Объектно-реляционная модель данных

Бессарабов Н.В.
bes@fpm.kubsu.ru
2017 г.

Введение в процедурный язык PL/SQL

Блоки.

Программы PL/SQL структурируются анонимными (не имеющими имени) блоками. Структура такого блока:

```
DECLARE
объявления
]
BEGIN
выполняемые операторы
[EXCEPTION
обработка исключительных ситуаций
]
END;
```

Блок обязательно заканчивается точкой с запятой.

Необязательный раздел обработки исключительных ситуаций мы рассматривать не будем. Конечно, вы понимаете, что реальные программы без секций исключительных ситуаций это просто генераторы неприятностей для пользователя. Но для целей предварительного знакомства можно временно считать, что исключительных ситуаций не бывает.

Пример блока

Объявление переменных и констант можно выполнить так:

```
имя_переменной тип_данных [[NOT NULL] := выражение_по_умолчанию];
имя_переменной тип_данных CONSTANT := выражение;
```

Присваивание обозначается знаком :=.

Разберём несложный пример анонимного блока (рис.). Набрав текст нажимаем Run.

```
DECLARE
   x      NUMBER(5,2);
   y      NUMBER(5,2):=2;
   result NUMBER(5,2);
 BEGIN
   -- Это однострочный комментарий
   result := 12.02;
   x := result / y;
   DBMS_OUTPUT.PUT_LINE('x равен ' || x);
END;
```

Язык PL/SQL, в отличие от SQL, “молчаливый” язык. Вывод на экран в нём необходимо прописать явно. Процедура PUT_LINE из пакета DBMS_OUTPUT позволяет вывести на экран строку, сформированную как конкатенация (обозначенная знаком ||) из двух строк – текста 'x равен ' и строки, представляющей значение x. Числовой тип x здесь неявно приводится к текстовому типу. Теперь строка DBMS_OUTPUT.PUT_LINE('x равен ' || x); понятна.

SQL внутри PL/SQL

Поскольку PL/SQL не персистентный язык, то работу с базами данных в нём выполняют операторы SQL. Запросы SELECT теперь должны выдавать результат не на экран, а в подготовленные переменные или другие структуры.

Создадим таблицу и введём в неё одну строку:

```
CREATE TABLE qq(c1 CHAR(10), c2 INTEGER);
INSERT INTO qq VALUES('QWERT', 1);
```

Извлечём значение первого столбца запросом вида SELECT .. INTO:

```
DECLARE
   result char(10);
  BEGIN
   SELECT c1 INTO result FROM qq WHERE c2=1;
   DBMS_OUTPUT.PUT_LINE(result);
END;
```

Удаление слов INTO result вызовет появление ошибки.

Разветвления и циклы

Синтаксис команды разветвления в общем обычный:

```
IF условие THEN последовательность_операторов_1
ELSIF условие2 THEN последовательность_операторов_2
ELSE  последовательность_операторов_3
END IF;
```

Обратите внимание на то, что вместо обычного ELSEIF почему-то пишется ELSIF.

Фраза ELSIF может отсутствовать или быть повторена более одного раза.

Циклы строятся на основе так называемого простого цикла:

```
[<<имя_цикла>>]
LOOP
последовательность_операторов_1
EXIT имя_цикла WHEN условие_выхода
последовательность_операторов_2
END LOOP;
```

Без инструкций останова EXIT или EXIT WHEN он зацикливается.

Фраза в двойных угловых скобках <<имя_цикла>> это метка имени цикла.

Примеры циклов

Пример простого цикла с меткой:

```
DECLARE
v_i INTEGER := 1;
BEGIN
   <<loop_1>>
   LOOP
      DBMS_OUTPUT.PUT_LINE('В цикле  v_i = ' || v_i);
      EXIT loop_1 WHEN v_i > 2;
      v_i := v_i + 1;
   END LOOP;
   DBMS_OUTPUT.PUT_LINE
      ('Вышли из цикла');
END;
```

Пример цикла WHILE с предусловием:

```
DECLARE
i INTEGER:=1;
BEGIN
WHILE i < 6
LOOP
DBMS_OUTPUT.PUT_LINE(i);
i :=i +1;
END LOOP;
END;
```

Пример цикла FOR:

```
BEGIN
FOR i IN 1 .. 5 LOOP
DBMS_OUTPUT.PUT_LINE(i);
END LOOP;
END;
```

Переменная i не была объявлена явно, и цикл сам создал её с типом INTEGER. Слово REVERSE во фразе FOR i IN REVERSE 1 .. 5 LOOP задало бы обратное направление перебора значений.

Процедуры и функции (1/3)

Хранимые процедуры и функции создаются на основе анонимного блока путём добавления в него четвёртой секции, содержащей спецификацию процедуры/функции. В неё входят имя и список формальных параметров. Особенность функции в том, что она возвращает значение, а процедура нет.

Уп едуры:

```
CREATE [OR REPLACE] PROCEDURE имя_процедуры
[(имя_параметра [IN | OUT | INOUT] тип [, …])]
IS | AS
BEGIN
   тело_процедуры
END имя_процедуры;
```

Это спецификация процедуры

Упрощённый синтаксис инструкции создания функции:

```
CREATE [OR REPLACE] FUNCTION имя_функции
[(имя_параметра [IN | OUT | INOUT] тип [, …])]
RETURN тип_возвращаемого значения
IS | AS
BEGIN
   тело_функции
END имя_функции;
```

Это спецификация функции. Добавлена фраза RETURN

Процедуры и функции (2/3)

Тело функции или процедуры это последовательность операторов.

Слова OR REPLACE во фразу CREATE добавляют, если необходимо заменить существующую функцию с таким же наименованием.

Режим параметров указывается как IN – входной, OUT – выходной, или INOUT – входо-выходной. Входной параметр нельзя изменять в теле функции, а выходному обязательно должно быть присвоено значение.

Пример функции, вычисляющей площадь круга по его радиусу:

```
CREATE OR REPLACE FUNCTION circle_area (p_radius IN NUMBER)
   RETURN NUMBER  AS
  v_pi   NUMBER := 3.1415926;
  v_area NUMBER;
BEGIN
  v_area := v_pi * POWER(p_radius, 2);
 RETURN v_area;
END circle_area;
```

Процедуры и функции (3/3)

Фраза RETURN, обязательна в спецификации функции. В ней указывается имя переменной которая будет возвращена. В теле функции, фраза RETURN записывается, по крайней мере, один раз и указывает значение возвращаемой переменной,.

Обратите внимание на то, что в именованиях переменных использован префикс “v_”, в имени формального параметра префикс “p_”. Подобные соглашения об именованиях очень удобны, особенно если они соблюдаются широким кругом разработчиков.

Вызовем функцию circle_area из анонимного блока:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(circle_area(2));
END;
```

Созданная функция может быть вызвана и из SQL, например,

```
SELECT circle_area(2) FROM dual;
```

Dual это такая необычная таблица в Oracle. В ней единственный столбец dummy. Если пользоваться средствами, разработанными фирмой Oracle, то кажется, что она всегда состоит из одной строки. Её используют чтобы “сделать вид” что все данных выбираются только из таблиц. Например, системная дата может быть получена запросом:

```
SELECT sysdate FROM dual;
```

Пакеты

Пакет это набор программных объектов, как правило, логически связанных и использующих общие данные. При ссылке на какой-нибудь объект пакета, в память загружается весь пакет, и все его объекты становятся доступными. Пока имеются ссылки на пакет, он остается в памяти.

Пакет состоит из двух частей:

- спецификация или заголовок пакета;
- тело пакета.

Заголовок пакета содержит спецификации переменных, типов данных, процедур, функций и курсоров которые предполагается сделать доступными извне. Здесь нет программного кода. Курсоры – это стандартное средство для реализации запросов SQL. Мы их не будем рассматривать.

Тело пакета содержит код всех перечисленных в спецификации модулей. Может содержать описания переменных и других объектов, но все они локальны и не доступны извне пакета.

Примеры использования пакетов

Функция GET_DDL пакета DBMS_METADATA позволяет получить тексты определения любого хранимого объекта базы. Примеры:

1. Запрос описания функции

```
SELECT dbms_metadata.get_ddl('PROCEDURE', 'имя_табл.', 'имя_сх.') FROM dual;
```

Например, для только что созданной функции circle_area:

```
SELECT dbms_metadata.get_ddl('FUNCTION', 'CIRCLE_AREA', 'HR') FROM dual;
```

2. Запрос описания таблицы

```
SELECT dbms_metadata.get_ddl('TABLE', 'имя_таблицы', 'имя_схемы') from dual;
```

Оказывается Oracle добавляет много параметров. Запрос

```
SELECT TO_CHAR(dbms_metadata.get_ddl('TABLE', 'QQ', 'HR'))  FROM dual;
```

возвращает текст:

```
CREATE TABLE "HR"."QQ" ( "C1" CHAR(10), "C2" NUMBER(*,0) ) PCTFREE 10 PCTUSED 40
INITRANS 1 MAXTRANS 255 NOCOMPRESS LOGGING STORAGE(INITIAL 65536 NEXT
1048576 MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0 FREELISTS 1 FREELIST
GROUPS 1 BUFFER_POOL DEFAULT) TABLESPACE "USERS"
```

3. Запрос описания пользователя

```
SELECT TO_CHAR(dbms_metadata.get_ddl('USER', 'имя_пользователя')) FROM
dual;
```

В спецификации запроса сначала вид хран.объекта, затем его имя и имя

Объектные типы данных (1/2)

Объекты могут быть постоянными и временными. Хранимый (persistent) объект может быть как значением столбца обычной таблицы, так и строкой таблицы (в этом случае таблицу называют объектной).

Временный (transient) объект создается в памяти и уничтожается после окончания работы программы. Отправив такой объект в базу данных вы делаете его постоянным. А постоянный объект можно извлечь из БД и поместить его во временный объект такого же типа.

Тип объекта (то есть абстрактный тип данных и методы) всегда хранится в словаре. Экземпляр хранимого объекта хранится в таблицах (одной или более) как значение столбца или строка объектной таблицы.

Выборка и манипулирование хранимыми объектами в базе данных осуществляется на языке SQL. Для работы с временными объектами на сервере предназначен язык PL/SQL. Язык C++ позволяет работать с объектами на стороне клиента. Используются вызовы программ на C++ из PL/SQL. В Java можно работать на всех

Объектные типы данных (2/2)

Можно выделить следующие разновидности объектных типов:

- простой объектный тип, который строится на скалярных предопределённых типах данных;
- составной объектный тип, использующий другие объектные типы;
- ссылочный объектный тип;
- типы коллекций двух разновидностей VARRAY и NESTED TABLES.

Ссылочный объектный тип REF это логический указатель, определяющий отношения между экземплярами классов. Он основывается на одноэлементных или коллекционных типах данных.

Указатели REF задают ассоциации UML и заменяют внешние ключи, предоставляя прямую навигацию между объектами разных типов.

VARRAY -- это упорядоченная коллекция фиксированной длины. Хранится в сегменте таблицы, использующей такой тип.

Вложенная таблица NESTED TABLE это неограниченная и неупорядоченная коллекция. Хранится в своём сегменте, не совпадающем с сегментом основной таблицы.

Создание пользовательского типа данных

Создадим тип данных dept_type, описывающий таблицу dept.

```
CREATE TYPE dept_type AS OBJECT (
    Deptno      NUMBER(2),
    Dname       VARCHAR2(14),
    Loc         VARCHAR2(23)
);
```

Похоже на задание таблицы? Правда, в типах имеется опция OR REPLACE.

Затем создадим таблицу emp_dept (то есть emp включающую в себя dept).

```
CREATE  TABLE emp_dept  (
empno           NUMBER(4),
ename           VARCHAR2(10),
job         VARCHAR2(9),
mgr         NUMBER(4),
hiredate                DATE,
sal         NUMBER(7,2),
comm            NUMBER(7,2),
dept1           DEPT_TYPE
);
```

Объектный тип

Спецификация типа

В предыдущем примере была создана только спецификация типа. Её синтаксис:

```
CREATE [OR REPLACE] TYPE [schema .]type_name
{ { IS | AS } OBJECT }
[ { attribute datatype [sqlj_object_type_attr] } ] |
{ [ {[[[NOT] OVERRIDING]
     [[NOT] FINAL] [[NOT] INSTANTIABLE]]
    { { MEMBER | STATIC }
     { procedure_spec | function_spec } |
     {{ MAP | ORDER } MEMBER function_spec}}}]
.,:}
[[NOT] FINAL] [[NOT] INSTANTIABLE];
```

Мы не будем изучать её в полном объёме. Отметим только, что если спецификации методов (MEMBER FUNCTION) в описании типа появляются то в теле типа эти методы должны быть раскрыты.

Методы могут быть функциями и процедурами.

Отметим, что методы MAP и ORDER задают отношения порядка.

Тело типа

Формат тела типа:

```
CREATE TYPE BODY type_name {IS | AS}
    { {MAP | ORDER}
     MEMBER function_body;
    | MEMBER {procedure_body |
     function_body};}
    [MEMBER {procedure_body |
     function_body};]... END;
--
```

Пример: спецификация и тело типа

Спецификация типа person:

```
CREATE TYPE person AS OBJECT (
  name VARCHAR2(40),
  birthday DATE, -- дата рождения
  address address_type,
MEMBER FUNCTION Age -- вернёт возраст
RETURN NUMBER -- возвр. значение
);
```

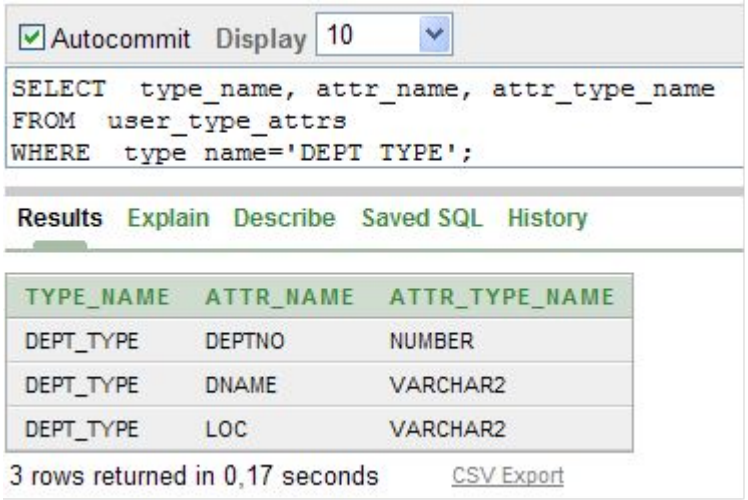
Тело типа person:

```
CREATE OR REPLACE TYPE BODY person IS  --
----- задание методов
MEMBER FUNCTION Age RETURN NUMBER IS
BEGIN -- вычисление возраста
RETURN ROUND(MONTHS_BETWEEN(sysdate,
birthday)/12);
END;
END;
```

Информация о созданном типе

А теперь получим информацию о созданном типе данных из представления словаря USER_TYPE_ATTRS:

```
SELECT type_name, attr_name, attr_type_name
FROM user_type_attrs
WHERE type_name='DEPT_TYPE';
```



Справа структура представления dba_type_attrs

Столбец	Описание столбца
OWNER	Владелец типа
TYPE_NAME	Имя типа
ATTR_NAME	Имя атрибута
ATTR_TYPE_MOD	Модификатор атрибута типа
ATTR_TYPE_OWNER	Владелец атрибута типа
ATTR_TYPE_NAME	Имя типа атрибута
LENGTH	Длина для атрибутов CHAR/VARCHAR
PRECISION	Точность для атрибутов типа number
SCALE	Диапазон для числового типа
CHARACTER_SET_NAME	Имя таблицы символов
ATTR_NO	Номер атрибута

Изменение и удаление типов. Зависимости

Объектные типы можно изменять оператором ALTER TYPE и удалять оператором DROP TYPE.

Форматы некоторых команд:

```
ALTER TYPE имя_типа COMPILE [SPECIFICATION|BODY]
```

Компилирует спецификацию или тело типа хранящиеся в словаре. Если не указано ни одно из слов SPECIFICATION, BODY, то компилируются оба.

```
ALTER TYPE имя_типа REPLACE AS OBJECT (спецификация_объектного_типа)
```

Новое описание, заданное спецификацией_объектного_типа должно во всем, кроме дополнительных методов совпадать с исходным.

После выполнения этого оператора, если тело типа существовало ранее, оно становится INVALID.

```
DROP TYPE [имя_схемы.]имя_типа [][FORCE]
```

Если указан параметр FORCE, то тип удалится, даже если существуют зависимые от него объекты. Естественно, все зависимые объекты становятся INVALID.

Пример зависимости объектов

Создаем два независимых типа

```
CREATE OR REPLACE TYPE Obj1 AS OBJECT (
   C1 NUMBER,
   C2 VARCHAR2(5)
);
CREATE OR REPLACE TYPE Obj2 AS OBJECT (
   C1 CHAR(2),
   C2 VARCHAR2(5)
);
```

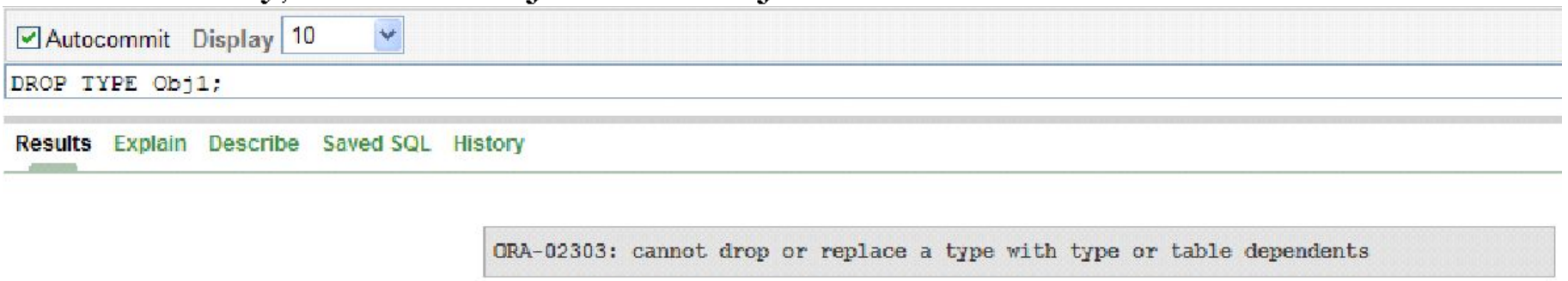
и зависящий от них Obj3

```
CREATE OR REPLACE TYPE Obj3 AS OBJECT (
   a1 Obj1,
   a2 Obj2
);
```

Тогда удаление

```
DROP TYPE Obj1;
```

вызовет ошибку, так как от Obj1 зависит Obj3.



А вот удаление сначала Obj3, затем Obj1 или Obj2 ошибкой не будет.

Конструкторы по умолчанию (1/2)

Метод конструктора возвращает новый экземпляр объектного типа.

Для демонстрации применения конструктора по умолчанию создадим тип address_type, затем на его основе объектный тип person и таблицу peoples со столбцом этого типа

```
CREATE  TYPE  address_type  as object (
    zipcode    NUMBER(5),
    country    VARCHAR2(20),
    city       VARCHAR2(30),
    street     VARCHAR2(30),
    numb            NUMBER(4));
CREATE  TYPE person AS OBJECT (
 name          VARCHAR2(40),
 birthday      DATE,         -- дата рождения
 address       address_type,
 MEMBER FUNCTION Age   -- спецификация метода, возвращающего возраст
          RETURN NUMBER      -- возвращаемое значение
);
```

Конструкторы по умолчанию (2/2)

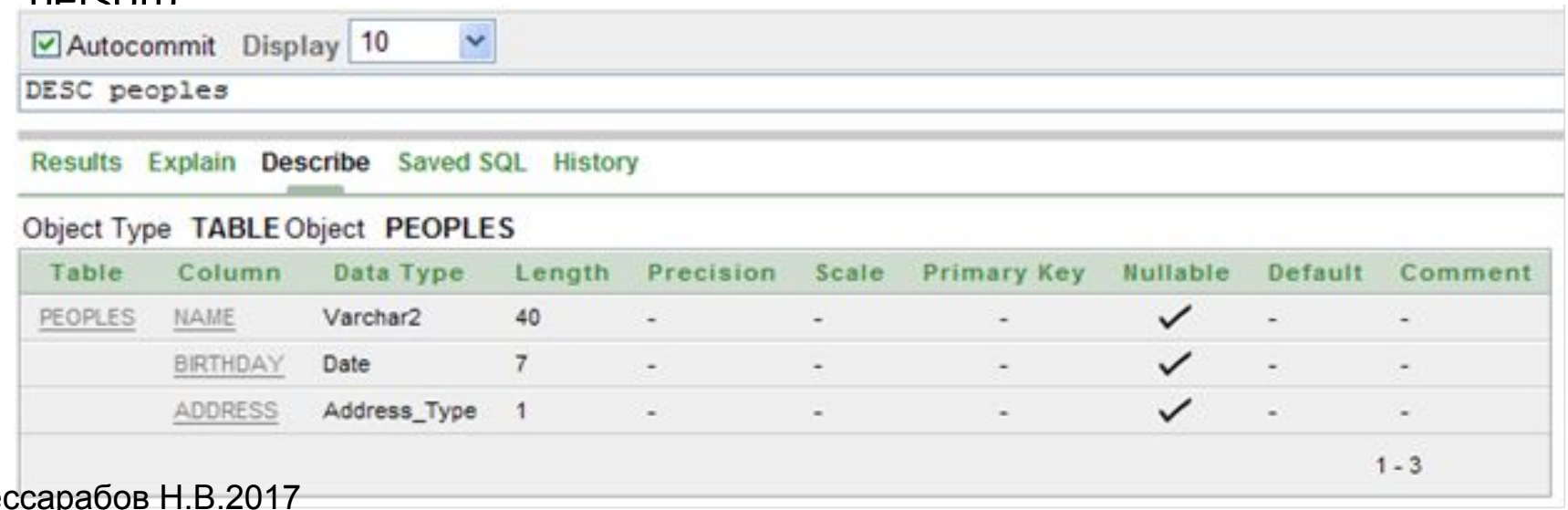
Поскольку определена только спецификация типа, создаём тело типа:

```
CREATE OR REPLACE TYPE BODY person IS -- задание методов
MEMBER FUNCTION Age RETURN NUMBER IS  -- вычисление возраста
  BEGIN
     RETURN ROUND(MONTHS_BETWEEN(sysdate, birthday)/12);
  END;
END;
```

Создадим таблицу типа person командой

```
CREATE TABLE peoples of person;
```

Посмотрим описание таблицы с созданным типом (команда desc person)



Особенности объектных таблиц(1/2)

Из ограничений целостности в объектных таблицах работают только primary key и unique key. Например, создадим таблицу peoples1 с первичным ключом:

```
create table peoples1 of person  (name  primary key);
```

Проверим ее структуру для сравнения с peoples. В столбце Primary Key в строке NAME появится значение 1.

В объектных таблицах вставка строки “по-старому” не удастся. Например, вставка

```
INSERT INTO peoples
  VALUES  ('Сидоров И. П.',  '22.11.80', 350000, 'Россия',
        'Краснодар','Ставропольская', 149);
```

вызывает появление ошибки.

Теперь вставка должна производиться с указанием иерархии типов пользователя, например так:

```
INSERT INTO peoples
  VALUES ('Сидоров И.П.', '22.11.80',
        address_type (35000, 'Россия', 'Краснодар','Ставропольская', 149)
        );
```

Особенности объектных таблиц (2/2)

Читать как в обычных реляционных таблицах также не удастся. Так, команда

SELECT * FROM peoples;

вызывает сообщение об ошибке.

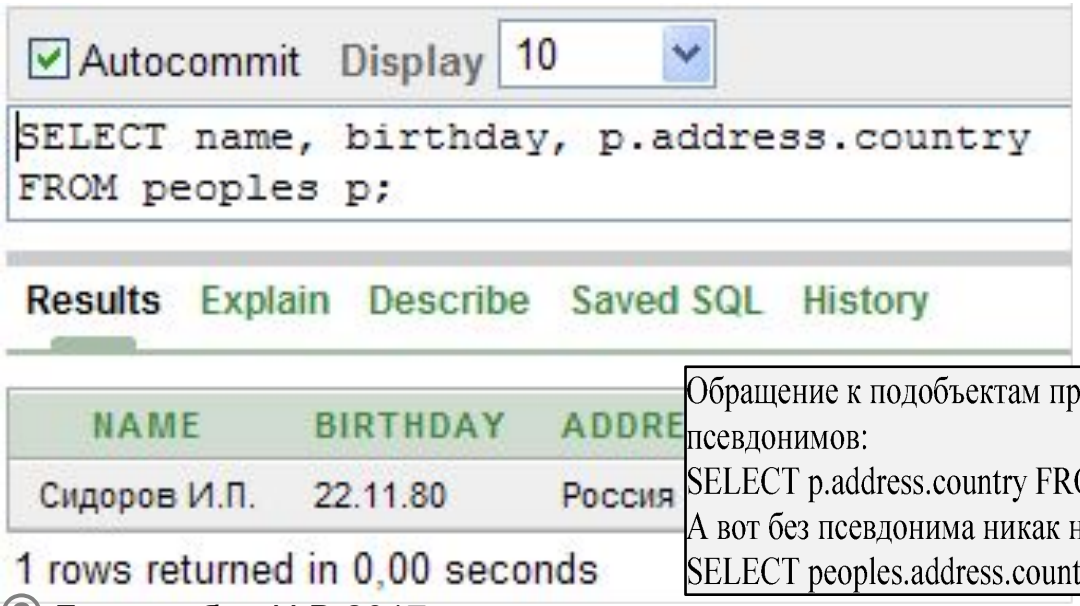
Тот же результат дают запросы:

SELECT name, birthday, address FROM peoples;

SELECT name, birthday, address.country FROM peoples;

А вот следующая команда позволяет работает нормально:

SELECT name, birthday, p.address.country FROM peoples p;



Псевдоним обязателен

Обращение к подобъектам производится с использованием точечного синтаксиса и псевдонимов:
SELECT p.address.country FROM peoples p;
А вот без псевдонима никак нельзя. Следующий запрос вызывает появление ошибки:
SELECT peoples.address.country FROM peoples;

Как хранятся объектные таблицы

Обратимся к представлению user_tab_cols, перечисляющему все столбцы указанной таблицы, в том числе, скрытые (hidden). Создана не совсем обычная таблица peoples со скрытыми столбцами.
name и birthday это обычные реляционные столбцы и читаются обычным запросом. Столбцы ADDRESS, SYS_NC_OID$ и SYS_NC_ROWINFO$ так не читаются. Столбец SYS_NC_OID$ содержит OIDы строк таблицы. Последние пять скрытых столбцов по сочетанию типов подозрительно напоминают столбцы zipcode, country, city, street и numb, определённые в типе address_type. Проверьте это предположение, выполнив запрос
SELECT SYS_NC00006$, SYS_NC00007$, SYS_NC00008$, SYS_NC00009$, SYS_NC00010$ FROM peoples;

Autocommit Display 10

```
SELECT column_name, hidden_column, data_type
FROM user_tab_cols
WHERE table_name = 'PEOPLES';
```

Results Explain Describe Saved SQL History

COLUMN_NAME	HIDDEN_COLUMN	DATA_TYPE
SYS_NC_OID$	YES	RAW
SYS_NC_ROWINFO$	YES	PERSON
NAME	NO	VARCHAR2
BIRTHDAY	NO	DATE
ADDRESS	NO	ADDRESS_TYPE
SYS_NC00006$	YES	NUMBER
SYS_NC00007$	YES	VARCHAR2
SYS_NC00008$	YES	VARCHAR2
SYS_NC00009$	YES	VARCHAR2
SYS_NC00010$	YES	NUMBER

10 rows returned in 0,02 seconds CSV Export

Вывод: объектно-рел. модель эмулируется в табличной (SQL) модели

Индексы и ограничения целостности

Индекс может быть создан на любой листовой столбец объектной таблицы, в том числе принадлежащий вложенной таблице или входящий в атрибут объектного типа.

Забегая вперёд, отметим, что индексировать можно и объектные ссылки, но только в том случае, когда при их определении не только указан тип, на который ссылаются, но и определена таблица этого типа.

Пример:

```
CREATE INDEX country_idx ON peoples (address.country);
```

В этих же ситуациях можно определять и вводить ограничения целостности, например,

```
ALTER TABLE peoples ADD CONSTRAINT peoples_cons1 CHECK
(address.country IS NOT NULL);
```

При создании таблицы также следует использовать ограничения целостности уровня таблицы.

Объектные модели ODMG и SQL (1/2)

Обе объектные модели построены на основе системы типов, хотя при изучении модели Caché в этом можно усомниться.

Структурная реализация объектной модели в Caché отличается от стандартной объектной модели ODMG из-за того, что на объектной основе эмулирована модель SQL, а все данные хранятся в глобалах, имеющих иерархическую структуру.

В объектной модели ODMG имеется два вида типов: литеральные и объектные типы. Литеральные типы определены традиционно как тройка:

<множество литералов, множество значений, набор операций>.

Литеральные типы делятся на атомарные и конструируемые типы. К конструируемым типам относятся структурные литеральные типы и коллекции.

Булевский тип двузначный поскольку в стандарте нет неопределённых значений.

Объектные типы делятся на определяемые пользователем (мы с ними работали в Caché как с классами) и типы коллекций. Как и следовало ожидать, имеется операции создания, инициализации и сохранения объекта.

Ссылочные типы в ODMG отсутствуют, точнее, под одним именем объединены определения типа значений объекта и ссылочного типа OID'ов объектов этого объектного типа.

Объектные модели ODMG и SQL (2/2)

Объектная модель SQL, определённая на уровне стандарта SQL:2003, использует традиционное понятие типа данных. Основные виды типов:

- числовые (точные и приближённые);
- строчные (символьные, битовые, анонимные);
- типы даты и времени;
- типы временных интервалов;
- булев тип – трёхзначный!
- типы коллекций;
- ссылочные типы;
- типы, определяемые пользователем (UDT).

Для расширения семантики хранимых данных была бы полезной реализация в СУБД так называемых индивидуальных типов данных. Например, если заданы два таких числовых типа

```
CREATE TYPE EMP_NO AS INTEGER;
CREATE TYPE DEPT_NO AS INTEGER;
```

то транслятор мог бы сообщить об ошибке сравнения в запросе

```
SELECT EMP_NAME FROM EMP
WHERE EMP_ID > DEPT_I
```

Заключение

В двух последних лекциях мы познакомились с двумя объектными моделями данных.

Обратите внимание на то, что объектно-реляционная модель была эмулирована фирмой Oracle в табличную модель SQL.

Итак, у нас в арсенале пять моделей данных:

- Сущность - связь
- Иерархическая
- Табличная модель SQL
- Объектная
- Объектно-реляционная

В повседневной практике используют ещё модели:

- Многомерные (для аналитики)
- NoSQL, в частности, документарные
- XML- модели

Некоторые полезные для практики модели могут эмулироваться самим пользователем.

Пример: таблицы принятия решений.