

Otwarty interfejs programistyczny w Oracle 10g – tworzenie własnych funkcji agregujących, tabelowych oraz indeksów

Bartłomiej Jabłoński

Uniwersytet Łódzki

e-mail: bartek@math.uni.lodz.pl

Streszczenie

Oracle 10g udostępnia API, w którym programiści PL/SQL-a, Javy lub C mogą rozszerzać możliwości serwera. Tworzone w tych językach moduły włączane są w motor bazy danych, a ustalony z góry interfejs pozwala na automatyczne wywołanie przez serwer procedur odpowiedzialnych za realizację zdarzeń wynikających z bieżącej sytuacji.

Funkcje agregujące – programista odpowiada za utworzenie procedur obsługi zdarzeń odpowiadających za inicjalizację środowiska, kolejne iteracje oraz za przygotowanie wyniku. Utworzone w ten sposób funkcje mogą być wykorzystywane również w zapytaniach równoległych oraz zaawansowanych opcjach analitycznych OLAP.

Funkcje tabelowe – wraz z udostępnieniem interfejsu funkcji tabelowych możliwe jest generowanie zawartości „tabeli” w sposób programowy. Mechanizm ten pozwala na przykład na skomplikowane łączenie danych z wielu tabel, przekształcanie ich, lub wykonywanie operacji niedostępnych z poziomu języka SQL. Oracle umożliwia tworzenie funkcji tabelowych za pomocą uproszczonego interfejsu PL/SQL oraz interfejsu zaawansowanego dostępnego z poziomu języków PL/SQL, Java lub C.

Indeksy własne – w Oracle 10g występują dwa rodzaje indeksów: B-drzewa oraz indeksy bitmapowe. Wystarczają one w znakomitej większości przypadków optymalizacji dostępu do danych relacyjnych. Istnieje jednak cała gama danych o strukturze mało uporządkowanej, dla których techniki indeksowania wykorzystują inne mechanizmy. Oracle pozwala na własną implementację takich algorytmów – moduł, w którym programista realizuje podstawowe operacje tworzenia, kasowania indeksu lub wstawiania, poprawiania, kasowania i wyboru rekordu(-ów) może być dołączony do serwera bazy danych i automatycznie przez niego wykorzystywany. Na szczególną uwagę zasługuje fakt, że moduł ten można wzbogacić o funkcje liczące statystyki, co pozwala na uwzględnienie specyfiki własnego indeksu w procesie optymalizacji planu wykonania zapytania.

Informacja o autorze

Bartłomiej Jabłoński od 1999 roku pracuje na stanowisku asystenta na Wydziale Matematyki Uniwersytetu Łódzkiego, gdzie zajmuje się głównie bazami danych Oracle oraz standardem XML ze szczególnym uwzględnieniem jego zastosowań w bazach danych. Od 1994 roku wielokrotnie uczestniczył w projektach informatycznych związanych z bazami danych Oracle jako konsultant. Od 1995 prowadzi szkolenia w Centrum Edukacyjnym Oracle Polska. Jest autorem 3 i tłumaczem 4 książek z zakresu informatyki.

1. Wstęp

1.1. Rozszerzanie możliwości systemów

Najczęściej kupując wybrany (drogi) system, program, bądź narzędzie do tworzenia aplikacji oczekujemy, że produkt ten spełni wszystkie nasze oczekiwania i wymagania. Gorzej, gdy takich „oczekujących” jest więcej, a wymagania wzajemnie sprzeczne. Bardzo często zdarza się, że problem jest jednostkowy i specyficzny tylko dla danego użytkownika – dołączanie rozwiązania do pakietu, za który każdy inny użytkownik musiałby dopłacić, mogłoby w efekcie spowodować wzrost ceny i spadek popularności produktu.

Wyjściem z tej sytuacji, jest umożliwienie użytkownikowi samodzielnej rozbudowy produktu w oparciu o narzędzia tego samego producenta. W różnych rodzajach aplikacji można spotkać rozmaite rozwiązania. Na przykład w Borland Delphi jest to system wizualnych komponentów lub bibliotek kodu, w Microsoft Office język Basic, w którym można pisać nawet bardzo skomplikowane makra. W wielu produktach dostępny jest nawet kod źródłowy (*ang. open source*) i każdy może go sobie zmodyfikować lub rozwinąć według własnych potrzeb.

Firma Oracle w swoim wiodącym produkcie – serwerze bazy danych – również udostępniła wiele rozwiązań umożliwiających umieszczanie własnego kodu i ścisłą integrację z serwerem.

2. Możliwości rozszerzania w Oracle

Możliwości te zaczęły się w 1988 roku wraz z wersją Oracle 6, gdzie po raz pierwszy pojawił się język programowania PL/SQL. Wystarczy napisać funkcję spełniającą kilka „oczywistych” kryteriów i można z niej korzystać w poleceniach SQL, tym samym rozszerzając możliwości tego języka. Możliwe stało się także używanie innych języków programowania, których kompilatory serwer potrafi uruchomić (*ang. user-exit*). Jednak prawdziwe rozszerzenie możliwości uzyskujemy dopiero poprzez zastosowanie funkcji zwrotnych (*ang. call-back*).

2.2. Funkcje zwrotne

Funkcją zwrotną nazywamy kod, który pisany jest z intencją wykonywania podczas zajścia określonego zdarzenia (inicjowanego przez zajście sytuacji losowej lub przez samego programistę) – mówimy wtedy o obsłudze tego zdarzenia. Aby serwer mógł wywoływać taką funkcję, musi oczywiście mieć pojęcie o jej istnieniu – funkcję zwrotną musimy wcześniej umieścić w określonym środowisku i zarejestrować.

Przykład. Chcemy, aby każdy wpisany rekord do tabeli *employees* zostawił ślad w tabeli dziennika. Rozwiązaniem jest napisanie procedury, która co jakiś czas przeszukuje tabelę *employees* w poszukiwaniu nowych rekordów, bądź jest automatycznie wywoływana przez serwer w momencie wystąpienia zdarzenia wstawienia nowego rekordu. Pierwsze rozwiązanie wymaga częstego przeszukiwania tabeli, co jak można się domyśleć, nie jest skuteczne. Drugie rozwiązanie opiera się na założeniu, że serwer posiada funkcjonalność wywoływania funkcji użytkownika w określonych warunkach. Oczywiście praktykujący programiści znajdą właściwe rozwiązanie bez trudu – jest to wykorzystanie mechanizmu wyzwalacza (*ang. trigger*), który, w naszej terminologii, jest po prostu rejestracją pewnej procedury i określeniem warunków jej automatycznego wywoływania.

W artykule tym omówione zostaną trzy sytuacje, w których rozwiązanie polega na napisaniu kodu obsługującego określone zdarzenia. Wszystkie wymienione w artykule przykłady są zgodne z wersją Oracle 10g i wykorzystują standardowe schematy demonstracyjne Oracle. W celu uprosz-

czenia kodu i zwrócenia uwagi na omawiane meritum zrezygnowano z obsługi błędów i optymalizację czasu działania.

2.3. Mechanizm generowania i obsługi zdarzeń

W celu lepszego zobrazowania techniki wykorzystywania funkcji zwrotnych spróbujemy przeanalizować mechanizm zdarzeń na przykładzie funkcji liczącej średnią arytmetyczną, na próbie z której odrzuca się skrajne wyniki¹. Jeżeli założyć, że taka funkcja istnieje (nazwijmy ją LimAvg), to pytanie o średnią (w nowym sensie) pensję w firmie wgląda wtedy tak:

```
SELECT LimAvg(salary)
FROM employees;2
```

Ponieważ takiej funkcji nie ma w SQL-u spróbujmy pierwsze podejście wykonać za pomocą procedury napisanej w PL/SQL-u:

```
create or replace procedure limavg_proc is
  v_max number := null;          /* Inicjalizacja */
  c_max number := 0;            /*          */
  v_min number := null;        /*          */
  c_min number := 0;            /*          */
  vsum number := 0;            /*          */
  cnt number := 0;             /* Inicjalizacja */
begin
  for rec in (SELECT salary FROM employees) loop
    if rec.salary < v_max then null; else          /* Iteracja */
      if rec.salary = v_max then                  /*          */
        c_max := c_max + 1;                       /*          */
      else                                        /*          */
        v_max := rec.salary;                       /*          */
        c_max := 1;                               /*          */
      end if;                                     /*          */
    end if;                                     /*          */
    if rec.salary > v_min then null; else          /*          */
      if rec.salary = v_min then                  /*          */
        c_min := c_min + 1;                       /*          */
      else                                        /*          */
        v_min := rec.salary;                       /*          */
        c_min := 1;                               /*          */
      end if;                                     /*          */
    end if;                                     /*          */
    vsum := vsum + rec.salary;                    /*          */
    cnt := cnt + 1;                              /* Iteracja */
  end loop;
  if cnt > c_max + c_min and cnt > 0 then          /* Wynik */
    dbms_output.put_line('Średnia wynosi: '      /*          */
      || (vsum - c_max * v_max - c_min * v_min) /*          */
      / (cnt - c_max - c_min));                  /*          */
  else                                           /*          */

```

¹ Zwykła średnia arytmetyczna jest bardzo czuła na skrajne wyniki, co łatwo sprawdzić licząc średnią pensję w firmie po tym, jak i w tabeli EMPLOYEES damy prezesowi \$100000 podwyżkę :)

² Oczywiście realizowany tu problem można wykonać też „tradycyjną” metodą:

```
SELECT avg(salary)
FROM employees
WHERE salary NOT IN (SELECT min(salary)
                    FROM employees
                    UNION ALL SELECT max(salary)
                    FROM employees);
```

```

        dbms_output.put_line('Brak wystarczającej ilości danych!');
    end if;
end limavg_proc;
/

```

Patrząc na tę procedurę można zauważyć trzy istotne cechy:

1. W kodzie da się wyraźnie wyróżnić trzy części:

- inicjalizacja zmiennych lokalnych,
- przebieg pętli, w której odczytywane są dane i modyfikowane zmienne lokalne,
- formowanie wyniku, który zależy już tylko wyłącznie od zmiennych lokalnych.

2. Na potrzeby wyliczenia wartości funkcji wymagane jest istnienie lokalnego segmentu danych (zmiennych), w których przechowywane są wartości potrzebne później do wyliczenia końcowego rezultatu funkcji.

3. Dane (rekordy) odczytywane są jeden(!) raz – rezultat funkcji wyliczony jest jednorazowo.

Cechy te zostaną wykorzystane później w programowaniu zdarzeniowym, w którym okaże się, że utworzenie odpowiedniego typu obiektowego pozwala na ich zachowanie.

Analizując kod PL/SQL nie sposób jednak nie zauważyć również pewnych wad:

1. Kod przystosowany jest do liczenia średniej tylko na zbiorze danych określonych w jednej kolumnie, bez możliwości wprowadzania dodatkowych warunków WHERE lub zapytań zagnieżdżonych.
2. Liczenie odbywa się nierównolegle, a zatem przeważnie gorzej niż zapytania wykonywane z opcją PARALLEL.
3. Otrzymany rezultat trudno jest wykorzystać w kolejnych zapytaniach SQL.

Częściowo mankamenty te można zniwelować poprzez wykorzystanie natywnego SQL-a (polecenie EXECUTE IMMEDIATE) lub zapytań przekazywanych poprzez typ REF CURSOR. Prowadzi to jednak do znacznego skomplikowania kodu. Nie da się ukryć, że zaprezentowane w dalszej części artykułu rozwiązanie jest dużo bardziej elastyczne i eleganckie.

Zgodnie z wnioskami z poprzedniego podrozdziału, implementując funkcję agregującą w Oracle należy ustanowić odrębny komplet zmiennych dla realizacji każdego wyliczenia oraz kod realizujący trzy wymienione wcześniej etapy: inicjalizacji, iteracji i finalizacji. Enkapsulację zmiennych i kodu uzyskuje się poprzez utworzenie typu obiektowego, którego instancja tworzona jest na bieżąco przez serwer Oracle w momencie liczenia średniej.

Na koniec pozostaje tylko umożliwić użycie utworzonego typu w zapytaniu SQL. Do tego celu wystarczy skonstruować prostą funkcję PL/SQL bez kodu (kompletny przykład znajduje się w dalszej części artykułu).

2.4. Obszary zastosowań

Opisany tu mechanizm wykorzystuje się do implementacji:

- funkcji agregujących
- funkcji tabelowych (zwracających dane, które można wykorzystywać np. w klauzuli FROM zapytania)
- indeksów własnych

W każdym z w/w przypadków należy utworzyć typ obiektowy oraz funkcję/indeks wskazujący na ten typ. Wyjątkowo dla funkcji tabelowych istnieje dodatkowy uproszczony interfejs, pozwalający implementować prostsze przypadki implementować bez konieczności znajomości technik programowania obiektowego. Każda z tych funkcji/indeksów wymaga oczywiście implementacji innych zdarzeń, które opisane są w dalszej części tego artykułu. Specyfikacja tych zdarzeń to inna część API (*ang. Application Programming Interface*), znaleźć ją można np. w [OraDok].

3. Funkcje agregujące

Funkcje agregujące wyliczają jakąś wartość statystyczną dla grupy rekordów. Jako argument pobierają zbiór rekordów przygotowany wcześniej ze względu na jakieś kryterium i zwracają jedną wartość³. Oracle udostępnia użytkownikom możliwość własnego definiowania takich statystyk. Większość pracy wykonuje serwer (odczyt rekordów, grupowanie, pobieranie kolejnych rekordów do analizy) - użytkownikowi pozostaje napisać tylko trzy etapowy (inicjalizacja, iteracja, wynik) algorytm liczenia. Kod użytkownika uruchamiany jest automatycznie w odpowiedzi na zdarzenie polegające na wykonywaniu zapytania, w którym użyta została taka funkcja. W miarę realizacji zapytania, serwer generuje własne zdarzenia, których obsługa zdefiniowana jest w procedurach przygotowanych przez użytkownika. Ponieważ, zgodnie z wcześniejszymi ustaleniami, oprócz kodu potrzebny jest też właściwy dla danego zapytania segment danych, procedury te zgrupowane są w postaci metod typu obiektowego.

Typ obiektowy realizujący funkcję agregującą musi posiadać przynajmniej jedno pole⁴ oraz szereg poniższych metod odpowiedzialnych za realizację algorytmu liczenia statystyki na określonym etapie zaawansowania prac.

Oto zdarzenia generowane przez serwer:

ODCIAggregateInitialize – Inicjalizacja zmiennych lokalnych. Najczęściej wykorzystuje się to zdarzenie do zerowania liczników, inicjalizacji kolekcji lub innych, złożonych typów danych, do odczytania parametrów z tablic bazy, itp. Zdarzenie zachodzi dla każdej grupy powstałej w wyniku zastosowania klauzuli GROUP BY.

ODCIAggregateIterate – Iteracja, zdarzenie generowane dla każdego rekordu. Najczęściej wykorzystywane jest do powiększenia liczników.

ODCIAggregateTerminate – Finalizacja obliczeń, formowanie wyniku, zwolnienie zasobów.

ODCIAggregateMerge – Zdarzenie wykorzystywane do łączenia liczników pochodzących z tej samej grupy, powstałych w wyniku równoległego wykonywania zapytania.

Ostatnie zdarzenie wymaga pewnego komentarza. Otóż oceniając kod procedury z rozdziału 2.1 jako jedną z wad wymieniono brak wykonywania obliczeń w sposób równoległy. Równoległe wykonywanie zapytań jest jedną z technik optymalizacyjnych, która polega na efektywniejszym wykorzystywaniu zasobów serwera: wolnego twardego dysku i szybkiego procesora. Polega ona na podziale zadania na kilka części i realizacji ich przez oddzielne procesy w tym samym czasie. Nad właściwym, nie wpływającym na wynik, podziale, czuwa serwer. Jednak to użytkownik musi scalić wyniki pośrednie w wynik ostateczny. W dostępnym modelu odbywa się to wieloetapowo: najpierw scalane są wyniki dwóch procesów, rezultat scalany jest z wynikiem cząstkowym kolejnego procesu i tak, aż do wyczerpania wszystkich procesów uczestniczących w obliczeniach.

Przykład.

³ Może to być typ skalarny (np. liczba) lub złożony (np. obiekt)

⁴ Jest to wymaganie Oracle dla typu obiektowego.

Poniżej przedstawiono implementację funkcji opisanej w rozdziale 2.1. Najpierw tworzymy interfejs typu obiektowego, w którym zawarte muszą być wszystkie metody zdefiniowane poprzez API:

```
create or replace type TLimAvg as object
(
  v_max number,
  c_max number,
  v_min number,
  c_min number,
  vsum number,
  cnt number,
  static function ODCIAggregateInitialize(sctx IN OUT TLimAvg) return number,
  member function ODCIAggregateIterate(self IN OUT TLimAvg,
                                          value IN number) return number,
  member function ODCIAggregateTerminate(self IN TLimAvg,
                                           returnValue OUT number, flags IN number) return number,
  member function ODCIAggregateMerge(self IN OUT TLimAvg,
                                       ctx2 IN TLimAvg) return number
);
/
```

Następnie tworzymy ciało typu, gdzie zawarta jest implementacja funkcji reagujących na zdarzenia. Należy zwrócić uwagę na podobieństwo kodu do kodu z rozdziału 2.1.

```
create or replace type body TLimAvg is
static function ODCIAggregateInitialize(sctx IN OUT TLimAvg) return number is
begin
  sctx := TLimAvg(null, 0, null, 0, 0, 0);
  return ODCIConst.Success;
end ODCIAggregateInitialize;
```

W części inicjalizującej metoda tworzy wystąpienie obiektu. W wywołaniu konstruktora zerowane są pola tworzonego obiektu. Metoda *ODCIAggregateInitialize* wywoływana jest dla każdej grupy rekordów na początku obliczeń.

```
member function ODCIAggregateIterate(self IN OUT TLimAvg, value IN number) return number is
begin
  if value < v_max then null; else
    if value = v_max then
      c_max := c_max + 1;
    else
      v_max := value;
      c_max := 1;
    end if;
  end if;
  if value > v_min then null; else
    if value = v_min then
      c_min := c_min + 1;
    else
      v_min := value;
      c_min := 1;
    end if;
  end if;
  vsum := vsum + value;
  cnt := cnt + 1;
  return ODCIConst.Success;
end ODCIAggregateIterate;
```

Metoda *ODCIAggregateIterate* wywoływana jest dla każdego rekordu w grupie. Pozwala ona na odczytanie danych tego rekordu, które mają wpływ na naszą średnią.

```
member function ODCIAggregateTerminate(self IN TLimAvg, returnValue OUT number, flags IN number) return number is
begin
  if cnt > c_max + c_min and cnt > 0 then
    returnvalue :=
      (vsum - c_max * v_max - c_min * v_min) /
      (cnt - c_max - c_min);
    return ODCIConst.Success;
  else
    returnvalue := 0;
    return ODCIConst.Error;
  end if;
end ODCIAggregateTerminate;
```

Metoda *ODCIAggregateTerminate* formułuje wynik. Liczony jest on na podstawie liczników aktualizowanych w *ODCIAggregateIterate*.

```
member function ODCIAggregateMerge(self IN OUT TLimAvg, ctx2 IN TLimAvg) return number is
begin
  if self.v_max = ctx2.v_max then
    self.c_max := self.c_max + ctx2.c_max;
  elsif self.v_max < ctx2.v_max then
    self.v_max := ctx2.v_max;
    self.c_max := ctx2.c_max;
  end if;
  if self.v_min = ctx2.v_min then
    self.c_min := self.c_min + ctx2.c_min;
  elsif self.v_min > ctx2.v_min then
    self.v_min := ctx2.v_min;
    self.c_min := ctx2.c_min;
  end if;
  self.vsum := self.vsum + ctx2.vsum;
  self.cnt := self.cnt + ctx2.cnt;
  return ODCIConst.Success;
end ODCIAggregateMerge;
end TLimAvg;
/
```

Metoda *ODCIAggregateMerge* łączy dwa wyniki realizowane przez dwa procesy równoległe przeglądające tę samą grupę rekordów. Dzięki temu zapytania wykorzystujące tę funkcję będą mogły korzystać z opcji PARALLEL.

Na kilka aspektów warto teraz zwrócić uwagę, gdyż powtarzają się one również w pozostałych zastosowaniach: Po pierwsze, niektóre metody są statyczne, inne nie. Z reguły metody statyczne wykorzystywane są do inicjalizowania obliczeń oraz tworzenia kontekstu dla określonego wykorzystania. Dla tak utworzonego kontekstu (czyli obiektu) wywoływane są następnie metody liczące, których głównym zadaniem jest modyfikacja zmiennych lokalnych – pól obiektu.

Po drugie, wszystkie metody to funkcje, których rezultatem jest tylko informacja o powodzeniu lub niepowodzeniu akcji. Wszystkie dane wejściowe i wyjściowe przekazywane są w postaci parametrów.

Na koniec należy powiązać identyfikator funkcji z typem realizującym jej zadanie:

```
create or replace function LimAvg (input number) return number
parallel_enable aggregate using TLimAvg;
```

/

Pozostaje już tylko policzyć średnią zarobków pracowników:

```
SELECT LimAvg(salary)
FROM employees;
```

4. Funkcje tabelowe

Funkcja tabelowa jako rezultat działania zwraca sobą kolekcję, której elementy w zależności od interpretacji można traktować jak „rekordy” lub „obiekty” i wykorzystywać w zapytaniach. Dzięki temu możemy w prosty sposób realizować wybór tych danych, dla których opis kryteriów w sposób proceduralny jest prosty i szybki, podczas gdy ten sam problem realizowany w „czystym” SQL-u prowadzi do stosowania karkołomnych konstrukcji.

Dla przykładu rozważmy przypadek tabeli sprzedaży, w której poszukujemy klientów dokonujących zakupu w kolejnych kilku dniach. Dla trzech dni zapytanie wygląda tak:

```
SELECT distinct s1.cust_id, s1.time_id
FROM sales s1, sales s2, sales s3
WHERE s1.cust_id = s2.cust_id
AND s1.cust_id = s3.cust_id
AND s1.time_id = s2.time_id - 1
AND s1.time_id = s3.time_id - 2;
```

Łatwo zauważyć, że jeżeli analiza ma obejmować więcej dni, to zapytanie stanie się bardziej rozbudowane i wystąpi w nim więcej złączeń. Bardziej eleganckim rozwiązaniem byłoby napisanie funkcji, w której liczba dni będzie parametrem, a w kodzie nie będzie wielu złączeń.

Dla funkcji tabelowych dostępne są dwa interfejsy: PL/SQL i obiektowy. Pierwszy z nich jest prosty, intuicyjny i nie wymaga znajomości technik obiektowych. Pozwala na implementacje większości prostszych przypadków. Drugi – obiektowy – opiera się na zasadzie obsługi zdarzeń (podobnie jak w opisanych wcześniej funkcjach agregujących).

4.1. Interfejs PL/SQL

W pierwszej kolejności należy zastanowić się jaka będzie struktura przyszłego zwracanego rekordu. Deklarujemy ja w postaci typu obiektowego *TDASales*:

```
create or replace type TDASales is object(
  cust_id number,
  time_id date
);
/
```

Następnie należy utworzyć kolekcję wcześniej zadeklarowanych rekordów – kolekcja ta będzie zwracana przez funkcję tabelową.

```
create or replace type TTDASales is table of TDASales;
/
```

Na koniec pozostaje utworzyć rzeczoną funkcję. Na uwagę zasługuje umieszczenie w nagłówku słowa kluczowego *pipelined*, iterowane wywoływanie instrukcji *pipe row* oraz pusty *return* na końcu funkcji.

```
create or replace function DASales(NumDays in number)
return TTDASales pipelined is
  last_cust number;
  last_time date;
  cnt number;
begin
```



```

static function ODCITablePrepare(sctx out TDASales2,
                                ti ODCITabFuncInfo, NumDays in number) return number,
static function ODCITableStart(sctx in out TDASales2,
                                NumDays in number) return number,
member function ODCITableFetch(self in out TDASales2,
                                nrows in number, rws out anydataset) return number,
member function ODCITableClose(self in TDASales2) return number,
);
/

```

Następnie ciało typu obiektowego:

create or replace type body **TDASales2** is

```

static function gettype(NumDays in number) return anytype is
    atyp anytype;
begin
    anytype.begincreate(dbms_types.typecode_object, atyp);
    atyp.addattr('CUST_ID', dbms_types.typecode_number, 0, -127, 22, 0, 0);
    for rc in 1..NumDays loop
        atyp.addattr('TIME'||rc, dbms_types.typecode_date, 0, 0, 0, 0, 0);
    end loop;
    atyp.endcreate;
    return atyp;
end gettype;

```

Stacyczna funkcja *gettype* definiuje typ rekordów wchodzących w skład zwracanej kolekcji. Można powiedzieć, że jest to dynamiczny odpowiednik deklaracji kolekcji *TDASales* z poprzedniego rozdziału. Struktura tego rekordu określa tym samym strukturę „tabeli” po zastosowaniu operatora *TABLE*. Typ ten definiowany jest poprzez iteracyjne dodawane atrybuty do standardowego typu *anytype*.

```

static function ODCITableDescribe(rtype out anytype,
                                    NumDays in number) return number is
begin
    anytype.begincreate(dbms_types.typecode_table, rtype);
    rtype.SetInfo(null, null, null, null, null, gettype(NumDays),
                 dbms_types.typecode_object, 0);
    rtype.endcreate();
    return ODCIConst.Success;
end ODCITableDescribe;

```

Metoda *ODCITableDescribe* wywoływana jest w momencie parsowania zapytania, w którym użyto funkcji tabelowej opartej o ten typ. Zwracany typ danych to kolekcja rekordów, każdy z tych rekordów opisany jest typem generowanym dynamicznie z pomocniczej metody *gettype* opisanej powyżej.

```

static function ODCITablePrepare(sctx out TDASales2,
                                ti ODCITabFuncInfo, NumDays in number) return number is
    i integer;
    tc      pls_integer;
    aname   varchar2(30);
    prec    pls_integer;
    scale   pls_integer;
    len     pls_integer;
    csid    pls_integer;
    csfrm   pls_integer;
    cnt     pls_integer;
    elem_typ anytype;

```

```

begin
  i := dbms_sql.open_cursor;
  tc := ti.rettype.GetAttrElemInfo(1, prec, scale, len, csid,
                                   csfrm, elem_typ, aname);
  sctx := TDASales2(NumDays, i, null, null, 0, elem_typ);
  return ODCIConst.Success;
end ODCITablePrepare;

```

Metoda *ODCITablePrepare* odpowiada za przygotowanie kontekstu, czyli zainicjalizowanie zmiennej obiektowej. Warto zwrócić uwagę, że jednym z parametrów konstruktora (czyli wartością inicjalną pola nowego obiektu) jest typ rekordowy, który przekazywany jest jako parametr wywołania. Typ ten wcześniej pobrany został przez serwer z metody *ODCITableDescribe*.

```

static function ODCITableStart(sctx in out TDASales2,
                                NumDays in number) return number is

  i          integer;
  cust       number;
  time       date;
begin
  dbms_sql.parse(sctx.cur, 'SELECT DISTINCT cust_id, time_id
                           FROM sales
                           ORDER BY cust_id, time_id', dbms_sql.native);
  dbms_sql.define_column(sctx.cur, 1, cust);
  dbms_sql.define_column(sctx.cur, 2, time);
  i := dbms_sql.execute(sctx.cur);
  return ODCIConst.Success;
end ODCITableStart;

```

Metoda *ODCITableStart* korzysta z wcześniej (w *ODCITablePrepare*) utworzonego kontekstu. Ponieważ rezultatem działania tej funkcji jest zbiór przetworzonych danych z tabeli *sales*, więc tworzony i wykonywany jest kursor dla takiego zapytania. Uchwyt do tego kursora znajdować się będzie w kontekście obiektu. Uwaga: w definicji metod typu obiektowego nie można stosować wygodniejszej instrukcji *EXECUTE IMMEDIATE*.

```

member function ODCITableFetch(self in out TDASales2, nrows in number,
                                rws out anydataset) return number is

  i          number;
  atyp       anytype;
  ads        anydataset := null;
  rec_cust   number := null;
  rec_time   date := null;
begin
  loop
    i := dbms_sql.fetch_rows(cur);
    if i > 0 then
      anydataset.begincreate(dbms_types.typecode_object, ret_type, ads);
      dbms_sql.column_value(cur, 1, rec_cust);
      dbms_sql.column_value(cur, 2, rec_time);
      if rec_cust=last_cust and rec_time = last_time + cnt then
        cnt := cnt + 1;
        if cnt >= nd then
          ads.addinstance;
          ads.piecewise();
          ads.setnumber(last_cust);
          for k in reverse 1..nd loop
            ads.setdate(last_time + cnt - k);
          end loop;
          ads.endcreate;
        end if;
      end if;
    end loop;
  end loop;
end ODCITableFetch;

```

```

        exit;
    end if;
else
    last_cust := rec_cust;
    last_time := rec_time;
    cnt := 1;
end if;
else
    ads := null;
    exit;
end if;
end loop;
rws := ads;
return ODCIConst.Success;
end ODCITableFetch;

```

Metoda *ODCITableFetch* wywoływana jest wielokrotnie, w zależności od liczby zwracanych rekordów. Serwer przed jej wywołaniem konstruuje odpowiedni bufor na wynik, a z zmiennej *nrows* podaje najbardziej oczekiwaną liczbę rekordów – tyle ile zmieści mu się w tym buforze. W wyjściowym parametrze *rws* metoda zwraca kolekcję rekordów. Ich liczba może być różna od wymaganej w *nrows* – jeśli będzie za duża, serwer przechowa nadmiar w dodatkowym buforze (stracimy jednak czas na jego alokację); jeśli za mało, serwer pobierze kolejne rekordy w następnym wywołaniu. Wyczerpanie wyniku polega na zwróceniu w parametrze *rws* wartości pustej *null*.

```

member function ODCITableClose(self in TDASales2) return number is
    i integer;
begin
    i := self.cur;
    dbms_sql.close_cursor(i);
    return ODCIConst.Success;
end ODCITableClose;

end TDASales2;
/

```

Na koniec wywoływana jest metoda, której zadaniem jest zwolnienie wszystkich zaalokowanych na potrzeby tego wywołania zasobów. W naszym przypadku jest to zamknięcie kursora.

Jak widać, główna różnica w stosunku do interfejsu PL/SQL polega na zdefiniowaniu typu rezultatu oraz samego rezultatu z wykorzystaniem kombinacji wbudowanych typów *anytype* oraz *anydataset*. Liczba kolumn wynikowego zbioru rekordów jest zmienna i zależy od parametru podawanego przy wywołaniu funkcji. Efekt ten jest nie do osiągnięcia przy wykorzystaniu „standardowych” technik SQL czy nawet PL/SQL.

Pozostaje jeszcze, podobnie jak w przypadku funkcji agregującej, utworzenie funkcji PL/SQL realizowanej przez typ opisany wyżej:

```

create or replace function DASales2(NumDays in number)
    return anydataset pipelined using TDASales2;
/

```

Rezultat działania tej funkcji można zobaczyć poniżej:

```

SQL> select * from table(DASales2(4));
CUST_ID TIME1      TIME2      TIME3      TIME4
-----
17 01/08/02 01/08/03 01/08/04 01/08/05
17 01/09/01 01/09/02 01/09/03 01/09/04
17 01/10/02 01/10/03 01/10/04 01/10/05

```

```

100 01/12/07 01/12/08 01/12/09 01/12/10
118 98/02/17 98/02/18 98/02/19 98/02/20
118 98/02/18 98/02/19 98/02/20 98/02/21
118 99/02/12 99/02/13 99/02/14 99/02/15
118 99/10/15 99/10/16 99/10/17 99/10/18
118 00/02/12 00/02/13 00/02/14 00/02/15
....

```

5. Indeksy

Interfejs implementujący funkcjonalność indeksu jest bodaj najbardziej skomplikowany w porównaniu z poprzednimi. Wiąże się to z liczbą zdarzeń, które wymagają obsługi oraz liczbą różnych rodzajów operacji, które wpływają na wewnętrzną zawartość danych indeksu. Dla przykładu rozważmy problem operatora *like*. Jak wiadomo zapytanie:

```

SELECT *
  FROM employees
 WHERE last_name LIKE '%ng%'; -- lub WHERE instr(last_name, 'ng') > 0

```

nie korzysta z indeksów. Spróbujemy utworzyć indeks do wyszukiwania napisów zawierających określony podciąg.

W pierwszej kolejności musimy określić jak będzie wyglądać operacja „wyszukaj napisu zawierającego podciąg”. Taką operację można bowiem zapisać na wiele sposobów a nie każdy z tych sposobów mógłby być zidentyfikowany przez optymalizator Oracle. W tym celu tworzymy operator i jego implementację w postaci funkcji:

```

create or replace function like_fun(wyraz in varchar2,
                                     podwyraz in varchar2) return number is
-- 0 - nie występuje, 1 - znaleziono wystąpienie
begin
  if InStr(upper(wyraz), upper(podwyraz)) > 0 then
    return 1;
  else
    return 0;
  end if;
end;
/

create or replace operator likef
  binding (varchar2, varchar2) return number
  using like_fun;

```

Następnie należy utworzyć typ obiektowy, który obsłuży zdarzenia związane z obsługą indeksu. Zdarzenia te można podzielić na trzy grupy:

- zdarzenia związane z DDL (tworzenie, kasowanie indeksu, tworzenie kasowanie partycji, import, eksport),
- zdarzenia związane z DML (modyfikacja danych wywołująca aktualizację danych indeksu),
- zdarzenia związane z wyszukiwaniem (wykorzystywanie danych indeksu do szybkiego wyszukiwania rekordu/-ów).

Ze względu na obszerną listę zdarzeń poniżej wymieniono tylko niektóre, wymagane do prezentacji działającego przykładu.


```

stmt varchar2(1000);
cnum integer;
s   varchar2(4000);
r   varchar2(18);
i   number;
begin
  stmt := 'CREATE TABLE ' || ia.IndexSchema || '.' || ia.IndexName
         || '_idxlt' || ' (skr varchar2(1000), skrowid ro-
wid)';
  execute immediate stmt;
  stmt := 'SELECT ' || ia.IndexCols(1).ColName || ', rowidtochar(ROWID) '
         || ' FROM ' || ia.IndexCols(1).TableSchema || '.'
         || ia.IndexCols(1).TableName;
  cnum := dbms_sql.open_cursor;
  dbms_sql.parse(cnum, stmt, dbms_sql.native);
  dbms_sql.define_column(cnum, 1, s, 4000);
  dbms_sql.define_column(cnum, 2, r, 18);
  i := dbms_sql.execute(cnum);
  loop
    i := dbms_sql.fetch_rows(cnum);
    exit when i=0;
    dbms_sql.column_value(cnum, 1, s);
    dbms_sql.column_value(cnum, 2, r);
    i := ODCIIndexInsert(ia, r, s, env);
  end loop;
  dbms_sql.close_cursor(cnum);
  stmt := 'DELETE ' || ia.IndexSchema || '.' || ia.IndexName
         || '_idxlt t '
         || ' WHERE (skr,skrowid) in (SELECT skr, skrowid '
         || ' FROM ' || ia.IndexSchema || '.'
         || ia.IndexName || '_idxlt '
         || ' GROUP BY skr, skrowid '
         || 'HAVING count(*)>1) '
         || ' AND rowid > (SELECT min(rowid) '
         || ' FROM ' || ia.IndexSchema || '.' || ia.IndexName
         || '_idxlt '
         || ' WHERE skr = t.skr '
         || ' AND skrowid = t.skrowid)';
  execute immediate stmt;
  stmt := 'CREATE UNIQUE INDEX ' || ia.IndexSchema || '.' || ia.IndexName
         || '_idxli'
         || ' ON ' || ia.IndexSchema || '.' || ia.IndexName
         || '_idxlt(skr, skrowid) global ';
  execute immediate stmt;
  return ODCIConst.Success;
end ODCIIndexCreate;

```

Metoda *ODCIIndexCreate* wywoływana jest podczas tworzenia indeksu. W metodzie tej w schemacie użytkownika tworzącego indeks tworzona jest tabela do przechowywania danych wewnętrznych, które indeksowane są standardowym indeksem b-drzewo. Dodatkowo, jeżeli indeksowana kolumna zawiera już dane, indeks aktualizowany jest bez pośrednictwa metody *ODCIIndexInsert*.

```

static function ODCIIndexDrop(ia ODCIIndexInfo, env ODCIEnv) return number is
  stmt varchar2(1000);
begin

```

```

    stmt := 'DROP TABLE ' || ia.IndexSchema || '.' || ia.IndexName || '_idx1t';
    execute immediate stmt;
    return ODCIConst.Success;
end ODCIIndexDrop;

```

Metoda *ODCIIndexDrop* wywoływana jest podczas jawnego lub niejawnego kasowania indeksu. W naszym przypadku służy ona do wykasowania wewnętrznej tabeli.

```

static function ODCIIndexInsert(ia ODCIIndexInfo, rid varchar2,
                                newval varchar2, env ODCIEnv) return number is

```

```

    stmt varchar2(1000);
    cnum integer;
    ii number;
    str varchar2(4000);
    minz constant number := 2;
    maxz constant number := 5;
begin
    if newval is not null then
        str := upper(newval);
        cnum := dbms_sql.open_cursor;
        stmt := 'INSERT INTO ' || ia.IndexSchema || '.' || ia.IndexName
                || '_idx1t(skr, skrowid) '
                || 'VALUES (:b1, :b2)';
        dbms_sql.parse(cnum, stmt, dbms_sql.native);
        for i in 1..length(str) loop
            for j in minz..case when length(str)-i+minz > maxz
                               then maxz
                               else length(str)-i+minz end loop
                dbms_sql.bind_variable(cnum, ':b1', substr(str, i, j));
                dbms_sql.bind_variable(cnum, ':b2', rid);
                begin
                    ii := dbms_sql.execute(cnum);
                exception
                    when others then null;
                end;
            end loop;
        end loop;
        dbms_sql.close_cursor(cnum);
    end if;
    return ODCIConst.Success;
end ODCIIndexInsert;

```

```

static function ODCIIndexUpdate(ia ODCIIndexInfo, rid varchar2,
                                oldval varchar2, newval varchar2, env ODCIEnv) return number is
    ii number;

```

```

begin
    ii := ODCIIndexDelete(ia, rid, oldval, env);
    ii := ODCIIndexInsert(ia, rid, newval, env);
    return ODCIConst.Success;
end ODCIIndexUpdate;

```

```

static function ODCIIndexDelete(ia ODCIIndexInfo, rid VARCHAR2,
                                oldval varchar2, env ODCIEnv) return number is

```

```

    stmt VARCHAR2(1000);
    cnum integer;
    ii number;

```

```

begin
  stmt := 'DELETE ' || ia.IndexSchema || '.' || ia.IndexName || '_idx1t '
        || 'WHERE skrowid = ''||rid||''';
  cnum := dbms_sql.open_cursor;
  dbms_sql.parse(cnum, stmt, dbms_sql.native);
  ii := dbms_sql.execute(cnum);
  dbms_sql.close_cursor(cnum);
  return ODCIConst.Success;
end ODCIIndexDelete;

```

Metody *ODCIIndexInsert*, *ODCIIndexUpdate*, *ODCIIndexDelete* odpowiadają za aktualizację wewnętrznych danych indeksu. We wszystkich tych metodach przekazywana jest wartość z indeksowanej kolumny oraz adres *ROWID* modyfikowanego rekordu.

```

static function ODCIIndexStart(sctx in out TIdx1, ia ODCIIndexInfo,
                               pi ODCIPredInfo, qi ODCIQueryInfo,
                               strt number, stop number, cmpval varchar2,
                               env ODCIEnv) return number is

  stmt  varchar2(1000);
  cnum  integer;
  ii    number;
  s     varchar2(18);
begin
  stmt := 'SELECT skrowid '
        || ' FROM ' || ia.IndexSchema || '.' || ia.IndexName || '_idx1t '
        || ' WHERE skr = upper(''||cmpval||'')';
  cnum := dbms_sql.open_cursor;
  dbms_sql.parse(cnum, stmt, dbms_sql.native);
  dbms_sql.define_column(cnum, 1, s, 18);
  ii := dbms_sql.execute(cnum);
  sctx := TIdx1(cnum);
  return ODCIConst.Success;
end ODCIIndexStart;

```

Wyszukiwanie (np. w wyniku zapytania SELECT) rozpoczyna się wywołaniem zdarzenia *ODCIIndexStart*. Parametry przekazywane do tej metody pozwalają na rozpoznanie charakteru wyszukiwania (pełne, zakresowe, punktowe), wyboru operatora i uczestniczących kolumn (w indeksach złożonych). W powyższym przykładzie wykorzystywany jest tylko jeden operator, który zwraca tylko dwie wartości, stąd analiza tych parametrów mogła być pominięta.

```

member function ODCIIndexFetch(self in TIdx1, nrows in number,
                                rids out ODCIRidList, env ODCIEnv) return number is

  s     varchar2(18);
  i     number;
  j     number;
  rlist odciridlist := odciridlist();
begin
  rlist.extend(nrows);
  j := 0;
  loop
    i := dbms_sql.fetch_rows(cnum);
    exit when i = 0;
    j := j + 1;
    dbms_sql.column_value(self.cnum, 1, rlist(j));
    exit when j = nrows;
  end loop;
  if j > 0 then
    rids := rlist;

```

```

else
  rids := null;
end if;
return ODCIConst.Success;
end ODCIIndexFetch;

```

Wywołując wielokrotnie metodę `ODCIIndexFetch` serwer pobiera adresy rekordów wyszukiwanych przez indeks. Zwrot pustej wartości NULL oznacza, że więcej rekordów nie znaleziono.

```

member function ODCIIndexClose(self in TIdx1, env ODCIEnv) return number is
  i integer;
begin
  i := self.cnum;
  dbms_sql.close_cursor(i);
  return ODCIConst.Success;
end ODCIIndexClose;

end TIdx1;
/

```

Na zakończenie wyszukiwania wywoływana jest metoda `ODCIIndexClose`, w której zamykany jest kursor.

Aby móc utworzyć indeks należy jeszcze stworzyć strukturę pomocniczą `INDEXTYPE`. Struktura ta pozwala na odseparowanie pakietu i samego indeksu, co ma znaczenie przy nadawaniu uprawnień.

```

CREATE OR REPLACE INDEXTYPE TTIdx1
  FOR likef(vvarchar2, vvarchar2)
  USING TIdx1;

CREATE INDEX Id1 ON test(t)
  INDEXTYPE IS TTIdx1;

```

6. Uwagi końcowe

Przedstawione w niniejszym artykule przykłady prezentują podstawowe możliwości Oracle API. Pozostałe, ze względu na obszerność kodu źródłowego nie mogły być tutaj opisane. Wiele z nich dotyczy dodatkowej funkcjonalności możliwej do zastosowania w specyficznych przypadkach oraz do optymalizacji, przyspieszającej działanie indeksów.

Własne **funkcje agregujące** mogą być stosowane w zaawansowanej analizie OLAP, gdzie stosuje się lokalne partycjonowanie danych, tworzenie okien, itp.

Funkcje tabelowe mogą tworzyć własne dane (np. generator tabeli o określonej liczbie rekordów – odpowiednik tabeli *dual*) lub pobierać i przetwarzać dane znajdujące się w LOB-ach lub plikach zewnętrznych.

Własne indeksy mogą posiadać dodatkowy moduł statystyczny pozwalający na lepsze jego wykorzystanie przez optymalizator kosztowy Oracle.

Wszystkie typy obiektowe służące do obsługi generowanych przez serwer zdarzeń mogą mieć implementacje swoich metod w języku Java lub C/C++.

Bibliografia

- [BKM99] Banerjee S., Krishnamurthy V., Murthy R.: All Your Data: The Oracle Extensibility Architecture

- [DoKo] Doeller M., Kosch H.: An MPEG-7 Multimedia Data Cartridge, Institute of Information Technology University Klagenfurt
- [DoKo02] Doeller M., Kosch H.: Enhancement of Oracle's Indexing Capabilities through GiST-implemented Access Methods, Institute of Information Technology University Klagenfurt, No TR/ITEC/02/2.09, April 2002
- [HoMa] Homer W., Mandl M.: Unified Storage And Searching Of Structures And Relational Data In Oracle®: An Overview Of AUSPYX™ 1.5, Discovery Informatics Solutions 2004
- [KILi] Kleiner C., Lipeck u. W.: OraGiST - How to Make User-Defined Indexing Become Usable and Useful, Universitat Hannover
- [OraDok] Oracle® Database Data Cartridge Developer's Guide, 10g Release 2 (10.2) Part Number B14289-01
- [OraMet] <http://metalink.oracle.com>