**Иерархии, обобщенные табличные выражения**

*Фраза* CONNECT BY *предложения* SELECT

Для моделирования в *БД* фрагментов прикладной области порой удобна иерархическая (многоуровневая) организация данных. Фраза CONNECT BY используется для запросов по иерархически организованным записям. Она представляет собой специфичное расширение *SQL* в *Oracle*, несовместимое с другими диалектами *SQL* (применяющими с той же целью собственные конструкции) и отсутствующее в стандарте.

Элементами фразы CONNECT BY являются:

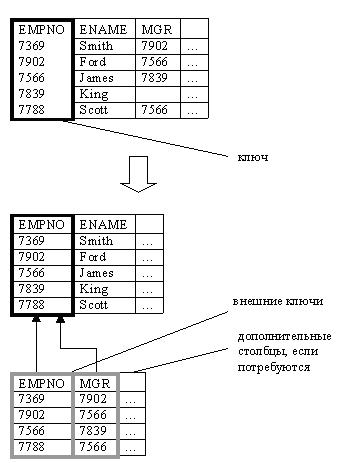
* указание строки, для которой следует вывести предков или потомков в иерархии (необязательная конструкция START WITH);
* указание направления движения по иерархии, вверх или вниз (системная функция *PRIOR*);
* условное выражение, определяющее иерархическую, по мнению программиста, связь между строками.

Последнее, то есть *условное выражение*, определяющее иерархию, приходится всякий раз указывать в запросе явочным порядком, так как ни *SQL*, ни тем более реляционная модель не позволяет задавать иерархическую зависимость между строками при определении таблицы (отношения). Об этом приходится сожалеть; это неизбежное следствие того, что подобная крайне ответственная часть описания данных не хранится в *БД* и вообще не хранится, иначе как в голове у программиста.

*Хранение древовидно зависимых данных в БД*

Наиболее востребованным видом иерархии является дерево. Древовидная зависимость сотрудников в таблице EMP представлена столбцами EMPNO (уникальный "табельный номер сотрудника") и MGR ("табельный номер руководителя сотрудника"). Это именно тот способ задания древовидной зависимости данных, который чаще всего встречается в жизни (классификаторы; устройство сложных агрегатов на производстве; структура организации и так далее).

В жизни, однако, этот способ часто применяется с той поправкой, что ссылка на предка устраняется из основной таблицы, а зависимость "предок — потомок" выносится в особую самостоятельную таблицу:



Достоинств у такого решения два:

1. столбец MGR теперь полностью заполнен и не имеет (единственного!) пропущенного значения; в отдельной таблице, описывающей подчиненность сотрудников, открылась возможность естественным образом указать дополнительные свойства такой подчиненности, как, например, время работы под началом конкретного руководителя или что-нибудь еще.

Есть и недостаток: запрос о сотрудниках теперь придется обращать уже к двум таблицам, и это более затратно. С точки зрения создателей реляционной модели ответственность за это несут разработчики СУБД.

*Примеры запросов по дереву*

Пример употребления фразы CONNECT BY в запросе о сотрудниках:

SELECT ename

FROM emp

CONNECT BY PRIOR empno = mgr

START WITH ename = 'SCOTT'

;

Будут выданы все подчиненные сотрудника SCOTT. Пример показывает использование слова *PRIOR* во фразе CONNECT BY и конструкции START WITH. Слово *PRIOR* (оператор, "системная функция") приписывается ведущему *столбцу* в отношении упорядочения, а не *выражению*, в котором упоминается столбец. Выбор программистом ведущего столбца из пары, участвующей в построении условного выражения, фактически задает направление движения по дереву: поиск родителей или же потомков (как в данном случае).

Ведущий столбец можно употребить и во фразе SELECT; сравните предыдущий пример со следующим:

SELECT ename, PRIOR ename

FROM emp

CONNECT BY PRIOR empno = mgr

START WITH ename = 'SCOTT'

;

Степень отдаленности от начального узла просмотра дерева показывает специальная системная функция без параметров LEVEL ("псевдостолбец", по терминологии Oracle), доступная исключительно в запросах с CONNECT BY:

SELECT LEVEL, ename

FROM emp

CONNECT BY empno = PRIOR mgr

START WITH ename IN ( 'SCOTT', 'ALLEN' )

;

(Все начальники сотрудников SCOTT и ALLEN с указанием уровня подчиненности).

SELECT LEVEL, ename

FROM emp

CONNECT BY PRIOR empno = mgr

;

("Лес" деревьев подчиненности всех сотрудников друг другу; практически этот запрос не очень интересен ввиду появившихся в версии 9 более удобных системных функций).

*Фильтрация узлов дерева*

При необходимости какие-то узлы дерева в выдачу можно не включать. Фильтрацию можно вставить во фразу CONNECT BY или во фразу WHERE. Так как логически CONNECT BY обрабатывается ранее WHERE, результаты фильтрации будут разными.

Пример исключения из списка подчиненных сотрудника SCOTT и всех его потомков:

SELECT ename, PRIOR ename

FROM emp

CONNECT BY PRIOR empno = mgr AND ename <> 'SCOTT'

START WITH mgr IS NULL

;

Пример исключения из списка потомков сотрудника SCOTT:

SELECT ename, PRIOR ename

FROM emp

WHERE ename <> 'SCOTT'

CONNECT BY PRIOR empno = mgr

START WITH mgr IS NULL

;

Заметьте, что в тексте WHERE предшествует CONNECT BY, но это не соответствует порядку обработки.

*Специальные системные функции в предложениях с* CONNECT BY

Специально для запросов по дереву создан ряд системных функций. Один пример — LEVEL — уже приводился. Другой пример — функция SYS\_CONNECT\_BY\_PATH, позволяющая получить для каждой строки ее полный "путь", считая от точки отсчета:

COLUMN epath FORMAT A100

SELECT LEVEL, SYS\_CONNECT\_BY\_PATH ( ename, '/' ) epath

FROM emp

CONNECT BY PRIOR empno = mgr

START WITH mgr IS NULL

;

Список функций, специально предназначенных для употребления в предложениях с CONNECT BY:

|  |  |
| --- | --- |
| *Функция* | *Описание* |
| LEVEL | Номер уровня в дереве (1 — корень, 2 — нижележащий уровень и т. д.) |
| SYS\_CONNECT\_BY\_PATH (столбец, разделитель) | Путь от корня дерева к узлу |
| CONNECT\_BY\_ISCYCLE | 1, если потомок узла является одновременно его предком, иначе 0 |
| CONNECT\_BY\_ISLEAF | 1, если узел не имеет потомков |
| CONNECT\_BY\_ROOT (столбец) или CONNECT\_BY\_ROOT столбец | Значение из строки-корня |
| *PRIOR* (столбец) или *PRIOR* столбец | Значение из строки — прямого предка |

*Упорядочение результата*

Фраза CONNECT BY выдает в результате дерево, но не заботится о порядке перечисления "веток" в пределах одного уровня. Упорядочить по заданному критерию ветви дерева традиционными средствами возможно, но делать это крайне неудобно. С версии 9 задачу много проще решить употреблением специальной фразы ORDER *SIBLINGS* BY:

SELECT LEVEL, SYS\_CONNECT\_BY\_PATH ( ename, '/' ) epath

FROM emp

CONNECT BY PRIOR empno = mgr

START WITH mgr IS NULL

ORDER SIBLINGS BY ename

;

Фразы ORDER BY и ORDER *SIBLINGS* BY в предложении SELECT — взаимоисключающие.

*Обработка зацикливания*

Поскольку Oracle не контролирует корректность иерархической взаимосвязи хранимых строк (БД попросту "не понимает" такой взаимосвязи), ответственность за ее соблюдение ложится на программиста. При изменении данных в БД он может нарушить взаимосвязь, случайно или намеренно. Если такое нарушение приводит к зацикливанию, рекурсивно исполняющаяся фраза CONNECT BY обнаружит это и выдаст ошибку:

SQL> UPDATE emp SET mgr = 7876 WHERE ename = 'JONES';

1 row updated.

У Джонса начальником поставлен Адамс (теперь Адамс → Джонс; здесь стрелка указывает на подчиненного), но тот же Адамс и среди его подчиненных (Джонс → Скотт → Адамс):

SQL> SELECT SYS\_CONNECT\_BY\_PATH ( ename, '/' ) epath

2 FROM emp

3 CONNECT BY PRIOR empno = mgr

4\* START WITH ename = 'JONES'

SQL> ;

ERROR:

ORA-01436: CONNECT BY loop in user data

Указание NOCYCLE заставит Oracle завершить рекурсивный просмотр записей при обнаружении зацикленности и не сообщать об ошибке:

SQL> SELECT SYS\_CONNECT\_BY\_PATH ( ename, '/' ) epath

2 FROM emp

3 CONNECT BY NOCYCLE PRIOR empno = mgr

4\* START WITH ename = 'JONES'

5 /

EPATH

--------------------

/JONES

/JONES/SCOTT

/JONES/SCOTT/ADAMS

/JONES/FORD

/JONES/FORD/SMITH

SQL> ROLLBACK;

В любом случае "бесконечного" выполнения запроса при использовании CONNECT BY не случится.

*Недревовидная иерархия*

Фраза CONNECT BY способна рекурсивно обрабатывать не только древовидно организованные данные, но и иерархию общего вида. Выполним:

CREATE TABLE way (

node VARCHAR2 ( 20 )

, parent VARCHAR2 ( 20 )

, distance NUMBER ( 5 )

);

INSERT INTO way VALUES ( 'Ленинград', 'Москва', 696 );

INSERT INTO way VALUES ( 'Новгород', 'Москва', 538 );

INSERT INTO way VALUES ( 'Ленинград', 'Новгород', 179 );

INSERT INTO way VALUES ( 'Выборг', 'Ленинград', 135 );

COMMIT;

Обратите внимание, что создана не "таблица с расстояниями", а таблица с направленными маршрутами, предоставляющая расстояния между городами с точки зрения Москвы (здесь — единственная вершина иерархии). Такое представление данных и приводимые ниже запросы плохо подходят для решения более общей задачи поиска маршрута между двумя произвольными точками.

Запрос вниз по иерархии от узла 'Москва' (присутствует только в качестве предка):

SQL> COLUMN route FORMAT a40

SQL> SELECT SYS\_CONNECT\_BY\_PATH ( node, '/' ) route

2 FROM way

3 CONNECT BY PRIOR node = parent

4 START WITH parent = 'Москва'

5 ;

ROUTE

---------------------------------------------------

/Ленинград

/Ленинград/Выборг

/Новгород

/Новгород/Ленинград

/Новгород/Ленинград/Выборг

Запрос вверх по иерархии от узла 'Выборг':

SQL> SELECT SYS\_CONNECT\_BY\_PATH ( node, '/' ) route

2 FROM way

3 CONNECT BY node = PRIOR parent

4 START WITH node = 'Выборг'

5 ;

ROUTE

---------------------------------------------------

/Выборг

/Выборг/Ленинград

/Выборг/Ленинград

/Выборг/Ленинград/Новгород

**Обобщенное табличное выражение, Oracle PL/SQL оператор** **WITH**

**Обобщенное табличное выражение** (OTB) оно же **Common Table Expression** (CTE) – это временный результат выполнения SQL подзапроса, который не сохраняется в БД, но к которому можно обращаться внутри запроса.

Длячего предназначены ОТВ:

* для написания рекурсивных запросов;
* для замены представлений (VIEW), когда нет необходимости сохранять в базе SQL запрос определяющий представления;

Обобщенные табличные выражения повышают читаемость кода путем разделения запроса на модули, и тем самым упрощают работу со сложными запросами. OTB допускают многократные ссылки на результирующие данные из одной SQL инструкции. ОТВ определяются с помощью **конструкции WITH** размещённой как в запросах, так и в функциях, хранимых процедурах, триггерах и представлениях.

**Описание****WITH**

Oracle PL/SQL оператор WITH позволяет дать блоку подзапроса имя/псевдоним, на которое можно ссылаться в нескольких местах основного SQL-запроса. Имя, присвоенное подзапросу, обрабатывается так, как если бы оно было встроенным представлением или таблицей. SQL оператор WITH по сути является заменой обычному подзапросу.

*Синтаксис*

Синтаксис Oracle PL/SQL WITH с одним подзапросом:

WITH query\_name AS (SELECT expressions FROM table\_A)

SELECT column\_list

FROM query\_name [,table\_name]

[WHERE conditions]

или синтаксис Oracle PL/SQL WITH с несколькими подзапросами:

WITH query\_name\_A AS

(SELECT expressions FROM table\_A),

query\_name\_B AS

([SELECT expressions FROM query\_name\_A] | [SELECT expressions FROM table\_B])

SELECT column\_list

FROM query\_name\_A, query\_name\_B [,table\_X | Join table\_Z]

[WHERE conditions]

expressions — поля или расчеты подзапроса.

column\_list — поля или расчеты основного запроса.

table\_A, table\_B, table\_X, table\_Z — таблицы или соединения для подзапросов.

query\_name\_A, query\_name\_B — псевдоним подзапроса. Если подзапросов несколько, то они перечисляются через запятую.

WHERE conditions — условия которые должны быть выполнены для основных запросов.

*Примечание*

* Формально предложение WITH называется факторингом подзапроса.
* Предложение SQL WITH используется, когда подзапрос выполняется несколько раз.
* Подзапросы в WITH перечисляются через запятую.

*Применение*

* SQL-предложение WITH было введено Oracle в базе данных Oracle 9i выпуск 2.
* Начиная с Oracle Database 12c Release 1 (12.1) в операторе WITH можно определять функции и процедуры.

Рассматриваемые примеры Oracle PL/SQL оператор WITH

* Пример WITH с одним подзапросом
* Пример WITH с двумя подзапросами
* Пример WITH с функцией
* Пример WITH с процедурой
* Пример PRAGMA UDF

Для примера использования оператор WITH, с одним и двумя подзапросами, создадим несколько таблиц с данными.

--удалим таблицы из базы данных, если существуют

-- DROP TABLE EMP PURGE;

-- DROP TABLE DEPT PURGE;

--Создадим две таблицы EMP и DEPT:

CREATE TABLE DEPT (

DEPTNO NUMBER(2) CONSTRAINT PK\_DEPT PRIMARY KEY,

DNAME VARCHAR2(14),

CITY VARCHAR2(15)

) ;

CREATE TABLE EMP (

EMPNO NUMBER(4) CONSTRAINT PK\_EMP PRIMARY KEY,

ENAME VARCHAR2(10),

JOB VARCHAR2(9),

MGR NUMBER(4),

HIREDATE DATE,

SAL NUMBER(7,2),

COMM NUMBER(7,2),

DEPTNO NUMBER(2) CONSTRAINT FK\_DEPTNO REFERENCES DEPT

);

--И добавим в таблицы EMP и DEPT следующие данные:

--DEPT

INSERT INTO DEPT VALUES (10,'ACCOUNTING','NEW YORK');

INSERT INTO DEPT VALUES (20,'RESEARCH','DALLAS');

INSERT INTO DEPT VALUES (30,'SALES','CHICAGO');

INSERT INTO DEPT VALUES (40,'OPERATIONS','BOSTON');

--EMP

INSERT INTO EMP VALUES (7369,'SMITH','CLERK',7902,to\_date('17.12.1980','dd.mm.yyyy'),800,NULL,20);

INSERT INTO EMP VALUES (7499,'ALLEN','SALESMAN',7698,to\_date('20.2.1981','dd.mm.yyyy'),1600,300,30);

INSERT INTO EMP VALUES (7521,'WARD','SALESMAN',7698,to\_date('22.2.1981','dd.mm.yyyy'),1250,500,30);

INSERT INTO EMP VALUES (7566,'JONES','MANAGER',7839,to\_date('2.4.1981','dd.mm.yyyy'),2975,NULL,20);

INSERT INTO EMP VALUES (7654,'MARTIN','SALESMAN',7698,to\_date('28.9.1981','dd.mm.yyyy'),1250,1400,30);

INSERT INTO EMP VALUES (7698,'BLAKE','MANAGER',7839,to\_date('1.5.1981','dd.mm.yyyy'),2850,NULL,30);

INSERT INTO EMP VALUES (7782,'CLARK','MANAGER',7839,to\_date('9.6.1981','dd.mm.yyyy'),2450,NULL,10);

INSERT INTO EMP VALUES (7788,'SCOTT','ANALYST',7566,to\_date('13.7.87','dd.mm.rr').85,3000,NULL,20);

INSERT INTO EMP VALUES (7839,'KING','PRESIDENT',NULL,to\_date('17.11.1981','dd.mm.yyyy'),5000,NULL,10);

INSERT INTO EMP VALUES (7844,'TURNER','SALESMAN',7698,to\_date('8.9.1981','dd.mm.yyyy'),1500,0,30);

INSERT INTO EMP VALUES (7876,'ADAMS','CLERK',7788,to\_date('13.7.87', 'dd.mm.rr').51,1100,NULL,20);

INSERT INTO EMP VALUES (7900,'JAMES','CLERK',7698,to\_date('3.12.1981','dd.mm.yyyy'),950,NULL,30);

INSERT INTO EMP VALUES (7902,'FORD','ANALYST',7566,to\_date('3.12.1981','dd.mm.yyyy'),3000,NULL,20);

INSERT INTO EMP VALUES (7934,'MILLER','CLERK',7782,to\_date('23.1.1982','dd.mm.yyyy'),1300,NULL,10);

COMMIT;

Пример WITH с одним подзапросом

Рассмотрим пример sql оператора with c одним подзапросом, чтобы понять, как использовать оператор with в Oracle PL/SQL.

Например:

WITH employee AS (SELECT \* FROM emp)

SELECT \* FROM employee WHERE empno > 7900

UNION ALL

SELECT \* FROM employee WHERE sal > 3000

В этом примере мы создали подзапрос с псевдонимом employee из таблицы Employees (сотрудников). Затем с помощью двух основных запросов и оператора UNION ALL запросили данные всех сотрудников у которых empno > 7900 и всех сотрудников у которых sal > 3000.

В следующем примере мы хотим узнать сколько людей в отделе для каждого сотрудника.

WITH dept\_count AS (

SELECT deptno, COUNT(\*) AS dept\_count

FROM emp

GROUP BY deptno)

SELECT e.ename AS employee\_name,

dc.dept\_count AS emp\_dept\_count

FROM emp e

JOIN dept\_count dc ON e.deptno = dc.deptno;

*Пример WITH с двумя подзапросами*

Для примера использования оператор WITH, создадим несколько таблиц с данными.

Например, нам может понадобиться выбрать отделы с заработной платой выше среднего. Для этого сначала в первом подзапросе dept\_costs определим сумму зарплат по отделам. Затем во втором подзапросе avg\_cost определим среднюю зарплату по отделам.

WITH

dept\_costs AS (

SELECT dname, SUM(sal) dept\_total

FROM emp e, dept d

WHERE e.deptno = d.deptno

GROUP BY dname

),

avg\_cost AS (

SELECT SUM(dept\_total)/COUNT(\*) avg

FROM dept\_costs

)

SELECT \*

FROM dept\_costs

WHERE dept\_total > (SELECT avg FROM avg\_cost)

ORDER BY dname;

**Пример WITH с функцией**

Для примеров оператора WITH необходимо создать следующую тестовую таблицу.

DROP TABLE t1 PURGE;

CREATE TABLE t1 AS

SELECT 1 AS id

FROM dual

CONNECT BY level <= 1000000;

-- Сбор статистики на CTAS больше не требуется в 12c,

-- при условии, что запрос выдается не-SYS пользователем.

-- EXEC DBMS\_STATS.gather\_table\_stats (USER, 't1');

В этом операторе WITH раздел объявления может использоваться для определения функций PL/SQL, как показано ниже.

WITH

FUNCTION with\_function(p\_id IN NUMBER) RETURN NUMBER IS

BEGIN

RETURN p\_id;

END;

SELECT with\_function(id)

FROM t1

WHERE rownum = 1

--результат

WITH\_FUNCTION(ID)

-----------------

1

С точки зрения разрешения имен функций, определенных в разделе объявлений PL/SQL оператора WITH, имеют приоритет над объектами с тем же именем, определенным на уровне схемы.

Пример WITH с процедурой

Мы также можем определить процедуры в разделе объявлений оператора WITH, даже если они не используются.

WITH

PROCEDURE with\_procedure(p\_id IN NUMBER) IS

BEGIN

DBMS\_OUTPUT.put\_line('p\_id=' || p\_id);

END;

SELECT id

FROM t1

WHERE rownum = 1

--результат

ID

----------

1

В действительности вы поместили бы процедуру в операторе WITH, только если бы планировали вызывать процедуру из функции в разделе объявлений.

WITH

PROCEDURE with\_procedure(p\_id IN NUMBER) IS

BEGIN

DBMS\_OUTPUT.put\_line('p\_id=' || p\_id);

END;

FUNCTION with\_function(p\_id IN NUMBER) RETURN NUMBER IS

BEGIN

with\_procedure(p\_id);

RETURN p\_id;

END;

SELECT with\_function(id)

FROM t1

WHERE rownum = 1

--результат

WITH\_FUNCTION(ID)

-----------------

1

p\_id=1

Похоже, что эта функция не поддерживается PL/SQL. Любая попытка ее использования приводит к ошибкам компиляции, как показано ниже.

BEGIN

FOR cur\_rec IN (WITH

FUNCTION with\_function(p\_id IN NUMBER) RETURN NUMBER IS

BEGIN

RETURN p\_id;

END;

SELECT with\_function(id)

FROM t1

WHERE rownum = 1)

LOOP

NULL;

END LOOP;

END;

FUNCTION with\_function(p\_id IN NUMBER) RETURN NUMBER IS

\*

ERROR at line 3:

ORA-06550: line 3, column 30:

PL/SQL: ORA-00905: missing keyword

ORA-06550: line 2, column 19:

PL/SQL: SQL Statement ignored

ORA-06550: line 5, column 34:

PLS-00103: Encountered the symbol ";" when expecting one of the following:

loop

Использование динамического SQL позволяет обойти это ограничение.

SET SERVEROUTPUT ON

DECLARE

l\_sql VARCHAR2(32767);

l\_cursor SYS\_REFCURSOR;

l\_value NUMBER;

BEGIN

l\_sql := 'WITH

FUNCTION with\_function(p\_id IN NUMBER) RETURN NUMBER IS

BEGIN

RETURN p\_id;

END;

SELECT with\_function(id)

FROM t1

WHERE rownum = 1';

OPEN l\_cursor FOR l\_sql;

FETCH l\_cursor INTO l\_value;

DBMS\_OUTPUT.put\_line('l\_value=' || l\_value);

CLOSE l\_cursor;

END;

l\_value=1

PL/SQL procedure successfully completed.

Поддержка этой функции с использованием статического SQL внутри PL/SQL ожидается в следующем релизе Oracle.

ПРЕИМУЩЕСТВА ПРОИЗВОДИТЕЛЬНОСТИ

Вся причина определения встроенного кода PL/SQL заключается в повышении производительности.

Создайте обычную функцию для использования в качестве сравнения.

CREATE OR REPLACE FUNCTION normal\_function(p\_id IN NUMBER) RETURN NUMBER IS

BEGIN

RETURN p\_id;

END;

Запустите следующий тест, который измеряет затраченное время и загрузку ЦП запроса, используя определение встроенной функции.

DECLARE

l\_time PLS\_INTEGER;

l\_cpu PLS\_INTEGER;

l\_sql VARCHAR2(32767);

l\_cursor SYS\_REFCURSOR;

TYPE t\_tab IS TABLE OF NUMBER;

l\_tab t\_tab;

BEGIN

l\_time := DBMS\_UTILITY.get\_time;

l\_cpu := DBMS\_UTILITY.get\_cpu\_time;

l\_sql := 'WITH

FUNCTION with\_function(p\_id IN NUMBER) RETURN NUMBER IS

BEGIN

RETURN p\_id;

END;

SELECT with\_function(id)

FROM t1';

OPEN l\_cursor FOR l\_sql;

FETCH l\_cursor

BULK COLLECT INTO l\_tab;

CLOSE l\_cursor;

DBMS\_OUTPUT.put\_line('WITH\_FUNCTION : ' ||

'Time=' || TO\_CHAR(DBMS\_UTILITY.get\_time - l\_time) || ' hsecs ' ||

'CPU Time=' || (DBMS\_UTILITY.get\_cpu\_time - l\_cpu) || ' hsecs ');

l\_time := DBMS\_UTILITY.get\_time;

l\_cpu := DBMS\_UTILITY.get\_cpu\_time;

l\_sql := 'SELECT normal\_function(id)

FROM t1';

OPEN l\_cursor FOR l\_sql;

FETCH l\_cursor

BULK COLLECT INTO l\_tab;

CLOSE l\_cursor;

DBMS\_OUTPUT.put\_line('NORMAL\_FUNCTION: ' ||

'Time=' || TO\_CHAR(DBMS\_UTILITY.get\_time - l\_time) || ' hsecs ' ||

'CPU Time=' || (DBMS\_UTILITY.get\_cpu\_time - l\_cpu) || ' hsecs ');

END;

WITH\_FUNCTION : Time=45 hsecs CPU Time=39 hsecs

NORMAL\_FUNCTION: Time=129 hsecs CPU Time=113 hsecs

PL/SQL procedure successfully completed.

Из этого мы видим, что определение встроенной функции занимает приблизительно одну треть затраченного времени и времени процессора для завершения.

**Пример PRAGMA UDF**

В ряде презентаций, предшествовавших официальному выпуску 12c, выступавшие упоминали PRAGMA UDF(User Defined Function), которая предположительно дает вам преимущества производительности встроенного PL/SQL, в то же время позволяя вам определять объект PL/SQL вне оператора SQL. Следующий код переопределяет предыдущую обычную функцию для использования этой прагмы.

CREATE OR REPLACE FUNCTION normal\_function(p\_id IN NUMBER) RETURN NUMBER IS

PRAGMA UDF;

BEGIN

RETURN p\_id;

END;

Как только функция скомпилирована, выполнение теста из предыдущего раздела для этой функции дает довольно интересные результаты.

SET SERVEROUTPUT ON

DECLARE

l\_time PLS\_INTEGER;

l\_cpu PLS\_INTEGER;

l\_sql VARCHAR2(32767);

l\_cursor SYS\_REFCURSOR;

TYPE t\_tab IS TABLE OF NUMBER;

l\_tab t\_tab;

BEGIN

l\_time := DBMS\_UTILITY.get\_time;

l\_cpu := DBMS\_UTILITY.get\_cpu\_time;

l\_sql := 'WITH

FUNCTION with\_function(p\_id IN NUMBER) RETURN NUMBER IS

BEGIN

RETURN p\_id;

END;

SELECT with\_function(id)

FROM t1';

OPEN l\_cursor FOR l\_sql;

FETCH l\_cursor

BULK COLLECT INTO l\_tab;

CLOSE l\_cursor;

DBMS\_OUTPUT.put\_line('WITH\_FUNCTION : ' ||

'Time=' || TO\_CHAR(DBMS\_UTILITY.get\_time - l\_time) || ' hsecs ' ||

'CPU Time=' || (DBMS\_UTILITY.get\_cpu\_time - l\_cpu) || ' hsecs ');

l\_time := DBMS\_UTILITY.get\_time;

l\_cpu := DBMS\_UTILITY.get\_cpu\_time;

l\_sql := 'SELECT normal\_function(id)

FROM t1';

OPEN l\_cursor FOR l\_sql;

FETCH l\_cursor

BULK COLLECT INTO l\_tab;

CLOSE l\_cursor;

DBMS\_OUTPUT.put\_line('NORMAL\_FUNCTION: ' ||

'Time=' || TO\_CHAR(DBMS\_UTILITY.get\_time - l\_time) || ' hsecs ' ||

'CPU Time=' || (DBMS\_UTILITY.get\_cpu\_time - l\_cpu) || ' hsecs ');

END;

WITH\_FUNCTION : Time=44 hsecs CPU Time=40 hsecs

NORMAL\_FUNCTION: Time=33 hsecs CPU Time=29 hsecs

PL/SQL procedure successfully completed.

Кажется, что автономная функция, использующая PRAGMA UDF, последовательно выполняет встроенную функцию.

У меня сложилось впечатление, что вызов функции, определенной с помощью PRAGMA UDF напрямую из PL / SQL, не удастся. Это не похоже на случайность.

Oracle PL/SQL

DECLARE

l\_number NUMBER;

BEGIN

l\_number := normal\_function(1);

END;

PL/SQL procedure successfully completed.