Министерство науки и высшего образования Российской Федерации
Ярославский государственный университет им. П. Г. Демидова
Кафедра теоретической информатики

А. В. Смирнов

ОСНОВЫ ORACLE PL/SQL

Учебно-методическое пособие

Ярославль
ЯрГУ
2020

УДК 004.43(075.8)
ББК З973.2-0181я73
С 50

Рекомендовано
Редакционно-издательским советом университета
в качестве учебного издания. План 2020 года

Рецензент
кафедра теоретической информатики ЯрГУ

Смирнов, Александр Валерьевич.

С 50 Основы Oracle PL/SQL : учебно-методическое пособие / А. В. Смирнов ; Яросл. гос. ун-т им. П. Г. Демидова. — Ярославль : ЯрГУ, 2020. — 100 с.

В учебно-методическом пособии рассматриваются основные объекты СУБД Oracle и основные конструкции языка PL/SQL. Особое внимание уделено работе с коллекциями, триггерами и объектными типами. Содержит теоретические сведения, необходимые для освоения Oracle PL/SQL, а также для выполнения лабораторных работ и подготовки к экзамену. Все главы сопровождаются упражнениями по изученному материалу.

Предназначено для студентов, обучающихся по дисциплинам «Объектные базы данных» и «Система управления базами данных Oracle».

УДК 004.43(075.8)
ББК З973.2-0181я73

Оглавление

Необходимые пояснения и обозначения

Все положения настоящего пособия будут иллюстрироваться примерами. Большинство из них будут построены для базы данных «Университет» со следующей схемой:

- **Student** (id*, name, birthday, gender, s_pass, n_pass, agrant, group_name);
- **AGroup** (name*, faculty, course, head, specialty, mag);
- **Discipline** (cipher*, name, hours, control);
- **Group_Disc** (group_name*, disc_cipher*).

В таблице Student содержится информация о студентах: идентификатор (номер студенческого билета), имя (в формате «Фамилия И. О.»), дата рождения, пол (одной буквой — «М» или «Ж»), серия и номер паспорта, размер стипендии, название группы.

В таблице AGroup содержатся сведения об учебных группах: название, факультет, курс (цифрой), идентификатор старосты, специальность, признак магистратуры (в текстовом формате — «Да» или «Нет»).

В таблице Discipline содержатся данные о дисциплинах: шифр (число), название, количество часов, форма отчётности («Зачёт» или «Экзамен»).

Поскольку одна и та же дисциплина может преподаваться разным группам, а одной группе читается несколько дисциплин, организуется дополнительная таблица Group_Disc для связей между группой и дисциплиной.

В приведённой схеме *первичные ключи* таблиц отмечены звёздочкой, а *внешние ключи* — подчёркиванием. Понятие *ключа* в Oracle трактуется так же, как и в любой реляционной СУБД, подробно данный вопрос мы рассмотрим в главе 2.

Отметим также, что мы используем идентификаторы AGroup и agrant вместо Group и grant, поскольку последние являются зарезервированными словами Oracle — они не могут быть идентификаторами.

В пособии приняты следующие *условные обозначения*:

- <текст> — пояснительный текст, указывающий на то, что должно быть написано в данном месте запроса или команды;

- выражение 1 | выражение 2 — с помощью вертикальной черты мы будем показывать, что в данном месте запроса или команды должно быть записано строго одно из объединённых таким образом выражений;

- {выражение 1 | выражение 2} — с помощью фигурных скобок мы будем ограничивать область действия операции выбора одной из альтернатив в тех случаях, когда возможна неоднозначность трактовки;

- [выражение] — с помощью квадратных скобок мы будем показывать опциональность (необязательность) присутствия той или иной части запроса или оператора;

- ... — с помощью многоточия мы будем показывать, что та или иная опциональная часть запроса или оператора может встречаться любое конечное число раз подряд.

СУБД Oracle нечувствительна к регистру текста в командах и запросах. Для лучшей читаемости кода команды зарезервированные слова и стандартные типы данных мы будем обозначать полностью заглавными буквами; названия объектов базы (таблиц, представлений, последовательностей и т. д.) — строчными буквами с заглавной первой буквой каждого слова; названия переменных, полей и пользовательских типов — полностью строчными буквами.

1. Язык манипулирования данными DML

В этой главе мы рассмотрим основные операторы DML (Data Manipulation Language, язык манипулирования данными):

1) `SELECT` — оператор выборки данных;

2) `INSERT` — оператор добавления записей;

3) `DELETE` — оператор удаления записей;

4) `UPDATE` — оператор обновления записей.

Отметим, что ряд функциональных возможностей указанных операторов должен быть знаком читателям из курса бакалавриата «Технологии баз данных», однако некоторые опции являются специфическими для Oracle. Подробную информацию о стандартном функционале языка SQL, общем для всех реляционных баз данных, можно найти в книге [3].

1.1. Создание простых запросов на выборку с помощью оператора SELECT

В простейшем случае оператор `SELECT` представляется в виде:

```
SELECT <attribute_list>
FROM <table_list>
[WHERE <condition>];
```

В списке выбора через запятую перечисляются имена атрибутов, по которым нужно осуществлять выборку, в том порядке, в котором они должны быть выданы. Если нужно осуществить выборку по всем атрибутам таблиц, это можно обозначить с помощью символа «`*`». Таблицы в ветке `FROM` перечисляются через запятую в произвольном порядке. К примеру, запрос

```
SELECT * FROM Student WHERE agrant > 2000;
```

вернёт список всех студентов, получающих стипендию, большую 2000 руб., причём будут выведены все атрибуты каждого кортежа.

В следующем запросе выборка осуществляется только по некоторым атрибутам:

```
SELECT name AS Имя,
       birthday AS "День рождения"
FROM Student WHERE birthday > '31.12.1998';
```

Если никоим образом не определить названия столбцов, возвращаемых запросом, они будут выданы с заголовками, соответствующими названиям атрибутов в таблице базы данных, преобразованных к верхнему регистру (в нашем случае — `NAME`

и BIRTHDAY). Поэтому целесообразным во многих случаях является задание названий столбцов в явном виде с помощью ключевого слова AS. Многословные названия обязательно указываются в двойных кавычках, однословные названия можно указывать без кавычек (но тогда они будут преобразованы к верхнему регистру). В примере будут сгенерированы столбцы с названиями ИМЯ и День рождения.

Обратите внимание также на то, что символьные константы и константы-даты в запросе должны быть заключены в одинарные кавычки, как в примере. Числовые константы должны быть указаны без кавычек. В Oracle определены несколько форматов записи даты, например, константа-дата '31-Dec-98' ('31-Дек-98' в русифицированной среде) тоже будет обработана верно.

Запрос из последнего примера выведет имена и дни рождения студентов, родившихся в 1999 году и позже.

Каждый кортеж в реляционной таблице является уникальным, однако в случае выборки по части атрибутов записи могут оказаться продублированными. Если эта ситуация недопустима, в запросе нужно указать ключевое слово DISTINCT — поиск только уникальных значений:

```
SELECT DISTINCT name AS "Имя",
                    birthday AS "День рождения"
FROM Student WHERE birthday > '31.12.1998';
```

Условия в ветке WHERE можно комбинировать, используя операторные скобки и ключевые слова NOT, AND, OR, соответствующие логическим операциям НЕ, И, ИЛИ. В Oracle используется стандартный приоритет операций: наивысший приоритет у отрицания, наименьший — у дизъюнкции. Запрос

```
SELECT * FROM Student
WHERE birthday > '31.12.1998' OR
      (birthday < '01.01.1996' AND NOT
      (name = 'Иванов И. И.'));
```

вернёт информацию о тех студентах, кто родился не ранее 1999 года, а также о тех студентах, кто родился не позже 1995 года, но чьё имя при этом не Иванов И. И.

Ключевое слово BETWEEN позволяет задать диапазон значений, который может быть представлен в любом формате (число, строка, дата). Для строк сравнение выполняется в соответствии с кодами символов. Левый и правый конец диапазона включается в выборку, при этом левый конец должен быть не больше правого, иначе запрос не вернёт ни одной строки. Концы диапазона соединяются с помощью ключевого слова AND. Так, запрос

```
SELECT * FROM Student
WHERE birthday BETWEEN
      '01.01.1996' AND '31.12.1998';
```
выдаст информацию о студентах, родившихся с 1996 по 1998 год включительно.

С помощью ключевого слова `IN` осуществляется проверка вхождения значения в указанный список:
```
SELECT * FROM Student
WHERE name IN ('Иванов И. И.', 'Петрова П. П.');
```
Список может быть также задан с помощью вложенного запроса (см. раздел 1.5).

Ключевое слово `IS NULL` осуществляет проверку на пустое значение, `IS NOT NULL` — на непустое значение. Выборка студентов, для которых не определена группа:
```
SELECT * FROM Student
WHERE group_name IS NULL;
```
Сортировку полученных значений можно осуществлять, указав ключевое слово `ORDER BY`. Порядок сортировки: `ASC` — по возрастанию; `DESC` — по убыванию. Количество уровней сортировки не ограничивается. По умолчанию используется сортировка по возрастанию. Следующий запрос вернёт список студентов, отсортированный по возрастанию имени группы, затем — по убыванию стипендии, затем — по возрастанию имени:
```
SELECT * FROM Student
ORDER BY group_name ASC, agrant DESC, name;
```
В средах SQL Plus и SQL Developer с помощью символа «&» можно определять переменные, значение которых будет предложено ввести пользователю, например:
```
SELECT * FROM Student
WHERE birthday > &min_birthday;
```
или
```
SELECT * FROM Student
WHERE birthday > '&min_birthday';
```
Второй вариант является более правильным, поскольку избавляет пользователя от необходимости указывать вводимую дату в одинарных кавычках. Первый вариант при этом допускает *инъекцию* SQL-кода — можно в качестве значения переменной указать вложенный SQL-запрос в круглых скобках.

Если значение переменной требуется сохранить для дальнейшего использования, нужно использовать знак «&&» при первом обращении к ней. Последующие обращения осуществляются с помощью «&». Например:
```
SELECT DISTINCT name FROM Student
WHERE birthday > '&&min_birthday';
...
```

```
SELECT DISTINCT group_name FROM Student
WHERE birthday > '&min_birthday';
```

Первый запрос выведет список имён студентов, дата рождения которых больше введённого пользователем значения `min_birthday`, а второй запрос отобразит названия групп, в которых есть такие студенты.

Также можно использовать операторы `DEFINE <переменная> = <значение>` для задания значения переменной и `UNDEFINE <переменная>` для сброса этого значения.

1.2. Некоторые функции для обработки данных в запросе

В данном разделе мы рассмотрим основные функции SQL, которые можно использовать внутри запросов. С полным списком функций можно ознакомиться в [1] и [3].

Функции мы будем рассматривать по группам. Параметры во всех функциях могут быть заданы как переменными, так и константами соответствующего типа, и именами атрибутов таблиц базы данных.

Функции работы с текстом

`LOWER(<text>)` — выполняет преобразование текста к нижнему регистру.

`UPPER(<text>)` — выполняет преобразование текста к верхнему регистру.

`INITCAP(<text>)` — выполняет преобразование текста к виду, где каждое слово начинается с заглавной буквы, а все остальные буквы — строчные. Эта функция удобна, например, при форматировании полей, содержащих ФИО.

`SUBSTR(<text>, <begin>, <length>)` — находит подстроку в строке `<text>`, начинающуюся с символа с номером `<begin>`, длины `<length>`. Начальный номер задаётся целым числом, индексация элементов строки начинается с 1. Если начальный номер отрицательный, отсчёт стартовой позиции ведётся с конца строки.

`LENGTH(<text>)` — возвращает длину строки.

`INSTR(<text1>, <text2>[, <start>[, <n_app>]])` — ищет в строке `<text1>` подстроку `<text2>` и возвращает номер символа, с которого начинается n-е вхождение подстроки в строку ($n = 1$, если не указать опциональный параметр `<n_app>`). Если нужное вхождение не найдено, функция вернёт 0. Опциональный параметр `<start>` указывает, с какого символа на-

чать поиск. При отрицательном значении параметра поиск ведётся с конца строки.

LPAD(<text>, <number>[, <symbols>]) — дополняет слева строку <text> до длины <number>, используя символы <symbols> (по умолчанию используется пробел). Функция обычно применяется для форматирования вывода.

RPAD(<text>, <number>[, <symbols>]) — аналогичная функция для дополнения строки справа.

REPLACE(<text1>, <text2>[, <text3>]) — в строке <text1> ищутся все вхождения подстроки <text2> и заменяются на <text3>. Если третий параметр равен NULL или не указан, то все вхождения подстроки <text2> просто исключаются из строки <text1>.

TRIM([[LEADING | TRAILING | BOTH] <text1> FROM] <text2>) — спереди, сзади или с обоих концов строки <text2> отрезаются все символы, соответствующие шаблону <text1>. По умолчанию используется опция BOTH. Если первый параметр не указан, отрезаются все пробелы. Обратите внимание на синтаксис функции — параметры соединяются ключевым словом FROM, а не запятой.

Функции работы с числами

ROUND(<number>, <position>) — округление числа с заданной точностью <position>. Значение 0 соответствует округлению до целого, положительное значение — округлению с точностью до указанного количества знаков после запятой, отрицательное — перед запятой (например, -2 определяет округление до сотен).

TRUNC(<number>, <position>) — срезание части числа с указанной точностью (округление вниз).

MOD(<number1>, <number2>) — нахождение остатка от деления первого числа на второе (для целых чисел).

ABS(<n>) — абсолютная величина числа.

FLOOR(<n>), CEIL(<n>) — округление числа до ближайшего целого снизу/сверху.

SIGN(<n>) — знак числа (-1, 0 или 1).

SIN(<n>), COS(<n>), TAN(<n>) — тригонометрические функции (параметр — в радианах).

ASIN(<n>), ACOS(<n>), ATAN(<n>) — обратные тригонометрические функции, возвращающие значение угла в радианах.

ASIN2(<n>), ACOS2(<n>), ATAN2(<n>) — обратные тригонометрические функции, возвращающие значение угла в градусах.

SINH(<n>), COSH(<n>), TANH(<n>) — гиперболические функции.

LN(<n>), LOG(<m>, <n>) — натуральный логарифм и логарифм по основанию <m>.

POWER(<n>, <m>) — возведение числа <n> в степень <m>.

SQRT(<n>) — квадратный корень неотрицательного числа.

Функции работы с датой

ROUND(<date>, <position>) — округление даты с заданной точностью <position>. Точность задаётся в строковом виде. Например, значению 'YEAR' будет соответствовать округление до годов (до ближайшего 1 января).

TRUNC(<date>, <position>) — срезание части даты с указанной точностью (округление вниз).

SYSDATE — системная дата и время (по умолчанию выводится только дата).

SYSTIMESTAMP — системная дата и время с долями секунд и временной зоной.

LAST_DAY(<date>) — находит последний день месяца, в который попадает указанная дата.

MONTHS_BETWEEN(<date1>, <date2>) — вычитает из первой даты вторую, результат выводится в месяцах (дробное число).

NEXT_DAY(<date>, <weekday>) — находит следующий относительно указанной даты день, являющийся указанным днём недели. День недели задаётся в виде целого числа от 1 до 7 либо в виде названия дня (например, 'Monday' для нерусифицированной СУБД или 'Понедельник' — для русифицированной). В зависимости от настроек СУБД, первым днём может быть понедельник или воскресенье.

ADD_MONTHS(<date>, <number>) — добавляет указанное целое число месяцев к дате. Функция корректно учитывает количество дней в месяцах, то есть при добавлении 1 месяца к 31 января результатом будет 28 или 29 февраля (в зависимости от года).

EXTRACT(<component> FROM <date>) — извлекает указанный компонент даты. Название компонента указывается в виде ключевого слова, без кавычек (например, MONTH или TIMEZONE_HOUR).

Функции преобразования формата

TO_NUMBER(<param>[, <format>[, nls_param]]) — функция преобразования к числовому формату.

TO_DATE(<param>[, <format>[, nls_param]]) — функция преобразования к формату даты.

`TO_CHAR(<param>[, <format>[, nls_param]])` — функция преобразования к строковому формату.

Все три функции устроены похожим образом — обязательно нужно указать преобразуемую величину, опционально — формат вывода (строковая константа) и параметры NLS (National Language Support, поддержка национальных языков). Параметры NLS предоставляют дополнительные возможности форматирования с учётом национальных стандартов. Подробней о способах форматирования можно прочитать в [1]. Если указать в функции преобразования только один параметр, будет применено стандартное форматирование СУБД.

Функции преобразования NULL-значений

Пустые (`NULL`) значения могут негативно повлиять на результат вычислений, поэтому во многих случаях имеет смысл преобразовать `NULL` к некоторому значению по умолчанию.

`NVL(<arg>, <value>)` — выводит значение аргумента, если оно не равно `NULL`, и `<value>` — в противном случае.

`COALESCE(<arg1>[, <arg2> ...], <value>)` — поочерёдно вычисляет значения аргументов, выводит первое неравное `NULL` значение. Если все аргументы равны `NULL`, выводится `<value>`.

Примеры

Рассмотрим несколько примеров использования функций.

Так, запрос

```
SELECT name, course FROM AGroup
WHERE INSTR(name,'ИТ') > 0 AND LOWER(mag) = 'да';
```

найдёт группы магистратуры, содержащие в своём названии аббревиатуру `'ИТ'`. Обратите внимание, что при сравнении признак магистратуры мы преобразуем к нижнему регистру, чтобы правильно учесть все возможные варианты написания.

В свою очередь, запрос

```
SELECT INITCAP(name),
       EXTRACT(YEAR FROM birthday)
FROM Student WHERE
      MONTHS_BETWEEN(SYSDATE, birthday) < 240;
```

выведет список ФИО студентов с указанием года рождения для всех студентов младше 20 лет. При этом ФИО будут отформатированы так, чтобы каждое слово начиналось с заглавной буквы.

Как уже было сказано, функция `SYSDATE` по умолчанию возвращает только дату, однако системная дата хранится вместе с временем. Следующий запрос позволяет получить системную дату и время в нужном нам формате:

```
SELECT TO_CHAR(SYSDATE,
        'DD-MON-YYYY HH24:MI:SS') FROM DUAL;
```

В запросе мы использовали функцию преобразования TO_CHAR с указанием формата, где DD, MON, YYYY, HH24, MI, SS — параметры формата, говорящие о том, что дни, часы, минуты и секунды нужно обозначать двумя цифрами, год — четырьмя цифрами, месяц — тремя заглавными буквами, а часы должны быть указаны в 24-часовом формате. Обратите внимание, что в списке выбора указана системная таблица DUAL. Она используется тогда, когда необходимо осуществить выборку значения, не входящего ни в одну из таблиц БД.

1.3. Агрегирование и групповые операции

Функции агрегирования используются для подсчёта разного рода сводных значений по множеству данных. Основные функции агрегирования в Oracle:

- AVG — подсчёт среднего арифметического;
- COUNT — подсчёт количества элементов выборки;
- MIN — нахождение минимального значения;
- MAX — нахождение максимального значения;
- SUM — подсчёт суммы.

Функции AVG и SUM работают только с числовыми значениями (NULL при этом трактуется как 0), остальные — с любыми, в том числе строковыми значениями. В Oracle определены и другие функции агрегирования, но они используются реже и остаются за рамками данного пособия.

В следующем примере определяется количество студентов, родившихся с 1996 по 1998 год включительно (будет выведено только одно число):

```
SELECT COUNT(*) FROM Student
WHERE birthday BETWEEN
      '01.01.1996' AND '31.12.1998';
```

Внутри функции агрегирования можно использовать ключевое слово DISTINCT, тогда функция будет учитывать только одну запись из нескольких записей с одинаковым значением атрибута. К примеру, учебные дисциплины могут иметь одинаковые названия при разных шифрах. Найдём количество дисциплин с различными названиями:

```
SELECT COUNT(DISTINCT(name)) FROM Discipline;
```

Мы рассмотрели простые примеры, в которых агрегирующие функции рассчитываются по всей выборке. Как правило, значения в выборке нужно группировать по определённым признакам и находить значение функции для каждой группы отдельно. Для этого нужно добавить в запрос ветку `GROUP BY`. Например, для нахождения средней стипендии по учебным группам нужно написать такой запрос:

```
SELECT group_name, AVG(agrant) FROM Student
GROUP BY group_name;
```

Группировку можно осуществлять сразу по нескольким атрибутам. Допустим, мы хотим найти среднюю стипендию в каждой учебной группе отдельно для девушек и для юношей. Тогда запрос примет вид:

```
SELECT group_name, gender, AVG(agrant)
FROM Student
GROUP BY group_name, gender
ORDER BY group_name;
```

Выводимые значения мы отсортировали по названию учебной группы для большей наглядности.

Отметим, что поля, по которым происходит агрегирование, могут не присутствовать в списке выбора, но при этом в списке выбора не может быть полей, не определяющих группу и не входящих в функцию агрегирования. Скажем, в список выбора последнего запроса нельзя добавить имя студента.

Проверку условия, относящегося к значению агрегирующей функции, нельзя проводить в ветке `WHERE`, так как фильтрация, выполняемая по условию из этой ветки, происходит до расчёта функций агрегирования. Для наложения условий на агрегированные значения следует использовать ветку `HAVING`:

```
SELECT group_name, AVG(agrant)
FROM Student
GROUP BY group_name
HAVING MAX(agrant) < 3000;
```

Здесь мы находим среднюю стипендию в учебных группах, максимальная стипендия в которых меньше 3000 руб.

В Oracle возможно использовать многоуровневую группировку. Для этого в ветке `GROUP BY` нужно использовать один из операторов `CUBE`, `ROLLUP` и `GROUPING SETS`. При этом различные виды многоуровневой группировки можно сочетать друг с другом, а также с обычной группировкой в рамках одного запроса, в том числе можно один вид группировки вкладывать в другой.

Оператор `CUBE` осуществляет группировку по *кубу*, то есть по всем возможным подмножествам (собственным и несобственным) указанного множества. Пустому множеству соответствует весь объём выборки. Так, запрос

```
SELECT group_name, gender, AVG(agrant)
FROM Student
GROUP BY CUBE (group_name, gender)
ORDER BY group_name, gender;
```

найдёт среднюю стипендию студентов по всем парам (название группы, пол), отдельно по группам безотносительно пола, отдельно для юношей и девушек безотносительно группы и по университету в целом.

Оператор ROLLUP осуществляет группировку по *свёртке*, то есть по всем подмножествам, состоящим из n, $n-1$, ..., 1, 0 первых атрибутов указанного множества. В отличие от группировки по кубу, при группировке по свёртке важно указать атрибуты в правильном порядке. К примеру, запрос

```
SELECT group_name, gender, AVG(agrant)
FROM Student
GROUP BY ROLLUP (group_name, gender)
ORDER BY group_name, gender;
```

найдёт среднюю стипендию студентов по всем парам (название группы, пол), отдельно по группам безотносительно пола и по университету в целом.

Оператор GROUPING SETS позволяет осуществить группировку по всем указанным множествам (пустое множество обозначается пустыми операторными скобками). Так, предыдущий пример со свёрткой может быть переписан в эквивалентном виде с использованием GROUPING SETS:

```
SELECT group_name, gender, AVG(agrant)
FROM Student
GROUP BY GROUPING SETS
      ((group_name, gender), (group_name), ())
ORDER BY group_name, gender;
```

Недостатком многоуровневой группировки является наличие NULL-значений атрибутов, по которым ведётся агрегирование. Скажем, в предыдущих примерах значение пола будет неопределённым, когда агрегирующая функция рассчитывается только по учебной группе. Указанный недостаток можно преодолеть, используя в списке выбора сочетание функций DECODE и GROUPING.

GROUPING(<attribute>) — возвращает 1, если в результате группировки значение атрибута стало неопределённым, и 0 в противном случае. Отметим, что, если значение атрибута изначально было равно NULL (например, рассчитывалось среднее для студентов с неопределённой группой), функция вернёт 0.

DECODE(<expression>, <val1>, <res1> [, <val2>, <res2> ...][, <def>]) — аналог условного оператора. Функция последовательно проверяет равенство выражения указанным

значениям, при первом совпадении выводится соответствующий результат. Если ни одно значение не подошло, будет выведено значение по умолчанию (или NULL, если значение по умолчанию не указано).

Преобразуем пример с группировкой по кубу, чтобы продемонстрировать возможности данных функций:

```
SELECT
  DECODE(GROUPING(group_name),1,'Все группы',
    NVL(group_name,'Без группы')) AS "Группа",
  DECODE(GROUPING(gender),1,
    'Все студенты',gender) AS "Пол",
  ROUND(AVG(agrant),0) AS "Средняя стипендия"
FROM Student
GROUP BY CUBE (group_name, gender)
ORDER BY group_name, gender;
```

В итоге все NULL-значения атрибутов группировки будут заменены на текстовые константы 'Все группы' или 'Все студенты'. Обратите внимание, что мы используем NVL для вывода имени группы. Это связано с тем, что у нас в БД может быть сохранена информация о студенте без указания учебной группы, то есть в таблице будет храниться NULL. По неопределённой учебной группе тоже будет выполнена группировка (функция DECODE при этом вернёт 0), соответственно, мы можем отдельно отформатировать вывод для этого случая. Отметим, что неопределённое значение пола в таблице Student не допускается, поэтому для этого поля мы функцию NVL не используем.

Кроме того, в последнем запросе мы выполнили некоторое «косметическое» форматирование — придали осмысленные русскоязычные названия столбцам и округлили среднее значение стипендии до рублей.

1.4. Использование соединений и операций над множествами в операторе SELECT

При использовании в запросе значений из нескольких таблиц эти таблицы необходимо соединить по тому или иному условию. Условие может быть записано либо в ветке WHERE, либо с помощью одного из операторов соединений. Рассмотрим запрос:

```
SELECT AGroup.name, AGroup.course,
       AGroup.faculty, Discipline.name
FROM AGroup, Discipline, Group_Disc
WHERE AGroup.name = Group_Disc.group_name
   AND Discipline.cipher =
                    Group_Disc.disc_cipher
```

```
ORDER BY AGroup.name, Discipline.name;
```

Здесь условие соединения задано явным образом в ветке `WHERE` как равенство соответствующих атрибутов. Отметим, что можно задать любое другое условие, диктуемое логикой задачи. В нашем же примере будет выведен список групп с указанием курса и факультета, а также названий предметов, которые нужно сдавать этим группам. Обратите внимание, что при обращении к полям таблиц мы использовали *квалификаторы* — название таблицы с точкой, записанное перед названием атрибута. Такой формат записи совершенно необходим в случаях, когда в соединяемых таблицах есть атрибуты с одинаковыми названиями.

При соединении таблиц можно использовать один из вариантов оператора `JOIN`. Наиболее простой из них — *естественное*, или *эквисоединиение* `NATURAL JOIN`. При естественном соединении таблиц СУБД ищет атрибуты с одинаковыми названиями и типами данных и соединяет их по условию-равенству, поэтому пользоваться эквисоединением нужно аккуратно. Скажем, в предыдущем запросе замена соединения на натуральное приведёт к тому, что таблицы `AGroup` и `Discipline` будут соединены по одинаковому атрибуту `name`, что приведёт к пустому результату запроса. Однако такой запрос:

```
SELECT name, course, faculty, disc_cipher
FROM AGroup NATURAL JOIN Group_Disc;
```

будет работать вполне корректно. Не найдя полей с одинаковым названием, Oracle соединит таблицы по полям `name` и `group_name`, совпадающим по типу данных.

Условие соединения можно задать с помощью регулярного соединения `JOIN`. Например, первый запрос можно переписать в виде:

```
SELECT AGroup.name, AGroup.course,
       AGroup.faculty, Discipline.name
FROM AGroup JOIN Group_Disc
   ON (AGroup.name = Group_Disc.group_name)
   JOIN Discipline ON (Discipline.cipher =
                       Group_Disc.disc_cipher)
ORDER BY AGroup.name, Discipline.name;
```

Здесь условие соединения задавалось в ветке `ON`. Если соединяемые атрибуты имеют одинаковые названия и тип данных, а соединение идёт по условию-равенству, тогда можно вместо `ON` использовать ветку `USING`, указывая только название атрибута.

Отметим, что во всех рассмотренных примерах неопределённые значения атрибутов, по которым ведётся соединение, не войдут в выборку. Если эти значения нужны, следует использовать одно из *внешних соединений*.

Левое внешнее соединение (`LEFT OUTER JOIN`) добавляет в выборку элементы первой таблицы, для которых нет соответствия во второй таблице. *Правое внешнее соединение* (`RIGHT OUTER JOIN`), наоборот, добавляет в выборку элементы второй таблицы, для которых нет соответствия в первой таблице. *Полное внешнее соединение* (`FULL OUTER JOIN`) работает как совокупность левого и правого соединения. В Oracle одностороннее внешнее соединение можно организовать с помощью регулярного соединения, если отметить в ветке `ON` предикатом `(+)` элементы той таблицы, для которой нужно вывести пустые записи. Для примера рассмотрим два варианта организации запроса, выводящего список студентов с указанием группы и факультета, включающего студентов, для которых группа не определена. Первый вариант:

```
SELECT Student.id, Student.name,
       NVL(AGroup.name,'Без группы'),
       NVL(AGroup.faculty,'Без факультета')
FROM Student LEFT OUTER JOIN AGroup
ON Student.group_name = AGroup.name;
```

Второй вариант:

```
SELECT Student.id, Student.name,
       NVL(AGroup.name,'Без группы'),
       NVL(AGroup.faculty,'Без факультета')
FROM Student JOIN AGroup
ON Student.group_name = AGroup.name (+);
```

Можно соединять между собой несколько экземпляров одной таблицы, но в этом случае нужно использовать *алиасы* для различения экземпляров. Так, запрос

```
SELECT s1.name ||
       ' родился в один день с ' || s2.name
FROM Student s1, Student s2
WHERE s1.birthday = s2.birthday AND
      s1.id < s2.id AND
      s1.group_name LIKE 'ИТ-%' AND
      s2.group_name LIKE 'ИТ-%';
```

выведет все пары студентов специальности «Фундаментальная информатика и информационные технологии» (название группы начинается с префикса «ИТ-»), родившихся в один день. Поскольку связь по дню рождения не является иерархической, мы добавили условие `s1.id < s2.id`, чтобы исключить дубликаты и связывание студента с самим собой. Для проверки префикса группы мы используем регулярное выражение `'ИТ-%'`. Oracle предоставляет широкие возможности по использованию регуляр-

ных выражений (см. [1], [2]), однако в данном пособии мы их рассматривать не будем.

Если два запроса и более имеют одинаковые списки выбора (одинаковые имена и типы атрибутов), с соответствующими результатами выборки можно работать, используя операции над множествами:

- `UNION` — объединение множеств (без дубликатов);
- `UNION ALL` — объединение множеств (с дубликатами);
- `INTERSECT` — пересечение множеств;
- `MINUS` — разность множеств.

Общий синтаксис:

```
<Запрос 1> <Оператор 1> <Запрос 2>
[<Оператор 2> <Запрос 3> ...];
```

Отметим, что приоритет всех операций в Oracle считается равным, для регулирования порядка вычислений следует использовать операторные скобки.

1.5. Использование вложенных запросов

Если по каким-то причинам невозможно получить требуемые данные с помощью одного простого запроса, имеет смысл использовать *вложенные запросы* (*подзапросы*), которые могут быть вставлены практически в любую ветку оператора `SELECT`. Вложенный запрос всегда заключается в скобки. Чаще всего вложенные запросы используются при фильтрации значений (ветка `WHERE`). Например, запрос

```
SELECT name, course, specialty
FROM AGroup WHERE faculty =
  (SELECT faculty FROM AGroup
   WHERE name = 'ИТ-21МО');
```

найдёт все группы факультета, к которому относится группа «ИТ-21МО». Заметим, что здесь мы использовали условие вида `<атрибут> = (<подзапрос>)`. Поскольку имя группы является первичным ключом, вложенный запрос вернёт не более одного значения и проверка будет корректна. В общем случае, когда подзапрос может вернуть любое количество значений, следует использовать вместо равенства ключевое слово `IN`:

```
SELECT name, birthday, group_name
FROM Student WHERE group_name IN
  (SELECT group_name FROM Student
```

```
      WHERE name = 'Петрова П. П.');
```
Здесь мы выводим всех одногруппников студентки с указанным именем. Таких студенток в базе может быть несколько, поэтому использовать равенство нельзя.

Ключевые слова ANY и ALL позволяют проверить соответствие любому и всем значениям из диапазона. Запрос
```
SELECT * FROM AGroup
WHERE faculty != 'ИВТ' AND specialty = ANY
  (SELECT specialty FROM AGroup
   WHERE faculty = 'ИВТ');
```
находит все группы университета, где обучение проходит по *какой-либо* из специальностей факультета ИВТ, исключая группы факультета ИВТ. В свою очередь, запрос
```
SELECT * FROM Student WHERE agrant > ALL
  (SELECT NVL(agrant,0) FROM Student, AGroup
   WHERE Student.group_name = AGroup.name
         AND LOWER(mag) = 'нет');
```
выведет список всех студентов магистратуры, получающих стипендию большую, чем *все* студенты бакалавриата.

Ключевые слова EXISTS и NOT EXISTS позволяют проверить, возвращает ли вложенный запрос значения. Скажем, для поиска всех пустых учебных групп мы можем написать такой запрос:
```
SELECT * FROM AGroup g WHERE NOT EXISTS
  (SELECT * FROM Student s
   WHERE s.group_name = g.name);
```
Отметим, что во вложенном запросе мы обращаемся к элементам таблицы из основного запроса, используя алиасы. Поэтому вложенный запрос будет заново выполняться для каждой строки выборки основного запроса.

С помощью ключевого слова IN можно проверять соответствие сразу нескольких полей списку выбора вложенного запроса:
```
SELECT * FROM Discipline
WHERE (hours, control) IN
  (SELECT hours, control
   FROM Discipline, Group_Disc
   WHERE disc_cipher = cipher AND
         group_name = 'ИТ-21МО');
```
В результате будет получен список дисциплин, у которых количество часов и форма отчётности совпадают с количеством часов и формой отчётности какой-то дисциплины, преподаваемой группе «ИТ-21МО». Если требуется проверить более сложное условие, нежели вхождение в список, целесообразно вложенный запрос переместить в ветку FROM:
```
SELECT * FROM Discipline,
  (SELECT hours AS h, control AS c
```

```
    FROM Discipline, Group_Disc
    WHERE disc_cipher = cipher AND
          group_name = 'ИТ-21МО')
  WHERE control = c AND hours >= 2 * h;
```

В списке выбора оператора SELECT можно использовать вложенный запрос CASE. Синтаксис оператора CASE напоминает синтаксис условного оператора:

```
CASE WHEN <условие> THEN <значение>
[WHEN <условие> THEN <значение> ...]
[ELSE <значение> ...] END
```

Условия будут последовательно перебираться, пока не встретится верное. Скажем, мы хотим посчитать для группы ИТ-21МО среднее количество часов по дисциплинам, дополняя при расчёте среднего каждое значение до ближайшей полусотни сверху, зная, что больше 250 часов быть не может. Тогда запрос будет выглядеть так:

```
SELECT AVG(CASE WHEN hours <= 50 THEN 50
                WHEN hours <= 100 THEN 100
                WHEN hours <= 150 THEN 150
                WHEN hours <= 200 THEN 200
                ELSE 250 END)
FROM Discipline, Group_Disc
WHERE disc_cipher = cipher AND
      group_name = 'ИТ-21МО';
```

Оператор WITH позволяет создать и именовать вложенный запрос до выполнения основного запроса. Именованные подзапросы перечисляются через запятую, в каждом следующем допустимы ссылки на предыдущие:

```
WITH head_grant AS
    (SELECT g.name, NVL(s.agrant,0) agr
     FROM AGroup g, Student s
     WHERE g.head = s.id),
avg_hd_grant AS
    (SELECT AVG(agr) avgr FROM head_grant)
SELECT * FROM head_grant
WHERE agr > (SELECT avgr FROM avg_hd_grant);
```

Такой запрос выведет список названий групп и стипендий старост для тех групп, в которых стипендия старосты больше средней стипендии всех старост в университете.

1.6. Иерархические запросы

Иерархические запросы позволяют осуществлять обход в глубину иерархии, реализуемой той или иной таблицей. Иерархический запрос определяется добавлением в оператор SELECT веток:

```
[START WITH <начальный элемент>]
CONNECT BY [NOCYCLE] <условие>
```

Условие в ветке `CONNECT BY` задаётся в виде равенства двух атрибутов, один из которых отмечен ключевым словом `PRIOR` — указателем на родительский элемент иерархии. Фактически `PRIOR` определяет направление движения по иерархии (сверху вниз или снизу вверх). В ветке `CONNECT BY` нельзя использовать вложенные запросы. Опция `NOCYCLE` используется в случае циклических иерархий для прерывания бесконечных циклов. С помощью ветки `START WITH` можно определить начальный элемент иерархии, условие в этой ветке подобно условию в ветке `WHERE`. Ветка `WHERE` тоже может присутствовать в запросе для фильтрации выбранных данных. Оператор `ORDER SIBLINGS BY` используется для сортировки выбранных значений на каждом уровне иерархии.

Пусть в нашей базе «Университет» имеется таблица подразделений `Departments`. У каждого подразделения есть номер `dept_id` , название `dept_name` и указатель на номер вышестоящего подразделения `chief_dept`. Реализуем запрос, который осуществляет обход иерархии сверху вниз:

```
SELECT dept_id, dept_name, LEVEL
FROM Departments
CONNECT BY PRIOR dept_id = chief_dept;
```

В запросе мы использовали псевдостолбец `LEVEL`, который содержит номер уровня иерархии. Если бы нам нужно было обойти иерархию снизу вверх, в условии соединения мы бы прописали `dept_id = PRIOR chief_dept`.

Реализуем теперь более сложный запрос: будем выводить только те части иерархии, которые начинаются с факультетов. Подразделения на каждом уровне иерархии упорядочим по названиям. Направление движения прежнее.

```
SELECT dept_id, dept_name, LEVEL
FROM Departments START WITH
        LOWER(dept_name) LIKE '%факультет%'
CONNECT BY PRIOR dept_id = chief_dept
ORDER SIBLINGS BY dept_name;
```

1.7. Операторы манипулирования данными

Оператор `INSERT` используется для добавления новых записей в таблицу. В простейшем случае при добавлении одного кортежа запрос будет выглядеть так:

```
INSERT INTO Discipline
VALUES (888888, 'Объектные базы данных',
        108, DEFAULT);
```

Здесь значения атрибутов указаны в явном виде в порядке следования атрибутов в определении таблицы, поле «Форма отчётности» заполняется значением по умолчанию («Экзамен»).

Более правильный способ составления запроса на вставку предполагает указание в произвольном порядке имён атрибутов, которым будут присвоены значения. Атрибуты можно задать списком:

```
INSERT INTO Discipline (cipher, name, hours)
VALUES(888888,'Объектные базы данных',108);
```

или с помощью подзапроса:

```
INSERT INTO
(SELECT cipher, name, hours FROM Discipline)
VALUES(888888,'Объектные базы данных',108);
```

Поля, заполняемые значениями по умолчанию, можно при этом не указывать.

При добавлении нескольких кортежей с помощью одного запроса ветка `VALUES` заменяется на вложенный запрос типа `SELECT`. Запрос может быть сколь угодно сложным. Пусть, например, мы добавили в таблицу `Group_Disc` информацию обо всех предметах группы ИТ-11БО. Понятно, что у группы ИТ-12БО должен быть такой же набор предметов. Их добавление можно осуществить с помощью одного запроса:

```
INSERT INTO Group_Disc
SELECT disc_cipher, 'ИТ-12БО'
FROM Group_Disc
WHERE group_name = 'ИТ-11БО';
```

Для удаления данных из таблицы используется оператор `DELETE`. Если нужно удалить все данные, оператор не содержит проверки условий:

```
DELETE FROM Student;
```

Как правило, удалять нужно всё-таки только часть кортежей, поэтому в операторе `DELETE` можно использовать ветку `WHERE` такую же, как в операторе `SELECT`:

```
DELETE FROM Student WHERE group_name IS NULL
                    AND NVL(agrant,0) = 0;
```

Для модификации данных в таблице используется оператор `UPDATE`. Новые значения обновляемых полей устанавливаются оператором `SET`. К примеру, перевод студентов группы ИТ-11БО на следующий курс будет выглядеть так:

```
UPDATE Student
SET group_name = 'ИТ-21БО'
WHERE group_name = 'ИТ-11БО';
```

При модификации данных также можно использовать вложенные запросы:

```
UPDATE AGroup g
SET head = (SELECT id FROM Student
            WHERE group_name = g.name
            ORDER BY agrant DESC
            FETCH FIRST ROW ONLY),
(faculty, course) =
   (SELECT 'ИВТ',
           (CASE WHEN course - 1 = 0 THEN 1
            ELSE course - 1 END)
    FROM AGroup h WHERE h.name = g.name)
WHERE faculty != 'ИВТ' AND specialty IN
   (SELECT specialty FROM AGroup
    WHERE faculty = 'ИВТ');
```

В этом запросе мы переводим на факультет ИВТ все группы других факультетов, у которых специальность совпадает с любой специальностью факультета ИВТ. Группы переводятся с потерей курса, если этот курс не первый, старостой назначается студент этой группы с максимальной стипендией (если таких студентов несколько, выбирается первый найденный). Последнее условие задаётся командой `FETCH FIRST ROW ONLY` внутри соответствующего подзапроса. Если бы нам нужно было получить несколько первых записей, возвращаемых запросом, мы бы использовали команду `FETCH FIRST <n> ROWS ONLY`.

Работа с операторами DML всегда осуществляется посредством *транзакций*. Каждая транзакция представляет собой последовательность операторов DML, сопровождаемых единственным оператором DDL. После применения последнего оператора DML в транзакции БД должна остаться целостной и непротиворечивой. Оператор DDL целиком подтверждает или отменяет транзакцию.

Оператор `COMMIT` подтверждает транзакцию. Только после выполнения этого оператора все изменения вступают в силу.

В случае если в ходе транзакции произошла ошибка, автоматически осуществляется откат, с тем чтобы не была нарушена целостность базы данных. Откат можно также осуществить с помощью оператора `ROLLBACK` (к предыдущему подтверждённому состоянию) или `ROLLBACK TO SAVEPOINT <name>` (к точке сохранения). При этом точка сохранения должна быть отмечена оператором `SAVEPOINT <name>`. В результате отката все неподтверждённые операции будут отменены, в том числе операции удаления.

Если нам необходимо удалить все данные из некоторой таблицы безвозвратно, можно использовать оператор `TRUNCATE <table>`, который является оператором DDL. Операция удаления в этом случае подтверждается автоматически.

При выходе из SQL Plus или SQL Developer также происходит автоподтверждение.

1.8. Упражнения

1. Напишите запросы на выборку следующих данных:

- список всех студентов 1 курса бакалавриата;
- список всех студентов с указанием возраста в годах (неполные годы отбрасываются);
- список студентов, получающих самую большую стипендию на своём курсе своего факультета;
- список зачётов, которые должен сдать студент Иванов И.И.;
- среднее количество студентов в группе, рассчитанное для каждой специальности безотносительно факультета, каждого курса каждого факультета, каждого факультета и университета в целом (всё должно быть реализовано в одном запросе);
- список групп с указанием средней стипендии, в котором отмечены группы, в которых средняя стипендия больше средней стипендии по этому курсу этой формы обучения этого факультета, а также отмечены группы бакалавриата, в которых средняя стипендия больше средней стипендии в магистратуре того же факультета.

2. Разработайте таблицу, реализующую иерархию, и напишите иерархический запрос к ней.

3. Напишите запрос на добавление практикумов с формой отчётности «Зачёт» по всем дисциплинам с формой отчётности «Экзамен», объём которых больше 200 часов. Количество часов новых дисциплин должно составлять половину от количества часов соответствующих старых дисциплин.

4. Напишите запрос на удаление пустых групп.

5. Напишите запрос, проводящий индексацию на 10 % стипендий всех студентов магистратуры.

2. Язык определения данных DDL

В этой главе мы рассмотрим основные операторы DDL (Data Definition Language, язык определения данных), связанные с созданием и управлением объектами схемы. Под *схемой* принято понимать набор объектов, находящихся во владении определённого пользователя. Так, схема `HR` — это набор объектов, принадлежащих пользователю `HR`. К примеру, таблица `Table1`, принадлежащая этому пользователю, будет именоваться `HR.Table1`. Отметим, что указание имени пользователя при обращении к объекту своей схемы необязательно.

Все команды DDL, в отличие от команд DML, подтверждаются автоматически.

2.1. Типы данных в Oracle

Прежде чем начать обсуждение операторов DDL, нам потребуется определить основные типы данных, использующиеся в Oracle (полный список можно найти в [1]). Будем рассматривать их по группам форматов.

`Текстовые форматы`

Представление текстовых форматов существенно зависит от кодировки символов, определяемой настройками СУБД.

`VARCHAR2(<size> [BYTE|CHAR])` — строка символов переменной длины (от 1 до 4000 символов, но не более 4000 байт; если в настройках СУБД параметр `MAX_STRING_SIZE` прописан как `EXTENDED`, максимальное значение — 32767). Параметр `<size>` определяет максимальную длину строки, при этом в БД хранится ровно столько символов, сколько записано в строке, пустые символы не добавляются. Опция `BYTE|CHAR` определяет семантику строки. Если выбран вариант `BYTE` (вариант по умолчанию), то строка хранится в виде набора байтов и все функции работы со строками индексируют её по байтам. Если выбран вариант `CHAR`, то строка рассматривается как набор символов. При однобайтовых кодировках разницы между опциями нет.

`NVARCHAR2(<size>)` — строка Unicode-символов переменной длины (до 4000 символов, но не более 4000 байт). Используется только символьная семантика.

`CHAR[(<size> [BYTE|CHAR])]` — строка символов фиксированной длины (от 1 до 2000 символов, но не более 2000 байт). Параметры имеют тот же смысл, что у `VARCHAR2`. При использовании без параметров длина строки равна 1.

NCHAR[(<size>)] — строка Unicode-символов фиксированной длины (до 2000 символов, но не более 2000 байт). По умолчанию — 1 символ. Используется только символьная семантика.

LONG — строка символов переменной длины (до 2 Гб). Устаревший формат, оставленный для обратной совместимости со старыми версиями СУБД.

CLOB — Character Large Object, большой символьный объект. Содержит одно- или многобайтные символы, допускает наборы символов с равномерным и неравномерным кодированием. Максимальный размер — (2^32 − 1) * DB_BLOCK_SIZE. DB_BLOCK_SIZE — это системный параметр (размер блока базы данных), определяемый настройками СУБД, значения от 2048 до 32768 байт (по умолчанию — 8192 байт).

NCLOB — вариант CLOB для Unicode-символов.

Числовые форматы

NUMBER[(<p>[, <s>])] — число с точностью <p> (от 1 до 38, по умолчанию — 38) и шкалой <s> (от –84 до 127, по умолчанию — 0). Точность задаёт количество десятичных цифр в числе, шкала — количество знаков до или после запятой. Рассматривается как основной числовой формат Oracle.

Примеры использования:

- NUMBER — целое 38-разрядное десятичное число;
- NUMBER(4) — целое 4-разрядное десятичное число;
- NUMBER(5,2) — десятичное число вида xxx.xx;
- NUMBER(2,3) — десятичное число вида 0.0xx;
- NUMBER(2,-1) — десятичное число вида xx0.

FLOAT[(<p>)] — число с плавающей точкой точности <p> двоичных цифр (от 1 до 126, по умолчанию — 53). В БД представляется как NUMBER.

INTEGER — стандартное целое число в диапазоне $[-2^{31}, 2^{31}-1]$.

BINARY_FLOAT — 32-битное число с плавающей точкой.

BINARY_DOUBLE — 64-битное число с плавающей точкой.

Форматы даты и времени

DATE — дата и время в диапазоне от 01.01.4712 до н. э. до 31.12.9999 н. э. Использует 7 байт. Содержит поля YEAR, MONTH, DAY, HOUR, MINUTE, SECOND.

TIMESTAMP[(<fsp>)] — дата и время с указанием долей секунды, точность определяется параметром (от 0 до 9 знаков после

запятой, по умолчанию — 6). Использует от 7 до 11 байт в зависимости от точности.

`TIMESTAMP[(<fsp>)] WITH TIME ZONE` — однобайтовые поля `TIMEZONE_HOUR` и `TIMEZONE_MINUTE` определяют временную зону.

`TIMESTAMP[(<fsp>)] WITH LOCAL TIME ZONE` — отличается от предыдущего формата тем, что при записи в БД дата и время преобразуются к часовому поясу СУБД, при запросе — к часовому поясу пользовательской сессии.

`INTERVAL YEAR [(<yp>)] TO MONTH` — интервал в годах и месяцах; точность представления года — от 0 до 9 цифр (по умолчанию — 2). Использует 5 байт.

`INTERVAL DAY [(<dp>)] TO SECOND[(<fsp>)]` — интервал в днях, часах, минутах и секундах; точность представления дня — от 0 до 9 цифр (по умолчанию — 2), точность представления секунд — от 0 до 9 знаков после запятой (по умолчанию — 6). Использует 11 байт.

Другие форматы

`RAW(<size>)` — произвольные бинарные данные длины `<size>` (до 2000 байт).

`BLOB` — Binary Large Object, большой двойчный объект. Произвольные бинарные данные, максимальный размер — `(`$2^{32}-1$`) * DB_BLOCK_SIZE`.

`BFILE` — указатель на большой внешний файл. Позволяет организовать потоковый ввод-вывод для внешних LOB, хранящихся на сервере БД. Максимальный размер — 4 Гб.

`LONG RAW` — произвольные бинарные данные переменной длины (до 2 Гб).

`ROWID` — формат уникального адреса строки в таблице. Используется в псевдостолбце `ROWID`.

`UROWID[(<size>)]` — формат логического адреса в индексируемой таблице. Используется в псевдостолбце `UROWID`. Максимальный размер — 4000 байт (он же используется по умолчанию).

2.2. Создание таблиц с помощью оператора CREATE TABLE

Создание таблиц осуществляется с помощью оператора `CREATE TABLE`. Базовый синтаксис оператора:

```
CREATE TABLE [<schema>.]<table_name>
(<column1> <data_type> [DEFAULT <expr1>]
                       [column1_constraint]
 [, <column2> ...]
```

```
  [, <table_constraints>]
);
```

С помощью ключевого слова DEFAULT для каждого столбца можно задать значение по умолчанию. Ограничения, относящиеся к одному столбцу, можно задать либо сразу после определения столбца, либо в конце определения таблицы. Ограничения, относящиеся к нескольким столбцам, задаются только в конце определения таблицы.

В Oracle существует 5 типов ограничений на столбцы:

- NOT NULL — не допускается неопределённое значение атрибута;
- UNIQUE — значения атрибута (атрибутов) должны быть уникальны;
- PRIMARY KEY — первичный ключ (простой или составной);
- FOREIGN KEY — внешний ключ;
- CHECK — проверка значений атрибута (атрибутов) на соответствие условию.

Общий синтаксис определения ограничения:

```
[CONSTRAINT <c_name>] <c_type> [<c_body>]
```

Ограничениям можно (и рекомендуется) присваивать осмысленные имена, но, если этого не сделать, Oracle автоматически присвоит ограничению стандартизированное имя. Тело ограничения не задаётся для ограничений NOT NULL, а также для первичных ключей и ограничений уникальности, заданных при определении атрибута.

Для первичного ключа и ограничения уникальности тело ограничения задаётся списком атрибутов:

```
(<attr1>[, <attr2> ...])
```

Внешний ключ, задаваемый при определении атрибута, записывается в виде:

```
REFERENCES <table> (<ref_attr>)
                              [ON DELETE <option>]
```

Внешний ключ, задаваемый в конце определения таблицы, записывается в виде:

```
FOREIGN KEY (<attr_list>) REFERENCES <table>
        (<ref_attr_list>) [ON DELETE <option>]
```

В обоих случаях необходимо указать, на какие атрибуты какой таблицы будет ссылаться внешний ключ. Как обычно, внешние ключи создаются для связей «один-к-одному» и «один-ко-многим», а связь «многие-ко-многим» преобразуется к двум связям «один-ко-многим» за счёт использования промежуточной таблицы.

Опция ON DELETE определяет порядок действий при удалении элемента, на который идёт ссылка по внешнему ключу. Если указано ON DELETE SET NULL, то соответствующие атрибуты кортежей дочерней таблицы будут очищаться при удалении кортежа родительской таблицы («мягкий» вариант, в большинстве случаев является наиболее предпочтительным). Если же указано ON DELETE CASCADE, то будет произведено удаление кортежей дочерней таблицы при удалении кортежа родительской таблицы («жёсткий» вариант). При развитой иерархии таблиц удаление будет происходить каскадно. Если опция не указана, используется вариант по умолчанию ON DELETE RESTRICT (ограничить удаление при конфликте ключей).

Рассмотрим пример создания таблицы Student:

```
CREATE TABLE Student
(
  id NUMBER(8) PRIMARY KEY,
  name VARCHAR2(100) NOT NULL,
  birthday DATE NOT NULL,
  s_pass NUMBER(4, 0) NOT NULL
                      CHECK (s_pass >= 1000),
  n_pass NUMBER(6, 0) NOT NULL
                      CHECK (n_pass >= 100000),
  agrant NUMBER(8, 2)
     CHECK (agrant IS NULL OR agrant >= 0),
  group_name VARCHAR2(10) REFERENCES
           AGroup (name) ON DELETE SET NULL,
  gender VARCHAR2(1)
                  CHECK (gender IN ('M', 'Ж')),
  CONSTRAINT passport_un
                      UNIQUE (s_pass, n_pass)
);
```

При создании таблицы мы прописали все естественные ограничения — требования наличия данных в ряде полей (имя, день рождения и т. п.), неотрицательности стипендии (но возможно пустое значение), ограничение на вводимые данные в поле gender и т. д. В качестве первичного ключа выбран id (номер студенческого билета), но при этом в базе имеется потенциальный ключ — серия и номер паспорта, на которые мы налагаем ограничение уникальности. Единственный внешний ключ group_name ссылается на первичный ключ таблицы AGroup. При этом мы устанавливаем опцию ON DELETE SET NULL, поскольку при удалении группы нельзя удалять студентов этой группы, нужно лишь очистить у каждого поле group_name.

В другом примере:

```
CREATE TABLE Group_Disc
(
  group_name VARCHAR2(10),
  disc_cipher NUMBER(6),
  CONSTRAINT group_disc_pk
     PRIMARY KEY (group_name, disc_cipher),
  FOREIGN KEY (group_name) REFERENCES
            AGroup (name) ON DELETE CASCADE,
  CONSTRAINT disc_fk
     FOREIGN KEY (disc_cipher) REFERENCES
     Discipline (cipher) ON DELETE CASCADE
);
```

мы устанавливаем все ограничения в конце определения таблицы, для иллюстративности чередуя именованные и неименованные ограничения. Поскольку таблица `Group_Disc` служит для разрешения связи «многие-ко-многим», в обоих внешних ключах мы устанавливаем опцию `ON DELETE CASCADE` — при удалении группы или дисциплины нужно исключить соответствующие связи.

Удаление таблицы осуществляется с помощью команды

```
DROP TABLE <table_name>;
```

Начиная с Oracle версии 10g, таблицы при удалении помещаются в корзину, откуда могут быть восстановлены.

2.3. Создание других объектов схемы

Рассмотрим основные, наиболее часто используемые объекты схемы и правила их определения с помощью операторов DDL.

Представление

Преставление (`VIEW`) — это фактически хранимый запрос типа `SELECT`, используя который можно осуществлять быстрый доступ к требуемым данным. Представление относительно независимо по отношению к таблицам БД. Существует два типа представлений. *Простое представление* отражает содержимое некоторой таблицы базы данных; через это представление можно осуществлять DML-запросы к таблице (если не выбрана опция «только для чтения»). Составное представление обращается к нескольким таблицам либо использует агрегацию данных; DML-операции с таблицами через это представление возможны только с той таблицей, которая определяет ключевые атрибуты представления, и только в том случае, когда в представлении не используется агрегация данных.

Представление создаётся с помощью оператора CREATE VIEW. Базовый синтаксис оператора:

```
CREATE [OR REPLACE] [FORCE|NOFORCE]
        VIEW [<schema>.]<view_name> AS
<SELECT statement>
[WITH {READ ONLY|
       CHECK OPTION [CONSTRAINT <c_name>]}];
```

Опция OR REPLACE предписывает Oracle заменить представление, если представление с таким названием уже существует. Опция FORCE предписывает Oracle создать представление, даже если запрос некорректен. Опция WITH READ ONLY запрещает выполнение DML-запросов к простому представлению, а опция WITH CHECK OPTION проверяет результат DML-запроса на соответствие условиям основного запроса. Как и ограничения в таблицах, это ограничение можно именовать.

Простое представление «Магистрант» будет содержать информацию о всех студентах магистратуры:

```
CREATE OR REPLACE VIEW Magister AS
SELECT * FROM Student
WHERE group_name LIKE '%МО';
```

К представлению можно осуществлять DML-запросы, например:

```
UPDATE Magister SET group_name = 'ИТ-11БО'
WHERE name LIKE 'Прохоров%';
```

Такой запрос переведёт всех магистрантов Прохоровых на первый курс бакалавриата, что является неверным. Поэтому запрос на создание представления стоит снабдить требованием проверки условия при DML-запросах:

```
CREATE OR REPLACE VIEW Magister AS
SELECT * FROM Student
WHERE group_name LIKE '%МО'
WITH CHECK OPTION;
```

Пример составного представления, выводящего список студентов с указанием названия группы и номера курса:

```
CREATE OR REPLACE VIEW Stud_Course AS
SELECT s.id, s.name, s.group_name, g.course
FROM Student s, AGroup g
WHERE s.group_name = g.name WITH READ ONLY;
```

Удаление представления осуществляется с помощью команды

```
DROP VIEW <view_name>;
```

Последовательность номеров

Объект SEQUENCE позволяет генерировать номера (целые числа) в соответствии с некоторой *последовательностью*. Базовый синтаксис:

```
CREATE SEQUENCE <seq_name> [<options>];
```

Рассмотрим опции, которые могут использоваться при задании последовательности:

- START WITH <n> — начальный элемент последовательности (по умолчанию — 1);
- INCREMENT BY <i> — шаг последовательности (положительный или отрицательный, по умолчанию — 1);
- MAXVALUE <m>|NOMAXVALUE — максимальное значение (от $-(10^{27}-1)$ до $10^{28}-1$, но не меньше начального значения); вариант по умолчанию NOMAXVALUE говорит о том, что максимальное значение не устанавливается (фактически оно равно максимально возможному);
- MINVALUE <m>|NOMINVALUE — минимальное значение (не больше начального и меньше максимального значения); вариант по умолчанию — NOMINVALUE (фактически минимальное значение равно начальному);
- CYCLE|NOCYCLE — циклическая генерация при достижении максимума (переход к минимуму); при выборе ациклической последовательности после достижения максимума попытка генерации нового значения приведёт к ошибке;
- CACHE <n>|NOCACHE — опция определяет, какое количество значений должно быть кэшировано для быстрого доступа к ним (от 2 до $10^{28}-1$, по умолчанию — 20); в случае ошибки в транзакции и отката кэшированные значения теряются;
- ORDER|NOORDER — опция определяет, будут значения выдаваться в порядке генерации или случайным образом из множества кэшированных значений.

Пример последовательности, генерирующей номера студенческих билетов:

```
CREATE SEQUENCE Stud_Num
START WITH 100000 INCREMENT BY 1
NOCACHE NOCYCLE;
```

Для обращения к элементам последовательности используются две функции: CURRVAL выдаёт текущее значение, NEXTVAL осуществляет переход к следующему значению и выдаёт его. Примеры использования:

```
SELECT Stud_Num.CURRVAL FROM DUAL;
SELECT Stud_Num.NEXTVAL FROM DUAL;
```

Удаление последовательности осуществляется с помощью команды

```
DROP SEQUENCE <seq_name>;
```

Индекс

Для ускорения выполнения запросов можно использовать *индексы*. Индексы создаются автоматически для ограничений PRIMARY KEY и UNIQUE, но можно их создавать и вручную. Базовый синтаксис:

```
CREATE [UNIQUE|BITMAP] INDEX <index_name>
ON [<schema>.]<table_name>
                    (<attr1>[, <attr2> ...]);
```

Опция UNIQUE позволяет создавать уникальный индекс (значения индексируемых атрибутов должны быть уникальны); опция BITMAP позволяет создать индекс, использующий для одинаковых значений ключа одну битовую карту вместо отдельных значений индекса для каждой записи. В битовой карте для каждого значения ключа указываются все ROWID записей с таким ключом.

Удаление ключа осуществляется с помощью команды

```
DROP INDEX <index_name>;
```

Синоним

Синонимы используются для задания альтернативного имени объекта или его части. Обычно они используются для присвоения краткого имени объекту с длинным именем и/или путём до объекта. Базовый синтаксис:

```
CREATE [OR REPLACE] [PUBLIC]
        SYNONYM <syn_name> FOR <object_name>;
```

Опция PUBLIC делает синоним доступным для других пользователей, имеющих право просматривать данную схему.

Удаление синонима осуществляется с помощью команды

```
DROP SYNONYM <syn_name>;
```

2.4. Управление объектами схемы

Структуру объектов схемы можно менять, используя оператор ALTER в различных его вариантах.

Для изменения структуры таблицы используется оператор ALTER TABLE. Синтаксис оператора во многом похож на синтаксис CREATE TABLE. При добавлении атрибутов и/или ограничений этот оператор нужно использовать в сочетании с оператором ADD. Запрос

```
ALTER TABLE Student ADD
(
```

```
    entrance_date DATE DEFAULT SYSDATE,
    city VARCHAR2(30),
    FOREIGN KEY (city) REFERENCES Cities (name)
                       ON DELETE SET NULL
);
```

добавляет в таблицу Student два атрибута и внешний ключ. При добавлении ограничений можно указать опции ENABLE|DISABLE (включено/выключено) и VALIDATE|NOVALIDATE (проверять ли уже существующую информацию на соответствие ограничению):

```
ALTER TABLE AGroup ADD CONSTRAINT mag_chk
      CHECK (mag IN ('Да','Нет'))
      DISABLE NOVALIDATE;
```

С помощью указанных опций можно включать и выключать существующие ограничения:

```
ALTER TABLE <table_name>
[ENABLE|DISABLE] [VALIDATE|NOVALIDATE]
{PRIMARY KEY|CONSTRAINT <cons_name>};
```

Для ограничения можно также задать свойство IMMEDIATE или DEFERRED. В первом случае ограничение будет проверяться сразу при выполнении операции DML, во втором — по окончании транзакции:

```
SET CONSTRAINT[S] {<cons1>[, <cons2> ...]| ALL}
                  IMMEDIATE|DEFERRED;
```

Для того чтобы данная операция была допустима, необходимо, чтобы при создании ограничения оно было объявлено с опцией DEFERRABLE.

Удаление столбцов и ограничений осуществляется с помощью оператора DROP:

```
ALTER TABLE Student DROP (entrance_date);
ALTER TABLE Student DROP COLUMN city;
ALTER TABLE AGroup DROP CONSTRAINT mag_chk;
```

При удалении атрибута удаляются и все связанные с ним ограничения. При удалении ограничения можно указать опцию CASCADE, тогда удалятся и ограничения, связанные с данным. Удаление столбца происходит немедленно. Если это недопустимо, можно столбец временно определить как неиспользуемый, с тем чтобы удалить позже:

```
ALTER TABLE Student SET UNUSED (city);
<...>
ALTER TABLE Student DROP UNUSED COLUMNS;
```

Модификация таблицы осуществляется с помощью оператора MODIFY, синтаксис которого похож на синтаксис оператора ADD:

```
ALTER TABLE Student
MODIFY (agrant NUMBER(9,2));
```

Таблицы, в которых уже есть кортежи, не могут быть изменены в сторону уменьшения размерности или точности данных в столбцах.

Оператор FLASHBACK TABLE используется для возврата к одной из предыдущих версий таблицы, а также для восстановления таблицы из корзины. Восстановление таблицы:

```
FLASHBACK TABLE Cities TO BEFORE DROP;
```

Возврат к предыдущему состоянию с использованием временного интервала:

```
FLASHBACK TABLE Cities TO TIMESTAMP
(SYSTIMESTAMP - INTERVAL '5' MINUTE);
```

2.5. Словарь данных

В *словаре данных* (Data Dictionary) хранятся все метаданные базы данных в виде некоторых таблиц, работа с которыми осуществляется через *представления* (Data Dictionary Views). Все представления делятся на 3 класса:

- ALL_ — информация о всех объектах, к которым возможен доступ данного пользователя (в том числе тех, которые ему принадлежат);
- USER_ — информация об объектах схемы данного пользователя;
- DBA_ — информация о всех объектах БД (доступна только администраторам БД).

Примеры (с полным списком представлений можно ознакомиться в [1]):

- USER_OBJECTS — все объекты пользователя;
- ALL_OBJECTS — все объекты, доступные пользователю;
- USER_TABLES — таблицы пользователя;
- USER_TAB_COLUMNS — столбцы таблиц, принадлежащие пользователю;
- USER_CONSTRAINTS — ограничения, принадлежащие пользователю;
- USER_CONS_COLUMNS — столбцы соответствующих таблиц;
- DBA_INDEXES — все индексы, существующие в БД;

- DBA_IND_COLUMNS — столбцы, соответствующие этим индексам.

Работать с Data Dictionary Views можно точно так же, как и с обычными представлениями, например:

```
SELECT * FROM USER_OBJECTS;
```

2.6. Упражнения

1. Напишите запросы на создание таблиц AGroup и Discipline, учитывающие все естественные ограничения.

2. Напишите запрос на создание простого представления, содержащего информацию о всех группах первого курса бакалавриата; представление должно проверять ограничения, задаваемые запросом.

3. Напишите запрос на создание составного представления, содержащего информацию о всех бакалаврах-первокурсниках; представление не должно дозволять DML-операции.

4. Напишите запрос на создание последовательности уникальных шифров дисциплин. Каждый шифр должен быть пяти- или шестизначным, интервал между шифрами равен 3.

5. Напишите запрос, добавляющий в таблицу Discipline атрибут tutor_id со ссылкой по внешнему ключу на первичный ключ таблицы Tutor.

6. Напишите запрос к словарю данных, выводящий названия всех таблиц, к которым вы имеете доступ, но которые вам не принадлежат.

3. Процедурное расширение Oracle SQL — язык PL/SQL

В данной главе мы рассмотрим основные конструкции процедурного расширения Oracle SQL — языка PL/SQL. Язык позволяет организовывать работу с базой данных как с помощью отдельных блоков команд, так и посредством процедур, функций и пакетов. Подробное руководство по командам языка можно найти в [1], [2].

3.1. Простые конструкции PL/SQL

Константы объявляются следующим образом:

```
<cons_name> CONSTANT <data_type> := <value>;
```

В отличие от констант, переменные могут быть инициализированы отдельно от объявления, если это не NOT NULL-переменные. Объявление переменной (два варианта):

```
<var_name> <data_type> [:= <value>];
<var_name> <data_type> NOT NULL := <value>;
```

В PL/SQL переменные допускают как непосредственное присваивание:

```
<var_name> := <value>;
```

так и присваивание из запроса (в этом случае результат запроса должен в точности соответствовать переменной по типу данных и размерности):

```
SELECT <expr> INTO <var_name>
FROM <table_list> WHERE <conditions>;
```

В качестве типа данных переменной можно использовать любой SQL-тип, равно как и специфические типы PL/SQL, например, BINARY_INTEGER и PLS_INTEGER (32-битные целые числа со знаком), NATURAL (PLS_INTEGER со значениями, большими или равными 0), NATURALN (NATURAL со встроенным ограничением NOT NULL), POSITIVE (PLS_INTEGER со значениями, большими 0), POSITIVEN (POSITIVE со встроенным ограничением NOT NULL), SIGNTYPE (PLS_INTEGER со значениями знакового типа: -1, 0 и 1), SIMPLE_INTEGER (PLS_INTEGER со встроенным ограничением NOT NULL) и другие.

В PL/SQL имеется возможность определять собственные типы данных на базе стандартных. Для этого используются конструкции на основе оператора SUBTYPE (подтип).

1. Синонимичные типы:

```
SUBTYPE <subtype_name> IS <base_type_name>;
```

Пример:

```
SUBTYPE item_counter IS NATURAL;
```

2. Базовые типы данных не содержат в определении подтипа ограничение длины. Если требуется это сделать, нужно использовать промежуточную переменную:

```
<temp_var> <base_type_name> (<max_length>);
SUBTYPE <subtype_name> IS <temp_var>%TYPE;
```

Предикат `%TYPE` возвращает тип переменной вместе с ограничениями точности и длины. Пример:

```
temp VARCHAR2(30);
SUBTYPE surname IS temp%TYPE;
```

3. Можно также определять подтип как тип записи (`RECORD`):

```
SUBTYPE <subtype_name> IS <object>%ROWTYPE;
```

Предикат `%ROWTYPE` возвращает тип строки таблицы или выборки (курсора). Пример:

```
SUBTYPE StudRec IS Student%ROWTYPE;
```

В главах 4, 6 мы рассмотрим вопросы, связанные с созданием более сложных типов (коллекций и объектов).

К переменным применимы стандартные операции (как арифметические, так и SQL-операции). Их приоритеты:

1. `NOT`.
2. `+`, `-` (унарные).
3. `*`, `/`.
4. `+`, `-` (бинарные), `||`.
5. `=`, `!=` (`<>`), `<`, `>`, `<=`, `>=`, `IS NULL`, `LIKE`, `BETWEEN`, `IN`.
6. `AND`.
7. `OR`.

Результат сравнения с `NULL` всегда `NULL`; применение `NOT` к значению `NULL` всегда `NULL`; `NULL` условие всегда `FALSE`.

В PL/SQL применимы все простые функции, рассмотренные в разделе 1.2.

Отдельные команды PL/SQL можно объединять в *анонимные блоки*. Базовый синтаксис:

```
[<<label>>]
[DECLARE <local declarations>]
BEGIN
   <operators>
[EXCEPTION <handle>]
END [label];
```

Все переменные должны быть объявлены в субблоке `DECLARE` и инициализированы до первого использования.

Комментирование кода осуществляется с помощью конструкций `-- комментарий` (однострочный комментарий) и `/* комментарий */` (многострочный комментарий).

3.2. Операторы управления

К *операторам управления* относят различные операторы переходов, ветвлений и циклов. Рассмотрим основные синтаксические конструкции.

Бесконечный цикл задаётся с помощью конструкции

```
[<<label>>] LOOP
    <operators>
END LOOP [label];
```

Выход из бесконечного цикла происходит по условию `EXIT WHEN <condition>`. Метку (`<<label>>`) можно присвоить любому оператору и впоследствии переходить к нему по этой метке. В дальнейшем мы не будем показывать метки в синтаксических конструкциях.

Цикл `WHILE` задаётся с помощью конструкции

```
WHILE <condition> LOOP
    <operators>
END LOOP;
```

Стандартная форма цикла `FOR` задаётся с помощью конструкции

```
FOR <param> IN [REVERSE] <min>..<max> LOOP
    <operators>
END LOOP;
```

Отметим, что параметр цикла можно не объявлять заранее — в этом случае он будет создан и инициализирован автоматически, область видимости — тело цикла. Опция `REVERSE` позволяет запустить счётчик цикла в обратном направлении. При этом `<min>` должно быть меньше или равно `<max>` даже в случае убывающей последовательности, иначе тело цикла не выполнится ни разу. Шаг цикла — всегда 1. Начиная с Oracle версии 11g, доступны операторы безусловного (`CONTINUE`) и условного (`CONTINUE WHEN <condition>`) перехода к следующему витку цикла.

Безусловный переход к некоторому оператору осуществляется по его метке с помощью команды `GOTO label`.

Условный оператор в Oracle может использоваться в одном из трёх вариантов. Первый вариант:

```
IF <condition> THEN
    <operators>
END IF;
```

Второй вариант:

```
IF <condition> THEN
    <operators>
ELSE
    <operators>
END IF;
```

Третий вариант:

```
IF <condition1> THEN
    <operators>
ELSIF <condition2> THEN
    <operators>
[ELSIF ...]
ELSE
    <operators>
END IF;
```

Отметим, что условия во всех операторах управления можно не заключать в скобки, при этом можно комбинировать несколько условий с помощью логических операторов NOT, AND и OR.

3.3. Пакеты, процедуры и функции PL/SQL

Пакет — это объект, который объединяет логически завершённые типы PL/SQL, элементы и подпрограммы. Состоит из двух частей: спецификации пакета и тела пакета. *Спецификация* — это интерфейсная public-часть пакета, её видят все пользователи, имеющие привилегию EXECUTE для данной схемы. *Тело* — это закрытая private-часть, скрытая от пользователей. Так, процедура или функция, реализованная в теле пакета, но не объявленная в спецификации, будет трактоваться как *утилита*, то есть её область видимости — только тело пакета. Фактически разделение пакета на спецификацию и тело реализует принцип *инкапсуляции*.

Синтаксис спецификации пакета:

```
CREATE [OR REPLACE]
                PACKAGE <package_name> AS|IS
    <public declarations>
    <specification of procedures and functions>
END [<package_name>];
```

Синтаксис тела пакета:

```
CREATE [OR REPLACE]
            PACKAGE BODY <package_name> AS|IS
    <private declarations>
    <bodies of procedures and functions>
END [<package_name>];
```

Основными элементами пакета являются *процедуры* и *функции* (у функции есть возвращаемое значение, у процедуры — нет). Спецификация (объявление) процедур и функций, являющихся частью пакета:

```
PROCEDURE <name> [(par1 [, par2 ...])];
```

```
FUNCTION <name> [(par1 [, par2 ...])]
                                RETURN <data_type>;
```

Реализация (тело) процедур и функций, являющихся частью пакета:

```
PROCEDURE <name> [(par1 [, par2 ...])] IS
   [<local declarations>]
BEGIN
   <operators>
[EXCEPTION <handle>]
END [<name>];

FUNCTION <name> [(par1 [, par2 ...])]
                            RETURN <data_type> IS
   [<local declarations>]
BEGIN
   <operators>
[EXCEPTION <handle>]
END [<name>];
```

Выход из процедур и функций осуществляется по команде `RETURN [<value>]`. Локальные объявления в процедурах и функциях разделяются точкой с запятой. Отметим, что если процедура или функция создаётся как *хранимая процедура* или *хранимая функция*, то реализуется сразу тело объекта, предваряемое командой `CREATE [OR REPLACE]`. Однако в хранимых процедурах и функциях не будет инкапсуляции.

Параметры процедур и функций, равно как и возвращаемое значение функции, указываются без ограничения точности и/или длины в типе данных (то есть можно написать `VARCHAR2`, но нельзя — `VARCHAR2(30)`).

Параметры процедур и функций объявляются следующим образом:

```
<param_name> [IN|OUT|IN OUT] <data_type>
                                [{:=|DEFAULT} <expr>]
```

Их можно объявлять как входные, выходные и одновременно входные и выходные (по умолчанию используется опция `IN`). Также для параметра можно определить значение по умолчанию.

При вызове процедур и функций можно использовать *позиционную* (позиция значения определяет, какому параметру оно будет присвоено), *именованную* (в явном виде указывается имя параметра, позиция значения не имеет) и *смешанную* нотацию.

Пусть имеется некая функция `f(x, y, z)`.

Позиционная нотация — `f(1, 2, 3)`.

Именованная нотация — `f(y => 2, x => 1, z => 3)`.

Смешанная нотация — `f(y => 2, x => 1, 3)`.

3.4. Курсоры

Курсор — это результирующий набор, полученный с помощью запроса на выборку, и связанный с этим набором указатель текущей записи. С помощью курсоров в PL/SQL удобно организовывать последовательную обработку данных, находящихся в таблицах БД.

Объявление курсора:

```
CURSOR <cursor_name> [(par1[, par2 ...])]
[RETURN <data_type>] IS <SELECT statement>;
```

Опция `RETURN <data_type>` определяет тип данных (формат представления) для записи, возвращаемой оператором `FETCH`. По умолчанию используется тип записи (`RECORD`), совпадающей по структуре со списком выборки запроса `SELECT`.

Объявление параметра курсора:

```
<param_name> [IN] <data_type>
                              [{:=|DEFAULT} <expr>]
```

Операторы управления курсором:

- `OPEN <cursor_name> [(<par_values>)]` — связывает курсор с данными, передаёт параметры;
- `FETCH <cursor_name>` — извлекает очередную строку из набора;
- `CLOSE <cursor_name>` — закрывает курсор, разорвав связь между курсором и выборкой. После закрытия курсор можно связать с новой выборкой.

Перед выполнением команды `FETCH` курсор обязательно должен быть открыт, при этом нельзя открывать уже открытый и ещё не закрытый курсор.

Для курсора определены 4 атрибута, которые можно использовать в программе:

- `%ISOPEN` — `TRUE`, если курсор открыт;
- `%FOUND` — `TRUE`, если найдена очередная строка (не достигнут конец выборки);
- `%NOTFOUND` — `TRUE`, если очередная строка не найдена;
- `%ROWCOUNT` — номер текущей записи.

Рассмотрим пример работы с курсором:

```
DECLARE
  CURSOR stud (grp VARCHAR2,
```

```
                grnt NUMBER DEFAULT 2000) IS
    SELECT id, name, agrant FROM Student
    WHERE agrant > grnt AND group_name = grp;
  stud_rec stud%ROWTYPE;
BEGIN
  OPEN stud('ИТ-21МО', 2500);
  LOOP
    FETCH stud INTO stud_rec;
    EXIT WHEN stud%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(stud_rec.id||' '||
      stud_rec.name||' '||stud_rec.agrant);
  END LOOP;
  CLOSE stud;
END;
```

Курсор осуществляет поиск студентов указанной группы, стипендия которых больше указанного значения (по умолчанию — 2000 руб.). В блоке PL/SQL мы сначала открываем курсор с параметрами `'ИТ-21МО', 2500`, затем последовательно перебираем строки выборки, пока не дойдём до её конца, и выводим соответствующие значения с помощью метода `PUT_LINE` пакета `DBMS_OUTPUT` (консольный вывод). После этого курсор закрывается. К элементам строки курсора мы обращаемся, используя «точечный» синтаксис:

```
<cursor_rec>.<attribute_name>
```

Те же самые действия с курсором можно осуществить, используя *курсорную форму цикла* `FOR`:

```
FOR <param>
    IN {<cursor_name> [(<par_values>)]|
        <SELECT statement>}
LOOP
    <operators>
END LOOP;
```

Такой цикл `FOR` сам откроет курсор (или создаст его, если в ветке `IN` указан запрос на выборку), определит параметр цикла как запись, совпадающую по структуре со строкой курсора, пройдёт по выборке, определяемой курсором, а после завершения цикла закроет курсор.

Тогда предыдущий пример можно переписать в виде:

```
DECLARE
  CURSOR stud (grp VARCHAR2,
                grnt NUMBER DEFAULT 2000) IS
    SELECT id, name, agrant FROM Student
    WHERE agrant > grnt AND group_name = grp;
BEGIN
  FOR stud_rec IN stud('ИТ-21МО', 2500)
```

```
  LOOP
    DBMS_OUTPUT.PUT_LINE(stud_rec.id||' '||
      stud_rec.name||' '||stud_rec.agrant);
  END LOOP;
END;
```

3.5. Обработка исключений

Как и в любом другом языке программирования высокого уровня, в PL/SQL можно инициировать, отлавливать и обрабатывать *исключения*. Существует широкий набор стандартных исключений, например, ACCESS_INTO_NULL, COLLECTION_IS_NULL, VALUE_ERROR, DUP_VAL_ON_INDEX, INVALID_CURSOR, INVALID_NUMBER, NO_DATA_FOUND, ZERO_DIVIDE, CURSOR_ALREADY_OPEN (с полным списком можно ознакомиться в [1]). Пользовательские исключения должны быть объявлены как переменные с типом EXCEPTION (инициализировать их не нужно, Oracle сам присвоит им номера). Например:

```
my_exception EXCEPTION;
```

Обычно исключения объявляют в спецификации пакета либо создают отдельный пакет со спецификациями всех исключений. Обращение к исключению:

```
[<schema>.][<pck_name>.]<exception_name>;
```

Для произвольного инициирования исключения (системного или пользовательского) используется оператор RAISE. Пример инициирования исключения по условию:

```
IF my_var < 0 THEN
    RAISE my_exception;
END IF;
```

Обработка исключений происходит в блоке EXCEPTION. Базовый синтаксис:

```
EXCEPTION
    WHEN <exception_name> THEN
        <exception_handle>
    [WHEN <exception_name> ...]
    [WHEN OTHERS THEN
        <exception_handle>]
```

Если при работе программы возникло исключение, обработка которого не предусмотрена в текущей процедуре или функции, оно будет пробрасываться выше (в вызывающую процедуру/функцию), пока не будет найден подходящий обработчик либо пока не произойдёт вылет программы.

3.6. Упражнения

1. Напишите процедуру, добавляющую в таблицу `AGroup` запись о новой группе, определяемую параметрами процедуры. Процедура должна проверять вводимые значения на соответствие формату данных в таблице БД, а также должна допускать использование значения по умолчанию для поля `mag`. Кроме того, в процедуре должен быть предусмотрен блок обработки исключений.

2. Напишите процедуру, выводящую упорядоченный по группам список студентов, которым нужно сдавать указанную в параметре дисциплину. В процедуре должен быть предусмотрен блок обработки исключений.

3. Напишите функцию, выводящую среднее количество часов по каждой форме отчётности для каждой группы, факультета и университета в целом. В качестве результата возвращается среднее количество часов по университету. В функции должен быть предусмотрен блок обработки исключений.

4. Напишите процедуру, выводящую список дисциплин, преподаваемых на двух и более факультетах, с указанием количества факультетов и количества групп; список сортируется сначала по убыванию количества факультетов, а затем — количества групп. В процедуре должен быть предусмотрен блок обработки исключений.

5. Напишите процедуру, удаляющую из таблицы `AGroup` все пустые группы (нет ни одного студента). В процедуре должен быть предусмотрен блок обработки исключений.

6. Организуйте все процедуры и функции, разработанные в упражнениях 1–5, в виде пакета с разделением на спецификацию и тело пакета.

4. Коллекции

Коллекции — это линейные структуры данных PL/SQL, напоминающие массивы и списки. Коллекции можно определять как на уровне программы (в блоках PL/SQL), так и на уровне схемы (в последнем случае — с некоторыми ограничениями).

Мы рассмотрим основные возможности работы с коллекциями. Более детально данный вопрос освещён в [2].

4.1. Виды коллекций и их характеристики

В PL/SQL определены три вида коллекций.

Ассоциативный массив — это неограниченная разреженная коллекция, в которой допустимо присваивание значения любому элементу с любым индексом в пределах формата индекса. Является достаточно гибкой структурой данных, однако данный вид коллекции доступен только в PL/SQL (определить тип ассоциативного массива на уровне схемы нельзя).

Вложенная таблица — это неограниченная коллекция (допускает расширение посредством встроенной процедуры `EXTEND`), представляющая собой мультимножество (элементы неупорядочены). Вложенная таблица заполняется плотно, но после удаления элементов может стать разреженной. Если экземпляр вложенной таблицы используется в столбце таблицы БД, то данные будут храниться в отдельной вспомогательной таблице.

Массив `VARRAY` (массив переменной длины) — это ограниченная коллекция с плотным заполнением, гарантирующая сохранение порядка элементов. Коллекция может быть расширена посредством процедуры `EXTEND`, но в пределах определённого в типе коллекции максимального количества элементов. Если экземпляр массива `VARRAY` используется в столбце таблицы БД, то данные хранятся непосредственно в этой таблице. Поэтому не рекомендуется использовать массивы `VARRAY`, превышающие по объёму размер блока БД (`DB_BLOCK_SIZE`), — это приведёт к существенному снижению производительности.

Итак, каждая коллекция представляет собой набор однородных индексируемых элементов. Элементы коллекции могут быть составными, тип элемента задаётся при определении *типа коллекции*. Одному типу коллекций может соответствовать множество *экземпляров коллекций*, или просто *коллекций* (объектно-ориентированный подход). Массивы `VARRAY` индексируются положительными целыми числами (начальный индекс — 1, максимальное значение — $(2^{31} - 1)$, но не больше максимального ко-

личества элементов), вложенные таблицы индексируются строками VARCHAR2, преобразованными к числу (начальный индекс — 1), ассоциативные массивы допускают индексацию как целыми числами (в том числе отрицательными), так и строками. Тип индекса ассоциативного массива задаётся при определении типа коллекции. Все коллекции являются одномерными, но допустимо создание коллекций, элементами которых являются другие коллекции.

Плотное заполнение коллекции предполагает, что все элементы от первого до последнего определены и им присвоены значения (возможно, NULL). При *разреженном* заполнении коллекции какие-то элементы могут не существовать.

При использовании коллекций в таблицах БД под *внешней таблицей* принято понимать таблицу БД, содержащую столбец-коллекцию. Саму коллекцию тогда называют *внутренней таблицей*, а дополнительную физическую таблицу, созданную для хранения данных из коллекции типа вложенной таблицы, — *вспомогательной таблицей*.

4.2. Типы и экземпляры коллекций

Любой экземпляр коллекции создаётся на основе заранее определённого типа коллекции. Последний можно создать двумя способами: на уровне PL/SQL с помощью оператора TYPE или на уровне схемы с помощью оператора CREATE [OR REPLACE] TYPE. При определении на уровне программы тип коллекции будет доступен только в том блоке, где определён, но если поместить команду TYPE в спецификацию пакета, то тип будет доступен всем программам, имеющим привилегию EXECUTE для данного пакета. Вид коллекции определяется синтаксисом оператора определения типа.

Ассоциативный массив задаётся только на уровне программы с помощью конструкции

```
TYPE <col_type> IS TABLE OF <data_type>
    [NOT NULL] INDEX BY <index_type>;
```

Указанием на то, что у нас именно ассоциативный массив, является ветка INDEX BY (для остальных типов коллекций индексирование предопределено). Типом элемента ассоциативного массива может быть любой тип PL/SQL, кроме курсора и исключения. Опция NOT NULL говорит о том, что значение элемента не может быть неопределённым (но сам элемент может отсутствовать).

В качестве типа данных индекса можно использовать любой подтип целочисленного типа PLS_INTEGER, а также VARCHAR2

с длиной строки, не превышающей 32767. Кроме того, в определении индекса можно использовать конструкции на основе %TYPE и пользовательские типы данных, являющиеся подтипами целочисленных и строковых типов.

К примеру, конструкция

```
TYPE group_data_arr IS TABLE OF
AGroup%ROWTYPE INDEX BY AGroup.name%TYPE;
```

создаст тип ассоциативного массива, элементами которого будут записи, по структуре совпадающие с кортежом таблицы AGroup; индексация в массиве будет осуществляться по имени группы (в таблице это первичный ключ, тип данных — VARCHAR2(10)).

Вложенная таблица объявляется с помощью конструкции

```
[CREATE [OR REPLACE]] TYPE <col_type> AS|IS
TABLE OF <data_type> [NOT NULL];
```

Массив VARRAY задаётся так:

```
[CREATE [OR REPLACE]] TYPE <col_type> AS|IS
VARRAY(<max_size>) <data_type> [NOT NULL];
```

При определении типа на уровне программы нельзя использовать ключевое слово AS. При определении типа коллекции на уровне схемы можно использовать только SQL-типы (то есть конструкции с использованием %TYPE и %ROWTYPE недопустимы). Коллекции при этом являются SQL-типами, что позволяет определять многоуровневые коллекции на уровне схемы.

Удаление типа на уровне схемы происходит по команде

```
DROP TYPE <col_type> [FORCE];
```

Ранее объявленные типы можно менять с помощью команды ALTER TYPE. При этом допускаются изменения только в сторону расширения типа данных элементов и увеличения максимального количества элементов в массиве VARRAY. Если указать опцию INVALIDATE, то все зависимые объекты (экземпляры коллекций) будут объявлены недействительными. При указании опции CASCADE изменения будут распространены на все зависимые объекты, в том числе на столбцы-коллекции в таблицах БД.

Увеличение количества элементов массива VARRAY:

```
ALTER TYPE <col_type> MODIFY LIMIT <value>
                          [CASCADE|INVALIDATE];
```

Изменение типа элемента:

```
ALTER TYPE <col_type> MODIFY ELEMENT TYPE
      <new_data_type> [CASCADE|INVALIDATE];
```

На основе типа коллекции можно создавать экземпляры коллекций. Общий синтаксис таков:

```
<col_name> <col_type>
  [:= <col_type>([<elem1>[, <elem2> ...]])];
```

Вложенные таблицы и массивы VARRAY должны инициализироваться перед первым использованием, это можно сделать сра-

зу при определении экземпляра. Ассоциативные массивы нельзя инициализировать — это приведёт к ошибке компиляции.

Создадим тип списка идентификаторов студентов (номеров студенческих билетов):

```
CREATE OR REPLACE TYPE st_ids_tab
                        IS TABLE OF NUMBER(8);
```

Инициализация экземпляров коллекций может производиться с помощью конструктора с пустым списком аргументов, в итоге коллекция будет пуста (NULL):

```
DECLARE
    stud_list st_ids_tab := st_ids_tab();
BEGIN
    <...>
END;
```

В конструкторе можно указать любое конечное число параметров (но не больше максимального числа элементов в случае массива VARRAY), тип которых совпадает с типом элемента коллекции, тогда эти параметры станут элементами коллекции. Кроме того, один экземпляр коллекции можно инициализировать на основе другого экземпляра того же типа коллекции. При этом типы с разными названиями, но одинаковыми типами элементов и индексов (в случае ассоциативного массива) считаются различными. Пример указанных способов инициализации:

```
DECLARE
    stud_list1 st_ids_tab :=
            st_ids_tab(111111,111112,111123);
    stud_list2 st_ids_tab;
BEGIN
    stud_list2 := stud_list1;
    <...>
END;
```

При выборке из столбца-коллекции некоторой таблицы БД в переменную-коллекцию будет произведена автоматическая инициализация этой переменной.

4.3. Встроенные методы коллекций

В PL/SQL для коллекций предусмотрен ряд встроенных методов, при вызове которых используется стандартный «точечный» синтаксис:

```
<col_name>.<method_name>[(param_list)]
```

Обратите внимание, что все методы применяются только к экземплярам коллекций, не к типам.

EXTEND[(<n>[, <i>])] — используется при работе с вложенными таблицами и массивами VARRAY. Метод выделяет память для добавления новых элементов в коллекции указанных видов. При запуске без параметров добавляется один NULL-элемент, при запуске с одним параметром — <n> NULL-элементов, при запуске с двумя параметрами — <n> элементов, которым присваивается значение уже существующего элемента с номером <i>. Последний вариант обязательно используется, когда тип коллекции объявлен с опцией NOT NULL.

DELETE[(<i>[, <j>])] — в зависимости от параметров удаляет все элементы, либо один элемент с указанным номером, либо все элементы с номерами из заданного диапазона. Стоит отметить, что при удалении элементов стирается только значение, фактического освобождения памяти не происходит (можно записать на освободившиеся места новые значения). Память освобождается только тогда, когда возможно за счёт удалённых элементов освободить целую страницу памяти, а также когда метод запускается без параметров. Ввиду плотности заполнения массивов VARRAY метод DELETE для них можно запускать только без параметров.

TRIM[(<n>)] — удаляет один или <n> последних элементов коллекции; неприменим для ассоциативных массивов. Если какие-то элементы уже удалены методом DELETE, то метод TRIM удалит их повторно.

EXISTS(<i>) — проверяет наличие элемента с заданным индексом в коллекции, возвращает TRUE или FALSE. Если элемент был удалён одним из предыдущих методов, он считается отсутствующим.

COUNT — возвращает количество элементов коллекции. Коллекция, если это не ассоциативный массив, должна быть инициализирована. Удалённые элементы в подсчёте не участвуют.

LIMIT — возвращает максимальное количество элементов для массива VARRAY.

FIRST, LAST — возвращают первый и последний индекс элемента в коллекции соответственно; при строковом индексировании элементов ассоциативного массива порядок определяется используемой кодировкой.

PRIOR(<i>), NEXT(<i>) — возвращают индекс предыдущего и следующего элемента коллекции соответственно; используются «разумные» правила для крайних значений: элемент, предшествующий первому или следующий за последним, имеет номер NULL; если <i> < FIRST, то следующий элемент имеет номер FIRST; если <i> > LAST, то предшествующий элемент имеет номер LAST.

4.4. Использование коллекций

Самые простые способы заполнения коллекций данными — это *стандартное присваивание*:

```
<col_name>(<index>) := <value>;
```

и *агрегатное присваивание*:

```
<col1_name> := <col2_name>;
```

В первом случае тип присваиваемого значения должен соответствовать типу элемента коллекции, а значение индекса — типу индекса коллекции. Если `<col_name>` — это вложенная таблица или массив `VARRAY`, то элемент с нужным индексом должен быть предварительно добавлен в коллекцию с помощью метода `EXTEND`.

Во втором случае коллекции должны совпадать по типу, причём совпадение должно быть полным — типы коллекций одинакового вида с одинаковыми форматами элемента и индекса, но с разными названиями считаются различными. Если вид коллекции — вложенная таблица и `<col2_name>` заполнена неплотно (какие-то элементы были удалены с помощью метода `DELETE`), при агрегатном присваивании произойдёт плотное заполнение коллекции `<col1_name>`.

Обращение к элементам *многоуровневой коллекции* (*коллекции коллекций*) осуществляется по упорядоченному набору индексов. При этом многоуровневая коллекция должна быть объявлена с помощью вложенных типов коллекции. Пример трёхмерного ассоциативного массива:

```
DECLARE
   TYPE type1 IS TABLE OF NUMBER
                      INDEX BY PLS_INTEGER;
   TYPE type2 IS TABLE OF type1
                      INDEX BY PLS_INTEGER;
   TYPE type3 IS TABLE OF type2
                      INDEX BY VARCHAR2(10);
   test_collection type3;
BEGIN
   test_collection('First')(1)(-30) := 123;
   <...>
END;
```

Количество вложений теоретически не ограничивается, оно определяется возможностями вычислительной системы.

Коллекции могут использоваться внутри *записей* (`RECORD`) и *объектных типов* (`OBJECT`) точно так же, как и другие типы данных. Например, с помощью конструкции

```
TYPE group_inf IS RECORD
(
```

```
    group_name VARCHAR2(30),
    course NUMBER(1),
    faculty VARCHAR2(30),
    students st_ids_tab
);
```

мы создадим тип записи с информацией о некоторой группе с указанием списка номеров студенческих билетов студентов этой группы. `st_ids_tab` — это тип списка идентификаторов студентов, определённый в примере из раздела 4.2. Естественно, это определение должно находиться в области видимости настоящего определения типа записи. На основе типа записи можно определить *экземпляр записи*. При работе с полями экземпляра записи, в том числе с полями-коллекциями, используется стандартный «точечный» синтаксис:

```
<rec_instance>.<col_field>[(<index>)]
```

Рассмотрим теперь самый важный для нас вариант использования коллекций — атрибуты-коллекции в таблицах БД. Напомним, что на уровне схемы можно задавать только коллекции вида вложенной таблицы или массива `VARRAY`. Экземпляр массива `VARRAY` хранится непосредственно во внешней таблице, поэтому при её создании используется «стандартное» определение столбца:

```
CREATE TABLE [<schema>.]<table_name>
(
    <...>
    <attr_name> <type_of_VARRAY>,
    <...>
);
```

При определении внешней таблицы со столбцом типа вложенной таблицы необходимо указать вспомогательную таблицу, в которой физически будут располагаться данные из внутренней таблицы. Работа со вспомогательной таблицей возможна только через обращение к элементам внешней таблицы. При попытке обратиться ко вспомогательной таблице «напрямую» будет инициировано исключение. Синтаксис определения такой таблицы:

```
CREATE TABLE [<schema>.]<table_name>
(
    <...>
    <attr_name> <type_of_nested_table>,
    <...>
) NESTED TABLE <attr_name>
  STORE AS <additional_table_name>;
```

Отметим, что использование столбцов-коллекций отрицает понятия нормальных форм реляционных БД. Фактически здесь реализуется объектный подход к организации БД. К атрибутам-

коллекциям неприменимы традиционные ограничения реляционных БД (первичный и внешний ключ, уникальность, CHECK) за исключением ограничения NOT NULL. Если же какое-то из указанных ограничений должно так или иначе присутствовать, нужно моделировать его с помощью *триггеров*, о которых мы будем подробно говорить в главе 5.

Перепишем определение таблицы Discipline. Сделаем так, чтобы в этой таблице для каждой дисциплины указывался список групп, которым она читается. Отметим, что промежуточная таблица Group_Disc будет уже не нужна. Поскольку порядок следования групп в списке для нас не важен, будем использовать тип вложенной таблицы, определённый на уровне схемы:

```
CREATE TYPE grp_tab AS TABLE OF VARCHAR2(10);
```

Тогда определение таблицы будет выглядеть так:

```
CREATE TABLE Discipline
(
  cipher NUMBER(6) PRIMARY KEY,
  name VARCHAR2(100) NOT NULL,
  hours NUMBER(3) NOT NULL
    CHECK (hours > 0 AND hours <= 250),
  control VARCHAR2(7)
    DEFAULT 'Экзамен' NOT NULL
    CHECK (control IN ('Экзамен', 'Зачёт')),
  grp_list grp_tab
) NESTED TABLE grp_list
  STORE AS grp_list_in_disc;
```

Вставка и обновление значений атрибутов-коллекций осуществляется с помощью стандартных операторов INSERT и UPDATE, при этом в качестве значения атрибута указывается либо конкретный экземпляр коллекции, либо конструктор типа коллекции с любым, в том числе нулевым, количеством параметров. При вставке и обновлении значений происходит плотное заполнение атрибутов типа вложенной таблицы.

С использованием конструкции SELECT <...> INTO <...> мы можем считать все значения из атрибута-коллекции в локальную переменную-коллекцию, которая всегда будет заполняться плотно и которая будет инициализироваться автоматически. Поскольку вложенная таблица является мультимножеством, сохранение порядка элементов при считывании из таблицы или сохранении в таблицу не гарантируется. В массивах VARRAY порядок следования элементов сохраняется всегда.

Пример, иллюстрирующий вышесказанное:

```
DECLARE
  grp_lst grp_tab;
```

```
BEGIN
  INSERT INTO Discipline
  VALUES(123123, 'Математический анализ',
         216, 'Экзамен',
         grp_tab('ИТ-11БО', 'ИТ-12БО'));
  INSERT INTO Discipline
  VALUES(234234, 'Объектные базы данных',
         108, 'Экзамен',
         grp_tab('ИВТ-21МО','ИТ-21БО'));
  COMMIT;
  SELECT grp_list INTO grp_lst
  FROM Discipline
  WHERE cipher = 234234;
  FOR i IN 1..grp_lst.COUNT LOOP
    IF grp_lst(i) = 'ИТ-21БО' THEN
      grp_lst(i) := 'ИТ-21МО';
    ELSIF grp_lst(i) = 'ИВТ-21МО' THEN
      grp_lst.DELETE(i);
    END IF;
  END LOOP;

  UPDATE Discipline
  SET grp_list = grp_lst
  WHERE cipher = 234234;
  COMMIT;
END;
```

Здесь мы сначала помещаем в таблицу `Discipline` два кортежа, а затем выбираем тот из них, который соответствует дисциплине «Объектные базы данных», изменяем группу «ИТ-21БО» на группу «ИТ-21МО» и удаляем группу «ИВТ-21МО». После этого мы обновляем значения в таблице.

Конструкция `SELECT <...> INTO <...>` может быть использована и для выборки одной строки данных в элемент коллекции:

```
DECLARE
    grp_lst grp_tab := grp_tab();
BEGIN
    grp_lst.EXTEND;
    SELECT name INTO grp_lst(1)
    FROM AGroup WHERE head = 111134;
END;
```

Для агрегирования множественных данных из таблицы БД в коллекцию можно воспользоваться курсорным циклом:

```
DECLARE
    grp_lst grp_tab := grp_tab();
```

```
BEGIN
    FOR grp IN (SELECT * FROM AGroup
                WHERE course = 1)
    LOOP
        grp_lst.EXTEND;
        grp_lst(grp_lst.LAST) := grp.name;
    END LOOP;
END;
```

или конструкцией BULK COLLECT:

```
DECLARE
    grp_lst grp_tab;
BEGIN
    SELECT name BULK COLLECT INTO grp_lst
    FROM AGroup WHERE course = 1;
END;
```

В обоих случаях будет сформирован неупорядоченный список групп первого курса бакалавриата и магистратуры. Второй вариант при этом является более эффективным с точки зрения времени выполнения операции. Конструкция BULK COLLECT может сохранять данные в коллекции любого из трёх видов. При этом вложенные таблицы и массивы VARRAY не нужно инициализировать перед использованием BULK COLLECT.

При необходимости заполнять коллекцию данными непоследовательно рекомендуется использовать ассоциативный массив. В следующем примере будет сформирован упорядоченный по названию список групп со всеми данными о них; индексация — по названию группы:

```
DECLARE
    TYPE grp_all IS TABLE OF AGroup%ROWTYPE
                    INDEX BY VARCHAR2(10);
    grp_lst grp_all;
    ind VARCHAR2(10);
BEGIN
    FOR grp_rec IN (SELECT * FROM AGroup)
    LOOP
        grp_lst(grp_rec.name) := grp_rec;
    END LOOP;

    ind := grp_lst.FIRST;
    WHILE ind <= grp_lst.LAST LOOP
        DBMS_OUTPUT.PUT_LINE(ind || ' - ' ||
                        grp_lst(ind).specialty);
        ind := grp_lst.NEXT(ind);
    END LOOP;
END;
```

Наконец, отметим, что коллекции, с учётом области видимости их типов, можно передавать в качестве параметров процедур и функций так же, как и стандартные типы данных. Кроме того, коллекции могут выступать в качестве возвращаемого значения функции. Рассмотрим пример. Пусть у нас определён тип массива VARRAY, состоящий из каким-то образом упорядоченных целых чисел:

```
CREATE TYPE number_arr AS
                        VARRAY(100) OF INTEGER;
```

Определим функцию, которая фильтрует любой массив такого типа, оставляя только чётные значения:

```
CREATE OR REPLACE FUNCTION
       even_numbers (in_array number_arr)
                            RETURN number_arr AS
    out_array number_arr := number_arr();
BEGIN
    FOR i IN 1..in_array.LAST LOOP
       IF MOD(in_array(i), 2) = 0 THEN
          out_array.EXTEND;
          out_array(out_array.LAST) :=
                                      in_array(i);
       END IF;
    END LOOP;
    RETURN out_array;
END;
```

Пример вызова и использования результатов такой функции:

```
DECLARE
    start_array number_arr;
    result_array number_arr;
    fifth_element INTEGER;
BEGIN
    start_array := number_arr(1,3,6,8,10,12,
                        11,14,0,35,-4,2,-8,-7);
    result_array:=even_numbers(start_array);
    fifth_element :=
                     even_numbers(start_array)(5);
END;
```

4.5. Псевдофункции

Псевдофункции — это инструмент, который позволяет работать с таблицами баз данных как с коллекциями и наоборот. Псевдофункции коллекций не поддерживаются в PL/SQL — только в SQL. Естественно, эти операторы можно применять в SQL-инструкциях, присутствующих в коде PL/SQL.

Существуют четыре разновидности псевдофункций:

- `CAST` — отображает коллекцию одного типа на коллекцию другого типа (даже если это коллекции разных видов — вложенная таблица и массив `VARRAY`);
- `COLLECT` — агрегирует данные в коллекцию в SQL;
- `MULTISET` — отображает таблицу базы данных на коллекцию. С псевдофункциями `MULTISET` и `CAST` появляется возможность выборки строки из таблицы базы данных как из столбца с типом коллекции;
- `TABLE` — отображает коллекцию на таблицу базы данных; функция является обратной по отношению к `MULTISET`.

Подробное рассмотрение псевдофункций выходит за рамки данного пособия. Дополнительную информацию по псевдофункциям можно найти в [2].

4.6. Операции над мультимножествами

С вложенными таблицами можно работать как с мультимножествами, для чего в PL/SQL предусмотрен ряд функций и операций. Пусть у нас имеются экземпляры вложенных таблиц (возможно, разных типов) `x`, `y`, `y1`, ..., `yk` и некоторое выражение `e` произвольной природы. Рассмотрим основные операции над мультимножествами.

Операции, проверяющие условие и возвращающие `TRUE` или `FALSE`:

- `x = y` — проверяет равенство двух вложенных таблиц: совпадение имён их типов, мощности и всех элементов в соответствующих парах. Дубликаты `NULL`-значений считаются различными;
- `x <> y` (`x != y`) — проверяет неравенство двух таблиц;
- `x [NOT] IN (y1[, y2, ..., yn])` — проверяет вхождение вложенной таблицы `x` в указанный список таблиц (по сути, выполняется n-кратная проверка равенства);
- `x IS [NOT] A SET` — проверяет, является ли вложенная таблица `x` множеством (нет дубликатов);

- `x IS [NOT] EMPTY` — проверяет условие $x = \varnothing$ ($x \neq \varnothing$), то есть является ли вложенная таблица `x` пустым множеством (инициализирована, но не содержит ни одного элемента);
- `e [NOT] MEMBER [OF] x` — проверяет условие $e \in x$ ($e \notin x$), то есть является ли выражение `e` элементом вложенной таблицы `x`;
- `x [NOT] SUBMULTISET [OF] y` — проверяет условие $x \subseteq y$ ($x \not\subseteq y$), то есть является ли мультимножество `x` подмножеством `y` (дубликаты элементов считаются различными).

В последних двух функциях ключевое слово `OF` опционально — с ним и без него функции работают одинаково.

Функции преобразования вложенных таблиц (если участвует несколько экземпляров коллекций, они должны относиться к одному типу коллекции):

- `SET(x)` — преобразует мультимножество `x` в множество;
- `x MULTISET UNION [DISTINCT] y` — строит объединение мультимножеств ($x \cup y$);
- `x MULTISET INTERSECT [DISTINCT] y` — строит пересечение мультимножеств ($x \cap y$);
- `x MULTISET EXCEPT [DISTINCT] y` — строит разность мультимножеств ($x \setminus y$).

Результатом применения последних четырёх операций является вложенная таблица того же типа. Если в последних трёх функциях использовать опцию `DISTINCT`, будут удалены дубликаты (в том числе дубликаты `NULL`-значений), то есть к результату будет фактически применена функция `SET`.

4.7. Упражнения

1. Напишите команды для создания типов коллекций:

- список с данными о численности населения стран, индексируемый названием страны;
- список книг, где запись о книге соответствует структуре кортежа таблицы `Book`;

- список треков музыкального альбома;
- список треков множества музыкальных альбомов.

2. Перепишите определение таблицы `AGroup` так, чтобы список студентов и список шифров преподаваемых каждой группе дисциплин хранились в виде атрибутов-коллекций.

3. Для новой реализации таблицы `AGroup` напишите процедуру, удаляющую дубликаты из списков студентов и дисциплин, а также исключающую из этих списков записи, для которых не найдено соответствие в таблицах студентов и дисциплин.

4. Перепишите процедуру из предыдущего упражения таким образом, чтобы каждая проверка осуществлялась с помощью операций над мультимножествами.

5. Напишите процедуру, устанавливающую значение `NULL` в поле `head` для всех групп, для которых указанный в этом поле студент не является студентом данной группы. В процедуре нельзя выполнять запросы к таблице `Student`.

5. Триггеры

Триггеры — это именованные программные блоки, выполняемые в ответ на происходящие в базе данных события.

К *событиям*, на которые могут реагировать триггеры, относятся команды DML (`INSERT`, `UPDATE`, `DELETE`), команды DDL (создание, изменение, удаление объектов), события базы данных (запуск, остановка, подключение/отключение сервера и т. п.) и приостановленные команды (команды, которые были отложены из-за проблем доступности пространства в базе данных). Кроме того, для представлений (`VIEW`) возможно создание замещающих триггеров `INSTEAD OF`, которые могут выполняться вместо соответствующих операций DML, а также могут преобразовывать представление.

В рамках данного пособия мы рассмотрим только *триггеры DML* — наиболее часто использующийся вид триггеров. Остальные виды триггеров подробно освещены в книге [2].

5.1. Триггеры DML

Рассмотрим базовый синтаксис создания триггера DML:

```
CREATE [OR REPLACE]
           TRIGGER [<schema>.]<trigger_name>
{BEFORE|AFTER}
{INSERT|DELETE|UPDATE|UPDATE OF <attr_list>}
[OR {...} ...] ON [<schema>.]<table_name>
[REFERENCING OLD AS <name1> NEW AS <name2>]
[FOR EACH ROW]
[WHEN (<condition>)]
[DECLARE
   <local_declarations>]
BEGIN
   <operators>
[EXCEPTION
   <exception_handle>]
END [trigger_name];
```

Триггеры DML всегда относятся только к одной таблице БД, указанной в ветке `ON`.

По времени исполнения триггеры DML делятся на два класса: триггеры `BEFORE`, срабатывающие до внесения изменений в таблицу, и триггеры `AFTER`, срабатывающие после внесения изменений. Класс триггера определяется соответствующим ключевым словом в его определении.

Также триггеры делятся на *триггеры уровня команды*, которые исполняются для команды DML в целом, и *триггеры уровня записи*, которые исполняются отдельно для каждого кортежа, на который влияет команда DML. Триггер уровня записи определяется опцией `FOR EACH ROW`, указываемой при создании триггера. Если её нет, триггер будет срабатывать на уровне команды.

Последовательность срабатывания триггеров: сначала срабатывают все триггеры `BEFORE` уровня команды, затем все триггеры `BEFORE` уровня записи, затем выполняется сама команда DML, затем срабатывают все триггеры `AFTER` уровня записи и затем, наконец, срабатывают все триггеры `AFTER` уровня команды. По умолчанию, порядок срабатывания триггеров одного вида случайный, однако его можно предопределить с помощью специальных команд (см. раздел 5.3).

В опциональной секции `WHEN` указывается условие, при котором должен срабатывать триггер. Условие записывается всегда в круглых скобках (в отличие от условия в операторе `IF`). Секция используется только в триггерах уровня записи.

Один триггер может реагировать сразу на несколько видов команд DML, в этом случае они должны быть перечислены в определении триггера и связаны ключевым словом `OR` (например, `DELETE OR UPDATE`). Для триггера, реагирующего на оператор модификации данных, можно сузить область определения, указав список атрибутов, изменение которых нас интересует (`UPDATE OF <attr_list>`).

В триггере, срабатывающем сразу для нескольких команд, могут быть участки кода, относящиеся только к одной из этих команд. Для их различения используются *операционные директивы*:

- `INSERTING` — истина, если триггер среагировал на операцию вставки;
- `DELETING` — истина, если триггер среагировал на операцию удаления;
- `UPDATING` — истина, если триггер среагировал на операцию обновления;
- `UPDATING('<attr_name>')` — перегруженная версия предыдущей директивы, позволяющая отследить обновление конкретного атрибута; при этом имя атрибута указывается в одинарных кавычках.

Операционные директивы могут быть вызваны в любом коде PL/SQL, но только в триггерах и вызванных из них процедурах или функциях эти директивы могут вернуть значение `TRUE`.

Как и в любом другом коде PL/SQL, в триггере может содержаться блок локальных определений DECLARE и блок обработки исключений EXCEPTION.

Триггер DML всегда является частью транзакции, поэтому подтверждать или отменять изменения таблиц БД в триггере, как правило, не нужно — это приведёт к подтверждению или отмене транзакции в целом. Стоит отметить важный момент, связанный с транзакциями и обработкой исключений: если во время выполнения триггера происходит исключение и мы его отлавливаем и обрабатываем прямо в триггере, это никак не повлияет на транзакцию в целом (она будет подтверждена или отменена независимо от того, возникло это исключение или нет). Но если в триггере мы исключение не обработаем, оно будет проброшено на уровень выше, в код, инициировавший операцию DML. Это означает, что с большой вероятностью произойдёт откат транзакции. Этой особенностью можно и нужно пользоваться — грамотная обработка или необработка исключений зачастую позволяет существенно упростить код триггера.

В триггерах уровня записи доступны две специальные структуры — *псевдозаписи* NEW и OLD. Они имеют такую же структуру (включая поле ROWID), как кортеж таблицы БД, для которой написан триггер, а к отдельным полям можно обращаться так же, как к полям записей RECORD. В псевдозаписи NEW содержатся новые значения обрабатываемой в настоящий момент записи, а в псевдозаписи OLD — старые значения.

При работе с полями псевдозаписей в основном блоке триггера их названия снабжаются префиксом-двоеточием — :NEW.<attr_name> и :OLD.<attr_name>. Если же псевдозаписи используются в определении триггера (в секции WHEN), обращение к ним выполняется без двоеточий — NEW.<attr_name> и OLD.<attr_name>.

В триггере, связанном с командой INSERT, по очевидным причинам не определена псевдозапись OLD, а в триггере, связанном с командой DELETE, не определена псевдозапись NEW. В триггере, связанном с командой UPDATE, определены обе псевдозаписи, при этом их ROWID совпадают.

Псевдозаписи нельзя обрабатывать и передавать «как целое», можно работать только с отдельными полями. Кроме того, нельзя модифицировать значения полей псевдозаписи OLD.

При определении триггера можно переименовать псевдозаписи в опциональной секции REFERENCING. При обращении к псевдозаписям по новым именам следует использовать тот же синтаксис — с двоеточием в основном блоке и без двоеточия в секции WHEN.

Рассмотрим некоторые примеры триггеров. Следующий триггер будет проверять корректность номера курса и признака магистратуры при вставке или обновлении записи о группе (триггер уровня записи):

```
CREATE OR REPLACE TRIGGER course_check
BEFORE INSERT OR UPDATE OF course, mag
ON AGroup FOR EACH ROW
BEGIN
  IF :NEW.mag = 'Да' THEN
    IF :NEW.course > 2 THEN
      DBMS_OUTPUT.PUT_LINE('Некорректный
            номер курса для магистратуры.');
      IF INSERTING THEN RAISE
         Exceptions_pck.invalid_course_mag;
      ELSE :NEW.course := :OLD.course;
      END IF;
    END IF;
  END IF;

  IF UPDATING THEN
    IF :OLD.mag = 'Да' AND :NEW.mag = 'Нет'
    THEN
      DBMS_OUTPUT.PUT_LINE('Нельзя группу
                  магистратуры сделать
                  группой бакалавриата.');
      RAISE
        Exceptions_pck.invalid_mag_to_bach;
    END IF;

    IF :OLD.mag = 'Нет' AND
      :NEW.mag = 'Да' AND
      (:OLD.course < 4 OR :NEW.course > 1) OR
      :OLD.mag = :NEW.mag AND
      :OLD.course > :NEW.course OR
      :NEW.course > :OLD.course + 1
    THEN
      DBMS_OUTPUT.PUT_LINE('Некорректное
                изменение номера курса.');
      RAISE
        Exceptions_pck.invalid_course_change;
    END IF;
  END IF;
END;
```

В таблице на номер курса наложено ограничение CHECK, однако оно не учитывает признак магистратуры (в магистратуре есть

только 1 и 2 курс). Так что первое действие в триггере проверяет именно это, причём в случае реакции на операцию вставки поднимается исключение `invalid_course_mag`, определённое в пакете `Exceptions_pck`, а в случае реакции на операцию обновления новое значение заменяется старым без ущерба для транзакции. Второй блок проверок относится только к операции обновления: нельзя группу магистратуры перевести в бакалавриат, нельзя понизить номер курса и нельзя увеличить номер курса больше чем на единицу (в последних двух случаях учитывается возможный переход из бакалавриата в магистратуру). Если реализовалась какая-то из «плохих» ситуаций, поднимается соответствующее исключение из того же пакета. Обратите внимание, что исключения в триггере не отлавливаются — задача по их обработке лежит на коде, инициировавшем операцию DML.

Из триггеров можно вызывать процедуры и функции, в том числе процедуры и функции пакетов. Предположим, что каждая из проверок из предыдущего триггера реализована в виде отдельной хранимой функции, возвращающей 0, если ошибки в данных нет, и возвращающей 1, если ошибка есть. Тогда код триггера может быть переписан так:

```
CREATE OR REPLACE TRIGGER course_check
BEFORE INSERT OR UPDATE OF course, mag
ON AGroup FOR EACH ROW
BEGIN
  IF chk_mag_course(:NEW.mag,:NEW.course) = 1
  THEN
    IF INSERTING THEN RAISE
        Exceptions_pck.invalid_course_mag;
    ELSE :NEW.course := :OLD.course;
    END IF;
  END IF;

  IF UPDATING THEN
    IF chk_mag_to_bach(:OLD.mag,:NEW.mag) = 1
    THEN RAISE
        Exceptions_pck.invalid_mag_to_bach;
    END IF;

    IF check_course_change(:OLD.mag,
        :OLD.course,:NEW.mag,:NEW.course) = 1
    THEN RAISE
        Exceptions_pck.invalid_course_change;
    END IF;
  END IF;
END;
```

С помощью триггеров можно создавать «продвинутые» внешние ключи взамен стандартных. В этом случае необходимо прописать реакцию на 4 действия: вставку и изменение дочернего кортежа, а также изменение и удаление родительского кортежа. К примеру, перепишем внешний ключ, отражающий вхождение студента в учебную группу с помощью триггеров:

```
ALTER TABLE Student
DISABLE CONSTRAINT student_agroup_fk1;
/
CREATE OR REPLACE TRIGGER group_del_upd
AFTER DELETE OR UPDATE OF name
ON AGroup FOR EACH ROW
DECLARE
  temp_var INTEGER;
BEGIN
  IF UPDATING THEN
    SELECT COUNT(*) INTO temp_var
    FROM Student
    WHERE :OLD.name = group_name;
    IF temp_var > 0 THEN
      :NEW.name := :OLD.name;
    END IF;
  ELSE
    UPDATE Student SET group_name = NULL
    WHERE group_name = :OLD.name;
  END IF;
END;
/
CREATE OR REPLACE TRIGGER student_ins
AFTER INSERT ON Student
BEGIN
  FOR grp_rec IN
    (SELECT DISTINCT group_name FROM Student
     WHERE group_name NOT IN
       (SELECT name FROM AGroup))
  LOOP
    INSERT INTO AGroup
    VALUES (grp_rec.group_name,
            'Без факультета', 1, NULL,
            'Без специальности', 'Нет');
  END LOOP;
END;
/
CREATE OR REPLACE TRIGGER student_upd
BEFORE UPDATE OF group_name ON Student
```

```
FOR EACH ROW
WHEN (NEW.group_name IS NOT NULL)
DECLARE
  temp_var INTEGER;
BEGIN
  SELECT COUNT(*) INTO temp_var
  FROM AGroup WHERE name = :NEW.group_name;
  IF temp_var = 0 THEN
    :NEW.group_name := :OLD.group_name;
  END IF;
END;
```

При удалении группы (родительский кортеж) будет происходить очистка поля «Название группы» у всех студентов этой группы (как при обычном внешнем ключе). После добавления нового студента (дочерний кортеж) будет создана запись о новой группе на основе поля «Название группы» этого студента, если такой группы в таблице AGroup раньше не было. Обычный внешний ключ такое действие инициировать не может. Обратите внимание, что здесь мы используем триггер уровня команды, поскольку нет нужды запускать одну и ту же проверку и один и тот же запрос на вставку отдельно для каждой вставляемой строки. При обновлении записей о группе или студенте будет запрещено менять название группы, если в ней есть хотя бы один студент, а также переводить студента в несуществующую группу (но можно оставить его без группы, в частности, для того, чтобы не было противоречия с триггером, удаляющим группу) — во всех этих случаях будет оставлено существующее название группы.

Замена внешнего ключа триггерами — сложная и не всегда реализуемая задача. Важно понимать, что один триггер может инициировать другой триггер и так далее, в результате чего триггеры «войдут в клинч» и транзакция не сможет выполниться. Особенно вероятна такая ситуация для триггеров, реагирующих на обновление двух связанных таблиц.

Триггер иногда бывает нужно объявить *автономной транзакцией* — в этом случае операции COMMIT и ROLLBACK в триггере не будут влиять на основную транзакцию, а результаты всех подтверждённых внутри автономной транзакции операций DML будут сохранены независимо от успешности основной транзакции. Автономая транзакция объявляется директивой PRAGMA AUTONOMOUS_TRANSACTION в блоке DECLARE. Часто этой возможностью пользуются при необходимости *журнализации* (*логирования*) операций DML:

```
CREATE OR REPLACE TRIGGER group_upd_log
BEFORE UPDATE OF name ON AGroup
```

```
FOR EACH ROW
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO Group_Log
         (old_name, new_name, update_time)
  VALUES (:OLD.name, :NEW.name, SYSTIMESTAMP);
  COMMIT;
END;
```

Триггер в приведённом примере будет отслеживать все попытки изменения названия учебной группы, *в том числе неудачные*, и записывать их в специальную таблицу-журнал `Group_Log` с указанием даты и времени изменения.

Обратите внимание, что триггер уровня записи *не может* считывать или записывать данные таблицы, с которой он связан, поскольку находящуюся в стадии изменения таблицу сложно анализировать. Однако при объявлении триггера автономной транзакцией появляется возможность чтения данных из «своей» таблицы, но запись всё равно будет невозможна.

Триггер уровня команды не может работать с псевдозаписями, зато он способен и читать, и писать данные в таблицу, с которой связан.

Триггер DML нельзя создавать в схеме администратора БД (`SYS`, `SYSTEM` и т. п.), однако от имени администратора его можно создать в схеме выбранного пользователя.

5.2. Сопровождение триггеров

Как и обычные ограничения, триггеры можно включать/выключать с помощью команды

```
ALTER TRIGGER <trigger_name> ENABLE|DISABLE;
```

Отключённый триггер остаётся в схеме, но не используется. Если триггер нужно удалить из схемы, запускается команда

```
DROP TRIGGER <trigger_name>;
```

Триггер можно компилировать в отключённом состоянии, для этого в заголовке триггера перед блоком `DECLARE` нужно вставить ключевое слово `DISABLE`.

Основную информацию о триггерах можно получить из словаря данных через представления `USER_TRIGGERS`, `ALL_TRIGGERS` и `DBA_TRIGGERS`. Однако ни одно из этих представлений не содержит информации о работоспособности триггера (триггер, созданный некорректной командой PL/SQL, всё равно сохраняется в схеме, но помечается как `INVALID`). Валидность триггера, равно как и любого другого объекта схемы, отмечает-

ся в поле STATUS представлений USER_OBJECTS, ALL_OBJECTS и DBA_OBJECTS.

5.3. Однотипные и составные триггеры

В Oracle с одной таблицей БД можно связать несколько триггеров одного типа. По умолчанию порядок их срабатывания не определён. Рассмотрим характерный пример из [2]. Пусть имеется таблица Inc_Val, содержащая единственный числовой атрибут val_inc. Создадим два триггера:

```
CREATE OR REPLACE TRIGGER inc_by_one
BEFORE INSERT ON Inc_Val FOR EACH ROW
BEGIN
  :NEW.val_inc := :NEW.val_inc + 1;
END;
/
CREATE OR REPLACE TRIGGER inc_by_two
BEFORE INSERT ON Inc_Val FOR EACH ROW
BEGIN
  IF :NEW.val_inc > 1
    :NEW.val_inc := :NEW.val_inc + 2;
  END IF;
END;
```

При выполнении команды вставки

```
INSERT INTO Inc_Val VALUES (1);
```

значение в таблице с равной вероятностью может оказаться как 2, так и 4.

В современных версиях Oracle порядок срабатывания триггеров можно строго определить с помощью команд PRECEDES и FOLLOWS. Скажем, в нашем примере логичным будет указать, что увеличение на 2 происходит после увеличения на 1:

```
CREATE OR REPLACE TRIGGER inc_by_two
BEFORE INSERT ON Inc_Val FOR EACH ROW
FOLLOWS inc_by_one
BEGIN
  IF :NEW.val_inc > 1
    :NEW.val_inc := :NEW.val_inc + 2;
  END IF;
END;
```

Отметим, что в любой выбранной паре однотипных триггеров только один может ссылаться на другой с помощью этих директив. При этом Oracle позволяет выстраивать сложные иерархии триггеров, однако триггер, на который идёт ссылка по директиве PRECEDES или FOLLOWS, должен быть откомпилирован раньше триггера, который эту директиву содержит.

В современных версиях Oracle также возможно использовать *составные триггеры* — объекты, объединяющие в себе триггеры уровня команды и уровня записи.

Общий синтаксис определения составного триггера:

```
CREATE [OR REPLACE]
       TRIGGER [<schema>.]<trigger_name> FOR
{INSERT|DELETE|UPDATE|UPDATE OF <attr_list>}
[OR {...} ...] ON [<schema>.]<table_name>
[REFERENCING OLD AS <name1> NEW AS <name2>]
[WHEN (<condition>)]
COMPOUND TRIGGER
   [<global_declarations>]
   [BEFORE STATEMENT IS
      [<local_declarations>]
   BEGIN
      <operators>
   [EXCEPTION
      <exception_handle>]
   END BEFORE STATEMENT;]
   [BEFORE EACH ROW IS
      [<local_declarations>]
   BEGIN
      <operators>
   [EXCEPTION
      <exception_handle>]
   END BEFORE EACH ROW;]
   [AFTER EACH ROW IS
      [<local_declarations>]
   BEGIN
      <operators>
   [EXCEPTION
      <exception_handle>]
   END AFTER EACH ROW;]
   [AFTER STATEMENT IS
      [<local_declarations>]
   BEGIN
      <operators>
   [EXCEPTION
      <exception_handle>]
   END AFTER STATEMENT;]
END;
```

Составной триггер содержит от 1 блока до 4, соответствующих различным типам триггеров (до/после операции, уровня записи/команды), каждый блок срабатывает в свою очередь. Как и в простых триггерах, работа с псевдозаписями возможна только

в блоках уровня записи, секции REFERENCING и WHEN относятся тоже только к блокам уровня записи, причём проверяемое условие будет одинаковым для обоих блоков.

Составной триггер является менее гибким, нежели простой — например, блок составного триггера или триггер в целом нельзя объявить автономной транзакцией. Зато в составном триггере появляется возможность определения глобальных (в пределах триггера) переменных, типов и даже процедур/функций. Все эти элементы будут создаваться непосредственно перед запуском блока BEFORE STATEMENT и существовать до окончания выполнения блока AFTER STATEMENT, то есть на всём протяжении «времени жизни» оператора DML, на который среагировал триггер. При выполнении нового оператора DML все элементы блока глобальных определений будут созданы заново. Соответственно, при использовании составного триггера не нужно создавать вспомогательные пакеты с определением переменных, типов и функций и следить за корректностью значений переменных (обнулять при запуске новой команды DML).

На составной триггер можно ссылаться с помощью директив PRECEDES и FOLLOWS. Если нужного блока в триггере нет, директива будет просто проигнорирована.

5.4. Работа с коллекциями в триггерах

В триггерах можно работать с переменными-коллекциями точно так же, как и в любой программе PL/SQL. Это особенно важно в тех случаях, когда триггер связывается с таблицей, содержащей атрибуты-коллекции.

Рассмотрим простой пример. Пусть определён тип коллекции на уровне схемы:

```
CREATE TYPE date_tt AS TABLE OF DATE;
```

Пусть также определена таблица:

```
CREATE TABLE Personal_Events
(
    person_id NUMBER(6) PRIMARY KEY,
    important_dates date_tt
) NESTED TABLE important_dates
  STORE AS imp_d_nt;
```

Реализуем триггер, фильтрующий события по году при их добавлении/обновлении так, чтобы оставались только даты с 2005 и большим годом:

```
CREATE OR REPLACE TRIGGER year_filter
BEFORE INSERT OR UPDATE ON Personal_Events
FOR EACH ROW
```

```
DECLARE
    date_list date_tt;
BEGIN
    date_list := :NEW.important_dates;
    FOR i IN 1..date_list.COUNT
    LOOP
        IF EXTRACT
             (YEAR FROM date_list(i)) < 2005
        THEN date_list.DELETE(i);
        END IF;
    END LOOP;
    :NEW.important_dates := SET(date_list);
END;
```

Обратите внимание, что для фильтрации значений создаётся отдельная вспомогательная таблица `date_list`. Это сделано для того, чтобы данные в таблицу были записаны плотно. Кроме того, мы используем функцию `SET` при последнем присваивании, чтобы устранить дубликаты дат.

Как уже отмечалось ранее, использование ограничения `FOREIGN KEY` для атрибутов-коллекций невозможно. Однако же содержимое коллекций в одной таблице зачастую должно соответствовать записям какой-то другой таблицы. Рассмотрим на примере, каким образом внешний ключ в описанной ситуации может быть реализован с помощью триггеров.

Определим тип таблицы идентификаторов:

```
CREATE OR REPLACE TYPE num_list
                          AS TABLE OF NUMBER(6);
```

Создадим две таблицы:

```
CREATE TABLE Producer
(
    id NUMBER(6,0) PRIMARY KEY,
    name VARCHAR2(30)
);
/
CREATE TABLE Performer
(
    id NUMBER(6,0) PRIMARY KEY,
    name VARCHAR2(30),
    list_of_producers num_list
) NESTED TABLE list_of_producers
  STORE AS prod_lst_of_perf;
```

В этих таблицах будет реализовываться связь «многие-ко-многим»: один исполнитель может работать с разными продюсерами, а один продюсер может работать с разными артистами. Будем хранить список идентификаторов продюсеров как атрибут-

коллекцию в таблице исполнителей. Естественно, этот список должен быть подмножеством всех идентификаторов продюсеров.

Для отслеживания ограничений внешнего ключа нам надо предусмотреть реакцию на 4 действия: вставку/обновление исполнителя и обновление/удаление продюсера. Потребуем такой реакции на данные действия:

- запрет удаления продюсера, если он работает хотя бы с одним артистом (действие типа `RESTRICT`), с инициированием исключения `prod_del_error` из пакета `Exceptions_pck`;
- запрет вставки исполнителя с некорректным списком продюсеров (действие типа `RESTRICT`), с инициированием исключения `perf_ins_error` из пакета `Exceptions_pck`;
- откат к старой версии записи об исполнителе (в том числе к старому списку продюсеров) без ущерба для транзакции в целом, если в новой версии списка продюсеров указан идентификатор несуществующего продюсера;
- каскадное изменение идентификатора продюсера во всех списках (самая сложная часть).

Набор триггеров, реализующих указанный функционал, выглядит так:

```
CREATE OR REPLACE TRIGGER trigger_on_performers
BEFORE INSERT OR UPDATE ON Performer
FOR EACH ROW
DECLARE
  list_ids num_list;
BEGIN
  FOR i IN 1..:NEW.list_of_producers.COUNT
  LOOP
    IF :NEW.list_of_producers(i) IS NULL
    THEN :NEW.list_of_producers.DELETE(i);
    END IF;
  END LOOP;
  :NEW.list_of_producers :=
                SET(:NEW.list_of_producers);

  SELECT id BULK COLLECT INTO list_ids
  FROM Producer;
  IF :NEW.list_of_producers
                NOT SUBMULTISET OF list_ids
  THEN
    IF INSERTING THEN
```

```
      DBMS_OUTPUT.PUT_LINE
        ('Некорректный список продюсеров.');
      RAISE Exceptions_pck.perf_ins_error;
    ELSE
      :NEW.id := :OLD.id;
      :NEW.name := :OLD.name;
      :NEW.list_of_producers :=
                    :OLD.list_of_producers;
      DBMS_OUTPUT.PUT_LINE
          ('Запись с идентификатором ' ||
           :OLD.id || ' не была обновлена
           из-за нарушения внешнего ключа.');
    END IF;
  END IF;
END;
/
CREATE OR REPLACE TRIGGER
                      trigger_on_producers_del
BEFORE DELETE ON Producer
FOR EACH ROW
BEGIN
  FOR perf_rec IN (SELECT * FROM Performer)
  LOOP
    FOR i IN
         1..perf_rec.list_of_producers.COUNT
    LOOP
      IF perf_rec.list_of_producers(i) = :OLD.id
    THEN
       DBMS_OUTPUT.PUT_LINE
          ('Продюсера с идентификатором ' ||
           :OLD.id || ' удалять нельзя -
           он ещё работает.');
       RAISE Exceptions_pck.prod_del_error;
      END IF;
    END LOOP;
  END LOOP;
END;
/
CREATE OR REPLACE TRIGGER
                      trigger_on_producers_upd
FOR UPDATE OF id ON Producer
COMPOUND TRIGGER
  TYPE changes_arr IS TABLE OF NUMBER(6)
                       INDEX BY PLS_INTEGER;
  prod_changes changes_arr;
```

```
    AFTER EACH ROW IS
    BEGIN
      prod_changes(:OLD.id) := :NEW.id;
    END AFTER EACH ROW;

    AFTER STATEMENT IS
      list_ids num_list;
      flag BOOLEAN := FALSE;
    BEGIN
      FOR perf_rec IN (SELECT * FROM Performer)
      LOOP
        flag := FALSE;
        list_ids := perf_rec.list_of_producers;
        FOR i IN 1..list_ids.COUNT LOOP
          IF prod_changes.EXISTS(list_ids(i))
          THEN
            list_ids(i) :=
                      prod_changes(list_ids(i));
            flag := TRUE;
          END IF;
        END LOOP;
        IF flag = TRUE
        THEN
          UPDATE Performer
          SET list_of_producers = SET(list_ids)
          WHERE id = perf_rec.id;
        END IF;
      END LOOP;
    END AFTER STATEMENT;
  END;
```

Первый триггер — `trigger_on_performers` — реагирует на вставку и обновление записей в таблице `Performer`. Обратите внимание, что мы используем директиву `UPDATE` в определении триггера, хотя нас интересует только обновление списка продюсеров. Это связано с тем, что перегруженная версия директивы `UPDATE OF` (равно как и директива `UPDATING('<attr_name>')`) применима только к атомарным атрибутам.

Первая группа операторов в теле триггера выполняет «естественное» форматирование списка продюсеров — устраняет `NULL`-значения и дубликаты записей. Поскольку список продюсеров имеет тип вложенной таблицы, последнее действие выполняется с помощью одной команды `SET`.

Вторая группа операторов отвечает непосредственно за реализацию функционала внешнего ключа, относящегося к табли-

це Performer. Для различения событий, на которые реагирует триггер, мы пользуемся операционной директивой INSERTING.

Второй триггер — trigger_on_producers_del — с помощью двух вложенных циклов ищет идентификатор удаляемого продюсера во всех списках продюсеров в таблице Performer и поднимает исключение, если находит.

Самый сложный, третий, триггер — trigger_on_producers_upd — отвечает за каскадное обновление идентификатора продюсера во всех списках. Сложность состоит в том, что нам нужно как-то разрешить возможную коллизию, когда триггер уровня записи для таблицы Performer будет инициирован во время обработки триггера уровня записи для таблицы Producer. Как уже отмечалось, такая ситуация наиболее вероятна именно для двух триггеров типа UPDATE (хотя и для других сочетаний она тоже может возникнуть). Возникновение обозначенной коллизии приведёт к «падению» оператора DML.

Для разрешения коллизии мы используем составной триггер на обновление. В блоке AFTER уровня записи мы просто сохраним значения изменяющихся идентификаторов в ассоциативном массиве, определяемом как глобальная переменная составного триггера. Поскольку мы данные не меняем, триггер на обновление исполнителя не инициируется.

Основная обработка будет происходить в блоке AFTER уровня команды. Раз изменения в таблице Producer уже произошли, инициирование триггера уровня записи для таблицы Performer не повлечёт за собой коллизию. Пользуясь сохранённым ассоциативным массивом и булевым флагом, мы переберём все списки продюсеров и внесём в них все необходимые изменения.

При создании ассоциативного массива реализован элегантный и экономный подход — старое значение идентификатора продюсера выступает в качестве индекса массива, а новое значение идентификатора выступает в качестве значения элемента массива. Поскольку в таблице Producer идентификатор — первичный ключ, накладки с индексами произойти не может.

5.5. Упражнения

1. Создайте триггер для таблицы AGroup, проверяющий, что староста — это студент той же группы.

2. Напишите набор триггеров, реализующих функционал внешнего ключа для атрибутов-коллекций таблицы AGroup, разработанных в упражнении 2 главы 4.

3. Напишите триггер, выполняющий журнализацию всех изменений (вставка, обновление, удаление) таблицы Student.

4. Придумайте для таблицы `Student` составной триггер, содержащий все четыре возможных блока.

5. Напишите запрос к словарю данных, позволяющий узнать, какие из триггеров, принадлежащих вам, являются валидными, а какие — нет.

6. Объектно-ориентированные возможности PL/SQL

В этой главе мы рассмотрим объектно-ориентированные возможности PL/SQL, которые реализуются посредством объектных типов OBJECT и экземпляров данных типов.

6.1. Создание объектных типов

Понятия *типа объекта* и *экземпляра объекта* подобны понятиям класса и экземпляра класса в объектно-ориентированном программировании. Как и классы в языках ООП, объекты в Oracle реализуют принципы *абстракции*, *инкапсуляции*, *наследования* и *полиморфизма*. Синтаксис определения типа объекта похож на синтаксис объявления класса в языке Java.

Тип объекта можно определять только на уровне схемы с помощью команды CREATE [OR REPLACE] TYPE. Базовый синтаксис:

```
CREATE [OR REPLACE] TYPE <type_obj>
            {AS OBJECT|UNDER <base_type_obj>}
(
  [<field_name> <field_type>]
  [, <...>]
  [, CONSTRUCTOR FUNCTION <type_obj>
     [(<param_list>)] RETURN SELF AS RESULT]
  [, <...>]
  [, [OVERRIDING|NOT INSTANTIABLE] MEMBER
        {PROCEDURE|FUNCTION} <declaration>]
  [, <...>]
  [, STATIC{PROCEDURE|FUNCTION}<declaration>]
  [, <...>]
  [, {MAP|ORDER} MEMBER FUNCTION <declaration>]
  [, <...>]
) [NOT INSTANTIABLE] [FINAL|NOT FINAL];
```

Указателем на то, что создаётся именно объектный тип, является ключевое слово AS OBJECT, если создаётся базовый тип, или предложение UNDER <base_type_obj>, если объект является производным (наследником). Для производного объекта указывать AS OBJECT не нужно, поскольку от объектного типа можно наследовать только другой объектный тип.

Объект может быть объявлен абстрактным с помощью ключевого слова NOT INSTANTIABLE в конце определения. В этом случае экземляру данного объектного типа нельзя будет присвоить значение. Ключевое слово FINAL|NOT FINAL определяет, будет

создаваемый тип объекта окончательным (от него нельзя наследовать другие объектные типы) или нет. Абстрактный тип должен быть определён как NOT FINAL. По умолчанию используется опция FINAL.

Объектный тип содержит набор *полей*, каждое из которых может быть определено как экземпляр любого простого (стандартного) или составного (коллекция, объект и т. д.) типа. Поля супертипа наследуются всеми подтипами, прописывать их заново в спецификации не требуется.

Кроме того, объектный тип содержит набор процедур и функций, называемых *методами объекта*. Все методы, за исключением статических, работают с конкретным экземпляром объекта. Рассмотрим виды методов.

Во-первых, для любого неабстрактного объектного типа можно определить *конструктор*, с помощью которого будет выполняться инициализация экземпляра объекта. Как и в языках ООП, возможно создать несколько перегруженных версий конструктора. Если не создано ни одного пользовательского конструктора, будет определён конструктор по умолчанию. Конструктор не может быть процедурой — это всегда функция, возвращающая инициализированный экземпляр объекта в качестве результата (RETURN SELF AS RESULT). Соответственно, работа конструктора должна завершаться либо оператором возврата (без параметров), либо исключением. На конструктор указывает ключевое слово CONSTRUCTOR в спецификации метода. Конструктор может инициализировать только часть полей экземпляра (или вовсе ни одного), неинициализированные поля получат неопределённое значение (NULL).

В любом объектном типе можно определять *методы экземпляров* (MEMBER PROCEDURE и MEMBER FUNCTION), работающие с полями экземпляра объекта. Эти методы организуются подобно методам пакетов, с разделением на спецификацию и тело метода. Абстрактные методы (NOT INSTANTIABLE) не имеют тела, но могут быть переопределены в наследном типе. Вообще, любой метод супертипа, необязательно абстрактный, может быть переопределён в подтипе, что задаётся директивой OVERRIDING.

Методы сравнения MAP и ORDER являются специальными разновидностями методов экземпляра. Методы MAP определяют критерии сравнения экземпляров объектов с помощью операторов =, != (<>), >, >=, <, <=. Методы ORDER определяют критерии сортировки экземпляров объектов с помощью оператора ORDER BY в SQL-запросах. Эти методы всегда являются функциями, сравнение осуществляется по возвращаемому значению (обычно это

число). Метод сравнения задаётся директивой MAP или ORDER в определении.

Статические методы — это методы объектных типов, не зависящие от экземпляров объекта. Статический метод задаётся директивой STATIC в определении типа объекта.

Рассмотрим пример для нашей базы данных «Университет». Создадим иерархию объектных типов: на верхнем уровне будет абстрактный объектный тип «Персона» с полями «Идентификатор», «Имя», «Пол» и «Вознаграждение», а также с абстрактными методами проверки полей и печати (вывода значений всех полей); от данного типа будут наследовать два типа — «Студент» и «Преподаватель». Поле «Идентификатор» будет соответствовать номеру студенческого билета или табельному номеру преподавателя, а поле «Вознаграждение» — стипендии студента или зарплате преподавателя. Кроме того, в каждом из производных типов будут специфические поля («Название группы» у студента и «Название кафедры» и «Должность» у преподавателя) и методы (снятие, назначение и индексация стипендии студента, назначение и индексация зарплаты преподавателя, а также смена кафедры и должности). Отметим, что модель объектов условна — она не предполагает полной информации о персонах университета, главная цель модели — демонстрация возможностей объектных типов.

```
CREATE OR REPLACE TYPE person_obj AS OBJECT
(
  id NUMBER(7),
  name VARCHAR2(100),
  payment NUMBER(10,2),
  gender VARCHAR2(1),
  NOT INSTANTIABLE MEMBER FUNCTION chk_fields
    RETURN BOOLEAN,
  MEMBER FUNCTION print RETURN VARCHAR2
) NOT INSTANTIABLE NOT FINAL;
/
CREATE OR REPLACE TYPE student_obj
                        UNDER person_obj
(
  group_name varchar2(10),
  CONSTRUCTOR FUNCTION student_obj
     (id IN NUMBER, name IN VARCHAR2,
       agrant IN NUMBER DEFAULT 0,
       gender IN VARCHAR2,
       group_name IN VARCHAR2 DEFAULT NULL)
    RETURN SELF AS RESULT,
```

```
  OVERRIDING MEMBER FUNCTION chk_fields
    RETURN BOOLEAN,
  OVERRIDING MEMBER FUNCTION print
    RETURN VARCHAR2,
  MEMBER PROCEDURE drop_grant,
  MEMBER PROCEDURE set_grant
                   (new_grant IN NUMBER),

  MEMBER PROCEDURE raise_grant
                   (percentage IN NUMBER)
);
/
CREATE OR REPLACE TYPE tutor_obj
                         UNDER person_obj
(
  dept_name VARCHAR2(50),
  job_name VARCHAR2(30),
  CONSTRUCTOR FUNCTION tutor_obj
      (id IN NUMBER, name IN VARCHAR2,
       salary IN NUMBER DEFAULT 12000,
       gender IN VARCHAR2,
       dept_name IN VARCHAR2 DEFAULT NULL,
       job_name IN VARCHAR2 DEFAULT NULL)
    RETURN SELF AS RESULT,
  OVERRIDING MEMBER FUNCTION chk_fields
    RETURN BOOLEAN,
  OVERRIDING MEMBER FUNCTION print
    RETURN VARCHAR2,
  MEMBER PROCEDURE set_salary
                   (new_salary IN NUMBER),
  MEMBER PROCEDURE raise_salary
                   (percentage IN NUMBER),
  MEMBER PROCEDURE change_dept
                   (new_dept IN VARCHAR2),
  MEMBER PROCEDURE change_job
                   (new_job IN VARCHAR2)
);
```

6.2. Создание тела объектного типа

Тело объектного типа создаётся с помощью конструкции, подобной конструкции создания тела пакета:

```
CREATE [OR REPLACE] TYPE BODY <type_obj> AS
   <proc_and_func_implementation>
END;
```

В теле объектного типа содержится реализация всех неабстрактных функций, в том числе неабстрактных функций абстрактного типа. То есть у абстрактного типа может быть тело типа, но экземпляр объекта при этом не может быть инициализирован (нет конструктора).

Вернёмся к нашему примеру. Создадим тело абстрактного типа `person_obj`, так как в нём есть один неабстрактный метод `print`. Пусть этот метод просто выводит список полей практически без дополнительного форматирования.

```
CREATE OR REPLACE TYPE BODY person_obj AS
  MEMBER FUNCTION print RETURN VARCHAR2 AS
    BEGIN
    RETURN 'Номер - ' || id ||
           ', имя - ' || name ||
           ', вознаграждение - ' || payment ||
           ', пол - ' || gender || '.';
  END print;
END;
```

Теперь создадим тело объектного типа `student_obj`. В конструкторе будем проверять, что номер студенческого, имя и пол непусты, а неопределённую стипендию превратим в нулевую. Также будем запускать метод `chk_fields` для проверки того, что стипендия неотрицательна, пол указан корректно, а группа существует в таблице `AGroup`. Если какая-то из перечисленных проверок не пройдена, будет поднято исключение `invalid_type_fields` из пакета `Exceptions_pck`. Если же данные корректны, произойдёт успешная инициализация объекта.

Функцию `print` перепишем для данного подтипа так, чтобы вывод осуществлялся с учётом возможных `NULL`-значений названия группы. Изменение стипендии в функциях `drop_grant`, `set_grant` и `raise_grant` реализуем так, чтобы нельзя было сделать стипендию пустой, отрицательной и чтобы нельзя было проиндексировать её на неположительный процент. В случае попытки некорректного изменения стипендии поднимается исключение `invalid_type_fields`.

Реализация тела типа:

```
CREATE OR REPLACE TYPE BODY student_obj AS
  CONSTRUCTOR FUNCTION student_obj
     (id IN NUMBER, name IN VARCHAR2,
      agrant IN NUMBER DEFAULT 0,
      gender IN VARCHAR2,
      group_name IN VARCHAR2 DEFAULT NULL)
  RETURN SELF AS RESULT AS
  BEGIN
```

```
  IF id IS NULL OR
     name IS NULL OR
     gender IS NULL
  THEN RAISE
     Exceptions_pck.invalid_type_fields;
  END IF;

  SELF.id := id;
  SELF.name := name;
  SELF.payment := NVL(agrant,0);
  SELF.gender := gender;
  SELF.group_name := group_name;

  IF SELF.chk_fields THEN RETURN;
  ELSE RAISE
     Exceptions_pck.invalid_type_fields;
  END IF;
END student_obj;

OVERRIDING MEMBER FUNCTION chk_fields
  RETURN BOOLEAN AS
    cnt NUMBER;
BEGIN
  IF id <= 0 THEN
    DBMS_OUTPUT.PUT_LINE
                (’Неправильный номер’);
    RETURN FALSE;
  ELSIF gender NOT IN (’М’,’Ж’) THEN
    DBMS_OUTPUT.PUT_LINE
                (’Неправильный пол’);
    RETURN FALSE;
  ELSIF payment < 0 THEN
    DBMS_OUTPUT.PUT_LINE
                (’Неправильная стипендия’);
    RETURN FALSE;
  ELSIF group_name IS NOT NULL THEN
    SELECT COUNT(*) INTO cnt FROM AGroup
    WHERE AGroup.name = group_name;
    IF cnt = 0 THEN
      DBMS_OUTPUT.PUT_LINE
                (’Неправильная группа’);
      RETURN FALSE;
    END IF;
  END IF;
  RETURN TRUE;
```

```
  EXCEPTION
    WHEN OTHERS THEN RETURN FALSE;
END chk_fields;

OVERRIDING MEMBER FUNCTION print
  RETURN VARCHAR2 AS
    res VARCHAR2(2000);
BEGIN
  res := 'Номер - ' || id;
  res := res || ', имя - ' || name;
  res := res || ', стипендия - ' || payment;
  res := res || ', пол - ' || gender;
  IF group_name IS NULL THEN
    res := res || ', без группы.';
  ELSE
    res := res || ', группа - ' ||
                        group_name || '.';
  END IF;
  RETURN res;
END print;

MEMBER PROCEDURE drop_grant AS
BEGIN
  payment := 0;
  RETURN;
END drop_grant;

MEMBER PROCEDURE set_grant
                  (new_grant IN NUMBER) AS
BEGIN
  IF new_grant IS NOT NULL AND
     new_grant >= 0
  THEN
    payment := new_grant;
    RETURN;
  END IF;
  RAISE Exceptions_pck.invalid_type_fields;
END set_grant;

MEMBER PROCEDURE raise_grant
                  (percentage IN NUMBER) AS
BEGIN
  IF percentage <= 0
  THEN RAISE
```

```
        Exceptions_pck.invalid_type_fields;
    END IF;
    payment := payment * (1 + percentage / 100);
    RETURN;
  END raise_grant;
END;
```

Реализация тела объектного типа `tutor_obj` идейно во многом похожа на приведённую выше, но теперь мы учитываем ещё и то условие, что зарплата не может быть меньше 12000 и не может уменьшаться.

```
CREATE OR REPLACE TYPE BODY tutor_obj AS
  CONSTRUCTOR FUNCTION tutor_obj
      (id IN NUMBER, name IN VARCHAR2,
       salary IN NUMBER DEFAULT 12000,
       gender IN VARCHAR2,
       dept_name IN VARCHAR2 DEFAULT NULL,
       job_name IN VARCHAR2 DEFAULT NULL)
    RETURN SELF AS RESULT AS
  BEGIN
    IF id IS NULL OR name IS NULL OR
       gender IS NULL OR salary IS NULL
    THEN RAISE
       Exceptions_pck.invalid_type_fields;
    END IF;

    SELF.id := id;
    SELF.name := name;
    SELF.payment := salary;
    SELF.gender := gender;
    SELF.dept_name := dept_name;
    SELF.job_name := job_name;
    IF SELF.chk_fields THEN RETURN;
    ELSE RAISE
       Exceptions_pck.invalid_type_fields;
    END IF;
  END tutor_obj;

  OVERRIDING MEMBER FUNCTION chk_fields
    RETURN BOOLEAN AS
  BEGIN
    IF id <= 0 THEN
      DBMS_OUTPUT.PUT_LINE
                 ('Неправильный номер');
      RETURN FALSE;
    ELSIF gender NOT IN ('M','Ж') THEN
```

```
    DBMS_OUTPUT.PUT_LINE
                (’Неправильный пол’);
    RETURN FALSE;
  ELSIF payment < 12000 THEN
    DBMS_OUTPUT.PUT_LINE
                (’Неправильная зарплата’);
    RETURN FALSE;
  END IF;
  RETURN TRUE;

  EXCEPTION
    WHEN OTHERS THEN RETURN FALSE;
END chk_fields;

OVERRIDING MEMBER FUNCTION print
  RETURN VARCHAR2 AS
    res VARCHAR2(2000);
BEGIN
  res := ’Номер - ’ || id;
  res := res || ’, имя - ’ || name;
  res := res || ’, зарплата - ’ || payment;
  res := res || ’, пол - ’ || gender;
  IF dept_name IS NULL THEN
    res := res || ’, без кафедры’;
  ELSE
    res := res || ’, кафедра - ’ || dept_name;
  END IF;
  IF job_name IS NULL THEN
    res := res || ’, без должности.’;
  ELSE
    res := res || ’, должность - ’ ||
                            job_name || ’.’;
  END IF;
  RETURN res;
END print;

MEMBER PROCEDURE set_salary
                  (new_salary IN NUMBER) AS
BEGIN
  IF new_salary IS NULL OR
     new_salary < payment
  THEN
    DBMS_OUTPUT.PUT_LINE
                (’Неправильная зарплата’);
    RAISE Exceptions_pck.invalid_type_fields;
```

```
    END IF;
    payment := new_salary;
    RETURN;
  END set_salary;

  MEMBER PROCEDURE raise_salary
                  (percentage IN NUMBER) AS
  BEGIN
    IF percentage <= 0
    THEN RAISE
       Exceptions_pck.invalid_type_fields;
    END IF;
    payment := payment * (1 + percentage / 100);
    RETURN;
  END raise_salary;

  MEMBER PROCEDURE change_dept
                  (new_dept IN VARCHAR2) AS
  BEGIN
    dept_name := new_dept;
    RETURN;
  END change_dept;

  MEMBER PROCEDURE change_job
                  (new_job IN VARCHAR2) AS
  BEGIN
    job_name := new_job;
    RETURN;
  END change_job;
END;
```

6.3. Работа с экземплярами объектных типов

Экземпляр объекта создаётся как переменная соответствующего объектного типа. Как и любая обычная переменная, созданный объект не инициализирован (NULL), соответственно, перед использованием его нужно инициализировать с помощью конструктора, путём агрегатного присваивания (приравнивание другому экземпляру того же объектного типа) либо посредством выборки из базы данных.

Синтаксис использования конструктора:

```
<object> := [NEW] <type_obj>
                      (param1[, param2 ...]);
```

Вызов метода экземпляра объекта выполняется с использованием стандартного «точечного» синтаксиса:

```
<object>.<method>(param1[, param2 ...])
```

При вызове методов можно использовать именованную, позиционную или смешанную нотацию (как при работе с процедурами и функциями в PL/SQL). Вызов статических методов осуществляется через обращение к типу объекта, не к экземпляру:

```
<type_obj>.<st_method>(param1[, param2 ...])
```

К отдельным полям объекта можно обращаться таким же образом:

```
<object>.<field>
```

Экземпляры абстрактных типов могут быть объявлены, но их нельзя инициализировать. Однако экземпляру абстрактного типа можно присвоить экземпляр любого его подтипа, тогда он будет работать как этот подтип и содержать все специфические поля этого подтипа.

При вызове методов выполняется всегда метод самого специализированного подтипа, связанного с данным экземпляром объекта. При этом реализуется *динамическая диспетчиризация методов* — выбор метода происходит на этапе выполнения. Если требуется выполнить метод супертипа, нужно воспользоваться конструкцией

```
(<object> AS <supertype>).<method>
                          (param1[, param2 ...])
```

Имя супертипа при этом прописывается в явном виде.

Пример с иллюстрацией вышесказанного для экземпляров объектов типа `student_obj` и `person_obj`:

```
DECLARE
    student1 student_obj;
    person1 person_obj;
BEGIN
    student1 := student_obj (id => 1,
          name => 'Петров', gender => 'M');
    DBMS_OUTPUT.PUT_LINE(student1.print());
    student1 := student_obj (id => 1,
          name => 'Иванов', gender => 'M',
          group_name => 'ИТ-21МО',
          agrant => 5000);
    DBMS_OUTPUT.PUT_LINE(student1.print());
    student1.set_grant(1800);
    DBMS_OUTPUT.PUT_LINE(student1.print());

    student1.name := 'Сидоров';
    DBMS_OUTPUT.PUT_LINE(student1.print());
    person1 := student1;
```

```
    DBMS_OUTPUT.PUT_LINE(person1.print());

    DBMS_OUTPUT.PUT_LINE
        ((student1 AS person_obj).print());
    DBMS_OUTPUT.PUT_LINE
        ((person1 AS person_obj).print());
END;
```

В первом блоке команд мы дважды инициализируем объект `student1` разными значениями, при этом в первом конструкторе часть полей заполняется значениями по умолчанию. Также мы меняем стипендию студента посредством метода `set_grant`. Значения всех полей на каждом шаге мы выводим с помощью метода `print`.

Во втором блоке команд мы сначала напрямую обращаемся к полю `name`, меняя его, а затем определяем объект `person1` абстрактного супертипа `person_obj` на основе имеющегося инициализированного объекта `student1` производного объектного типа. Метод `print` для каждого из этих объектов выполнится как метод типа `student_obj` (наиболее специализированный подтип), выведя одно и то же.

В третьем блоке команд демонстрируется обращение к методу `print` супертипа `person_obj`. Результат вывода будет другим, нежели во втором блоке, но по-прежнему одинаковым для обоих объектов.

6.4. Объектные таблицы и объектные атрибуты. Функции для работы с объектами

На основе объектных типов можно создавать два вида таблиц — объектные таблицы и таблицы с объектными атрибутами. Рассмотрим оба варианта.

Объектная таблица — это таблица, структура которой полностью соответствует структуре объекта. На поля объектного типа в данной таблице можно накладывать ограничения точно так же, как на атрибуты обычных таблиц. Если не объявить первичный ключ, он будет сгенерирован автоматически как уникальный идентификатор объекта (`OID`). Важной особенностью объектных таблиц является то, что в них можно помещать объекты производных типов. Специфические для подтипов поля будут храниться в скрытых столбцах, по одному скрытому столбцу на каждое специфическое поле каждого подтипа. Поэтому объектные таблицы — достаточно гибкие структуры данных. Синтаксис создания объектной таблицы:

```
CREATE TABLE [<schema>.]<table_name>
OF [<schema>.]<type_obj>
[(<constraints>)];
```

Для примера создадим объектную таблицу на основе нашего абстрактного типа `person_obj`. В качестве первичного ключа выберем поле `id` и наложим ограничение типа `CHECK` на поле `gender`:

```
CREATE TABLE Person OF person_obj
(CONSTRAINT pers_pk PRIMARY KEY (id),
 CONSTRAINT pers_gend_chk
                CHECK (gender IN ('M','Ж')));
```

При вставке элементов в объектную таблицу нужно использовать либо существующий экземпляр объекта, либо конструктор:

```
BEGIN
   INSERT INTO Person VALUES
      (student_obj(id => 1,
                   name => 'Петров',
                   agrant => 2000,
                   gender => 'M',
                   group_name => 'ИТ-21МО'));
   INSERT INTO Person VALUES
      (tutor_obj(id => 2,
                    name => 'Иванов',
                    salary => 42000,
                    gender => 'M',
                    dept_name => 'ИВТ',
                    job_name => 'доцент'));
   COMMIT;
END;
```

Таблица с объектным атрибутом во многом подобна таблице с атрибутом-коллекцией, для объектного атрибута действуют те же правила. Так, например, на объектный атрибут нельзя накладывать стандартные ограничения реляционных БД кроме ограничения `NOT NULL`, можно только моделировать их с помощью триггеров. Создание таких таблиц и работа с ними осуществляется как с таблицами, содержащими атрибут-коллекцию (но с добавлением специфических для объектных типов функций). Как и объектная таблица, объектный атрибут может содержать экземпляр производного типа.

Для примера создадим таблицу участников некоторой конференции, каждый из которых может быть и студентом, и преподавателем:

```
CREATE TABLE Participant
(
   id NUMBER(6) PRIMARY KEY,
```

```
    arole VARCHAR2(11) CHECK
                (arole IN ('Докладчик',
                           'Слушатель',
                           'Организатор')),
    arrival DATE NOT NULL,
    departure DATE NOT NULL,
    person_inf person_obj NOT NULL,
    CHECK (arrival <= departure)
);
```

Добавим в таблицу участника-студента и участника-преподавателя:

```
BEGIN
    INSERT INTO Participant VALUES
       (11111, 'Слушатель',
        '01.10.2020', '05.10.2020',
        student_obj(id => 1,
                 name => 'Петров',
                 agrant => 2000,
                 gender => 'M',
                 group_name => 'ИТ-21МО'));
    INSERT INTO Participant VALUES
       (11122, 'Докладчик',
        '30.09.2020', '05.10.2020',
        tutor_obj(id => 2,
                  name => 'Иванов',
                  salary => 42000,
                  gender => 'M',
                  dept_name => 'ИВТ',
                  job_name => 'доцент'));
    COMMIT;
END;
```

Для выборки и удобной работы с объектными данными из объектных таблиц и таблиц с объектными атрибутами в PL/SQL предусмотрены две функции — VALUE и TREAT.

Функция VALUE служит для извлечения экземпляров объектов из объектных таблиц. При этом данные преобразуются к объектному типу, на основе которого определена данная таблица. По сути, это будет супертип верхнего уровня. Однако если в таблице содержался экземпляр подтипа, то все специфические для него поля будут сохранены, а при обращении к перегруженным методам будут вызываться именно методы подтипа. Синтаксис обращения к функции VALUE:

```
VALUE(<alias>)[.{<field>|
                 <method>([<param_list>])}]
```

Пример:

```
SELECT VALUE(p).print() FROM Person p;
```

Поскольку при выборке из объектных таблиц или таблиц с объектными атрибутами экземпляры объектов возвращаются как объекты супертипа верхнего уровня, для обращения к специфическим для подтипа полям и методам нужно выполнять *понижающее преобразование*, то есть предписать Oracle трактовать экземпляр объекта как экземпляр подтипа. Для этого используется функция TREAT:

```
TREAT(<object> AS <subtype_obj>)
      [.{<field>|<method>([<param_list>])}]
```

Если мы не знаем заранее, какой именно подтип нам нужен, можно использовать предикат IS OF для различения подтипов:

```
<object> IS OF ([ONLY] <subtype_obj>[, ...])
```

Предикат вернёт значение TRUE, если объект принадлежит к одному из подтипов, указанных в скобках. Директива ONLY предписывает проверять строгое соответствие, без неё предикат будет истинным даже в том случае, когда объект принадлежит к одному из производных типов указанного подтипа.

Осуществим следующую выборку из объектной таблицы Person:

```
DECLARE
  p1 person_obj;
BEGIN
  FOR pers_rec IN (SELECT VALUE(p) pers
                   FROM Person p)
  LOOP
    p1 := pers_rec.pers;
    CASE
      WHEN p1 IS OF (student_obj)
      THEN DBMS_OUTPUT.PUT_LINE
        (p1.name || ' - студент, ' ||
         TREAT(p1 AS student_obj).group_name);
      WHEN p1 IS OF (tutor_obj)
      THEN DBMS_OUTPUT.PUT_LINE
        (p1.name || ' - ' ||
         TREAT(p1 AS tutor_obj).job_name || ' ,'
         || TREAT(p1 AS tutor_obj).dept_name);
    END CASE;
  END LOOP;
END;
```

Обратите внимание, что, в зависимости от принадлежности объекта к тому или иному подтипу, мы выводим значения специфических для типов student_obj и tutor_obj полей, используя понижающее преобразование. При этом поле name определено

в супертипе person_obj, поэтому для него понижающее преобразование выполнять не нужно.

6.5. Эволюция объектных типов

В объектные типы можно добавлять поля с помощью команды

```
ALTER TYPE <type_obj>
    ADD ATTRIBUTE <field_name> <data_type>
    CASCADE INCLUDING TABLE DATA;
```

Последняя директива предписывает Oracle выполнить каскадное обновление всех созданных экземпляров объектов, в том числе внести изменения во все объектные таблицы и объектные атрибуты таблиц.

Однако команда ALTER TYPE ... ADD ATTRIBUTE не способна изменить методы и/или их сигнатуры.

Если требуется изменить метод без изменения сигнатуры, нужно просто переписать и перекомпилировать его в теле объекта (и, при необходимости, в телах всех производных объектов).

Если же меняется не только сам метод, но и его сигнатура, то этот метод нужно сначала удалить из спецификации объекта, затем снова его туда поместить с новой сигнатурой, а после этого уже переписать тело метода, в том числе в телах производных объектов, и перекомпилировать. Базовый синтаксис для замены метода:

```
ALTER TYPE <type_obj> DROP
    {MEMBER|CONSTRUCTOR|MAP|ORDER|STATIC}
    {PROCEDURE|FUNCTION} <method_name>
        (<old_params>) [RETURN <data_type>]
    CASCADE;
ALTER TYPE <type_obj> ADD
    {MEMBER|CONSTRUCTOR|MAP|ORDER|STATIC}
    {PROCEDURE|FUNCTION} <method_name>
        (<new_params>) [RETURN <data_type>]
    CASCADE;
```

Удаление объектного типа осуществляется командой

```
DROP TYPE <type_obj> [FORCE|VALIDATE];
```

Ключевое слово FORCE предписывает Oracle удалить объектный тип даже в том случае, если этот тип используется в таблицах БД. Все производные объектные типы и объектные таблицы, созданные на основе данного типа, станут неприменимы. Объектные атрибуты удаляемого типа станут недоступны. Опция VALIDATE позволяет удалять объектный тип только в том случае, если ни один его экземпляр не используется ни в одной таблице.

6.6. Упражнения

1. Разработайте объектные типы со следующей иерархией: на верхнем уровне — абстрактный тип «Подразделение», на нижнем уровне — производные типы «Учебная группа», «Кафедра», «Лаборатория». Поля должны учитывать все естественные характеристики. Также должны быть предусмотрены необходимые процедуры и функции.

2. Реализуйте тела всех подтипов, разработанных при решении предыдущего упражнения.

3. Создайте объектную таблицу на основе объектного типа «Подразделение».

4. Перепишите тела объектных типов `student_obj` и `tutor_obj` так, чтобы в их конструкторах проверялось наличие подразделения в объектной таблице из упражнения 3 с учётом типа подразделения (преподаватель не может состоять в учебной группе и т. п.).

5. Реализуйте хранимую процедуру, выводящую из разработанной объектной таблицы список подразделений с полной информацией о них; для каждого подразделения должен выводиться список состоящих в нём персон из таблицы `Person` с полной информацией о каждой персоне. В итоге должен получиться двухуровневый список.

Методические указания по подготовке к экзамену

Дисциплина «Объектные базы данных» является довольно сложной. Для успешной сдачи экзамена студенту следует посетить все лекции, внимательно изучить материал настоящего пособия, выполнить упражнения. Теоретическая составляющая экзамена во многом базируется именно на информации, изложенной в пособии. Для понимания материала также рекомендуется не ограничиваться базовым уровнем при выполнении задания лабораторной работы, а постараться реализовать полный функционал.

Порядок проведения экзамена

Экзамен проводится в письменной форме. Задание представляет собой тест, каждый вопрос которого предполагает развёрнутый ответ. Каждый ответ оценивается целым числом баллов, которые впоследствии суммируются. Максимальный балл за экзаменационный тест равен 24. Список задач теста и максимальный балл за каждую из них приведены ниже в разделе «Структура экзаменационного билета».

Длительность экзамена — 3 часа (180 минут) в очном формате или 2 часа (120 минут) в дистанционном формате.

На экзамене запрещается использование любых источников информации, как электронных, так и аналоговых. Знания соседа приравниваются к аналоговому источнику информации. В случае несанкционированного их использования студент вместе с источником удаляется из аудитории с оценкой 0.

Баллы, полученные за экзаменационный тест, складываются с баллами, полученными за лабораторную работу, и переводятся в итоговую экзаменационную оценку. При этом максимальный балл за лабораторную работу равен 30. Правила выведения итоговой оценки:

Условие	**Оценка**
0–24	«неудовлетворительно»
25–34	«удовлетворительно»
35–44	«хорошо»
45–54	«отлично»

Структура экзаменационного билета

№	Задание	Балл
1	Теоретический вопрос по теме 1	1
2	Теоретический вопрос по теме 2	1
3	Теоретический вопрос по теме 3	1
4	Теоретический вопрос по теме 4	1
5	Теоретический вопрос по теме 5	1
6	Теоретический вопрос по теме 6	1
7	Найти ошибку в коде PL/SQL	2
8	Написать простой запрос типа `SELECT`	1
9	Написать запрос типа `SELECT`, использующий агрегацию данных	2
10	Написать запрос типа `SELECT`, содержащий вложенный запрос	2
11	Написать простой DML-запрос	1
12	Написать сложный DML-запрос	2
13	Написать процедуру/функцию на языке PL/SQL	2
14	Написать триггер	2
15	Написать процедуру/функцию, работающую с коллекциями	2
16	Написать запрос для создания таблицы	2
Максимальный балл:		**24**

Комментарии к заданиям

Задания 1–6 предполагают краткий ответ (одно-два предложения) на поставленный теоретический вопрос. Теоретические вопросы ставятся в виде: «Что такое ...?», «Зачем нужно ...?», «Что нужно сделать, чтобы получить ...?», «Какие вы знаете виды ...?», «Чем отличаются ...?», «В каком случае ...?», «Можно ли ...?» — или в аналогичных формах.

Задание 7 требует найти одну ошибку в данном коде PL/SQL и выписать её в форме:

```
№ строки (строк): краткое описание ошибки.
```

В *заданиях 8–12* дана схема (структура) таблиц базы данных, для которых нужно реализовать указанные запросы. При этом запросы в заданиях 8 и 11 работают только с одной табли-

цей и не используют агрегацию или вложенные запросы. Каждое из решений задач 8–12 должно содержать *только один* SQL-запрос.

В *задании 13* также дана схема таблиц БД. Необходимо написать процедуру или функцию на языке PL/SQL, обязательно использующую курсоры и/или курсорные циклы.

В *задании 14* необходимо реализовать триггер для одной из таблиц БД. При этом в триггере не потребуется обращаться к сложным типам данных (коллекциям или объектам).

Задание 15 предполагает, что нужно организовать некоторые данные в виде коллекции, а затем обработать их.

В *задании 16* требуется создать таблицу БД с помощью оператора `CREATE TABLE`, прописав все естественные ограничения.

В *заданиях 13, 15* должен присутствовать блок обработки исключений.

Литература

1. Oracle Database Documentation. URL: https://docs.oracle.com/en/database/oracle/oracle-database/index.html.

2. Фейерштейн, С. Oracle PL/SQL. Для профессионалов / С. Фейерштейн, Б. Прибыл. — 6-е изд. — СПб. : Питер, 2015. — 1024 с.

3. Кригель, А. SQL. Библия пользователя / А. Кригель, Б. М. Трухнов. — 2-е изд. — М. : Диалектика, 2010. — 752 с.

Учебное издание

Смирнов Александр Валерьевич

Основы Oracle PL/SQL

Учебно-методическое пособие

Редактор, корректор Л. Н. Селиванова
Компьютерная вёрстка А. В. Смирнов

Подписано в печать 22.06.2020. Формат 60 × 84/16.
Усл.-печ. л. 5,8. Уч.-изд. л. 4,0.
Тираж 3 экз. Заказ №

Оригинал-макет подготовлен
в редакционно-издательском отделе ЯрГУ.

Ярославский государственный университет им. П. Г. Демидова
150003, г. Ярославль, ул. Советская, 14