться операторы DELETE, но не операторы UPDATE.
- **SET TRANSACTION** - Этот оператор позволяет устанавливать атрибуты транзакции, такие как уровень изолированности и то, будет ли она использоваться только для чтения данных или для чтения и записи. Этот оператор также позволяет привязать транзакцию к определенному сегменту отката.

Некоторые особенности выполнения транзакций в Oracle:

1.Транзакция обычно состоит из нескольких операторов DML. Если один оператор дает сбой, то он один откатывается. То есть все операторы, которые раньше были выполнены, не откатываются автоматически – результаты их работы не пропадают. Вы можете дальше продолжать транзакцию. Затем её или зафиксировать, или откатить. Получается такой эффект потому, что Oracle каждый оператор транзакции помещает в неявные операторы Savepoint так, как это показано далее:

Savepoint statement1;

Оператор1;

If error then rollback to statement1;

Savepoint statement2;

Оператор2;

If error then rollback to statement2;

2. Понятие неделимости распространяется на необходимую глубину. Например, мы вставляет записи в таблицу 1, что вызывает срабатывание триггера на вставку записей в таблицу 2, что в свою очередь вызывает срабатывание триггера на обновление таблицы 3 и так далее. Если в какой-то момент происходит откат нашего оператора по таблице 1, то отменяются и все изменения, произведенные в таблице 2,3, и т.д. То есть или все изменения фиксируются, или все отменяется.

3. Oracle анонимный блок PL/SQL считает оператором. Например, begin оператор1; оператор2; end; То есть для него применимо предыдущее замечание.

4. Ограничение целостности проверяются после выполнения каждого sql-оператора. Oracle разрешает делать некоторые строки таблицы несогласованными до конца выполнения sql-оператора.

5. В Oracle есть возможность отложить проверку целостности на любой момент времени до конца транзакции. Это реализуется с помощью ограничения deferrable таблицы и перевода ограничения в режим deferred.

6. В целях экономии места в сегментах отката не фиксируйте изменения чаще, чем это нужно по логике программы. Просто нужно определить оптимальный размер сегментов отката.

7. В Oracle можно использовать распределенные транзакции, то есть выполнять транзакцию, в которой операторы работают на удаленных сервера (распределенная база данных). Для доступа к удаленной базе данных используется объект database link. Распределённая транзакция выглядит примерно так:

```
update table1 set x=1;
update remote_table1@remote_database set x=1;
commit;
```

Распределённая транзакция имеет то же свойство, что и обычная: все или ничего. Только фиксация происходит в две стадии (двухфазная фиксация транзакции): сначала мастер-сервер опрашивает о готовности все подчинённые сервера, затем, в случае если все сервера готовы, дает команду фиксировать транзакцию. Если хотя бы один сервер при опросе был не готов, то транзакция откатывается на всех серверах.

8. В Oracle продолжительность транзакции не ограничивается, потому что проблемы поедания ресурсов блокировками не существует. Транзакция длится столько, сколько нужно приложению. Единственная проблема: при очень длительных транзакциях и маленьком сегменте отката возможна ошибка ORA-1555.

Существует рекомендации Тома Кайта, которые (как мне кажется) очень верны:

При разработке приложений баз данных я использую очень простую мантру:

если можно, сделай это с помощью одного оператора SQL(потому что это будет быстрее);

если это нельзя сделать с помощью одного оператора SQL, сделай это в PL/SQL;

если это нельзя сделать в PL/SQL, попытайся использовать хранимую процедуру на языке Java;

если это нельзя сделать в Java, сделай это в виде внешней процедуры на языке C;

если это нельзя реализовать в виде внешней процедуры на языке C, надо серьезно подумать, зачем это вообще делать.

Пример: Во время удаления записи из таблицы TEST случайно стерты все данные этой таблицы. Ошибка исправляется, посылается правильная команда, и изменения фиксируются.

```
SQL> DELETE FROM test;
25,000 rows deleted.

SQL> ROLLBACK;
Rollback complete.

SQL> DELETE FROM test
2       WHERE       id = 100;
1 row deleted.

SQL> SELECT *
2       FROM test
3       WHERE id = 100;
no rows selected.

SQL> COMMIT;
Commit complete.
```