IX Konferencja PLOUG
Kościelisko
Październik 2003

Strojenie zapytań SQL – czyli, „można jeszcze szybciej”

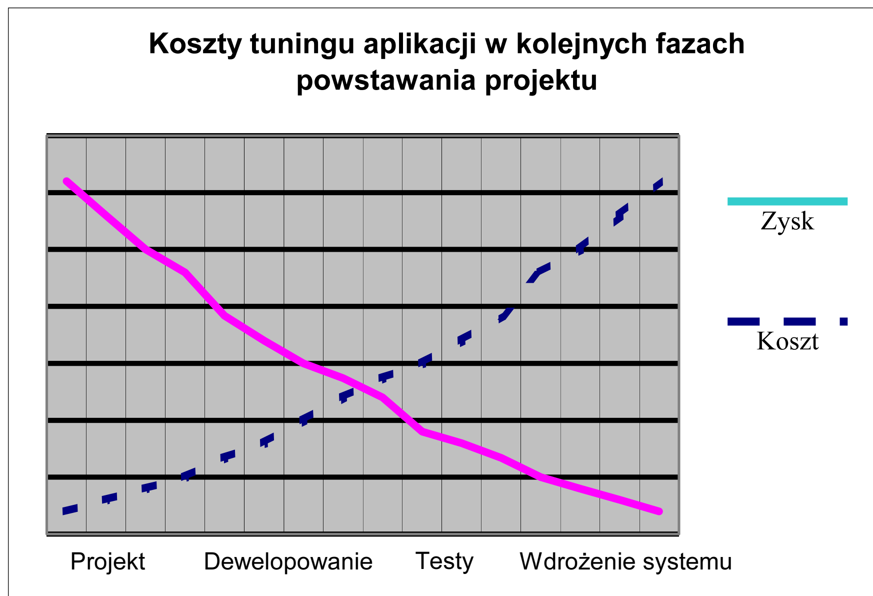
Krzysztof Kawa

Strojeniu zapytań SQL poświęcono już bardzo wiele miejsca w literaturze. Jako osoba z dość dużym doświadczeniem w dziedzinie strojenia zapytań SQL baz danych Oracle 8i i 9i chciałbym przedstawić kilka rad, których stosowanie z pewnością korzystnie wpłynie na szybkość realizacji zapytań.

Informacja o autorze:
Krzysztof Kawa, absolwent informatyki na Politechnice Poznańskiej w Poznaniu oraz Wydziału Biznesu Międzynarodowego na Akademii Ekonomicznej w Poznaniu. Od kilku lat pracuje przy projektowaniu i eksploatacji dużych systemów informatycznych w kraju i zagranicą. Specjalizuje się w strojeniu aplikacji opartych na rozwiązaniach firmy Oracle oraz projektowaniu rozwiązań J2EE.

1. Wstęp

Znana z inżynierii oprogramowania zasada mówi, iż tuning systemu powinien zostać prowadzony już od wczesnych faz budowania/projektowania systemu. Tylko takie podejście pozwala na uzyskanie znaczących efektów stosunkowo małym kosztem. Koszt tuningu aplikacji rośnie wraz z kolejnymi fazami powstawania projektu, a co więcej, założenia [w przypadku systemów bazodanowych chodzi tu choćby o schemat tabel] powstałe w pierwszych fazach będą wpływać negatywnie [bądź korzystnie] na szybkość pracy całego systemu w przyszłości. Obrazowo przedstawia to rysunek 1.



Rys. 1. Koszty tuningu aplikacji w kolejnych fazach realizacji projektu

2. Aspekty strojenia aplikacji

Strojenie SQL jest tylko jednym z wielu aspektów wpływających na szybkość pracy całego systemu. Innymi, nie mniej ważnymi aspektami wpływającymi na wydajność całego systemu są:

- architektura całej aplikacji uwzględniająca aspekty związane nie tylko z bazą danych, ale i aspekty doboru technologii, języka programowania użytego przy tworzeniu aplikacji
- fizyczny koncept bazy danych wyznaczający reguły przechowywania danych w odpowiednich strukturach, zastosowanie indeksów, replikacji, tabel agregacyjnych
- aspekty związane z doborem platformy systemowo-sprzętowej oraz jej tuningowanie [zarówno samego systemu operacyjnego jak i sprzętu]

3. Metody dostępu do danych, metody łączenia tabel, optymalizatory, histogramy

Zanim przedstawię praktyczne aspekty tuning SQL i PLSQL zamierzam przypomnieć i w skrócie omówić podstawowe aspekty związane z wykonywaniem zapytań.

Metody dostępu do tabel

- **full scan** – to chyba najprostsza metoda dostępu do danych, polegająca na tym, iż Oracle czyta wiersz po wierszu zapisanym w tabeli, aż do osiągnięcia tzw. high-water mark
- **rowid access** – rowid jest pseudokolumną, zawierającą w treści fizyczną reprezentację danych. Odczyt przy jej pomocy jest szybki
- **index lookup** – Oracle przy dostępie do danych korzysta z indeksu pozwalającego na szybsze dotarcie do danych [Oracle wspiera 2 typy indeksów – bitmapowe oraz B*]
- **hash key access –** metoda dostępu oparta na funkcji haszującej tj. przekształceniu matematycznemu wyznaczającemu lokalizację przechowywania danych na podstawie wartości w kolumnie [hash cluster]

Metody łączenia tabel

- **sort merge** – do wykonania połączenia nie jest wymagane użycie indeksów; Oracle sortuje dwie łączone tabele względem kolumn, które maja być połączone, a następnie łączy posortowane tabele
- **nested loop –** algorytm ten zwykle korzysta z indeksów przynajmniej dla jednej z tabel. Na mniejszej z tabel wykonywany jest full-scan (często z pominięciem indeksu) i dla każdej krotki z tego zbioru używany jest indeks, aby odnaleźć krotkę do połączenia w drugiej tabeli
- **hash join –** dla większej z obu tabel tworzona jest tabela haszowa, następnie odczytywana jest zawartość mniejszej tabeli, a tabela haszowa używana jest do odnalezienia wiersza w większej tabeli [metoda sprawdza się dobrze, gdy tabela haszowa może w całości zostać utworzona w pamięci]

Optymalizatory a statystyki

Po sprawdzeniu składni zadanego polecenia w języku SQL (język SQL jest językiem deklaratywnym, tym samym polecenie mające na przykład wyświetlić określone krotki z bazy danych nie zawiera algorytmu, według którego baza danych ma postępować, aby dotrzeć do danych) Oracle musi w przeciągu ułamka sekundy wybrać najbardziej optymalną drogę dostępu do danych – do tego celu zostały zaimplementowane 2 rodzaje optymalizatorów:

- **regułowy** (ang. rule)– optymalizator podejmuje decyzję, w jaki sposób najszybciej dotrzeć do danych na podstawie zestawu reguł oraz rankingu różnych ścieżek dostępu do danych. Optymalizator regułowy dla przykładu zawsze faworyzuje dostęp poprzez indeks, nie potrafi rozróżnić i dostosować się względem dostępu do małych tabel jak i do wielkich. Decyzja o dostępie zapada na podstawie indeksów. Optymalizator nie jest rozwijany od wersji Oracle 6
- **kosztowy** (ang. cost-based) – został wprowadzony w wersji Oracle 7. Algorytm korzysta z zgromadzonych wcześniej informacji informacjach o samych danych. Jest on obecnie domyślnym optymalizatorem. Jest on również zalecany przez Oracle, gdyż tylko nad nim trwają dalsze prace badawcze.

Jak to zostało już wyżej zaznaczone optymalizator kosztowy przy wyborze ścieżki dostępu do danych korzysta ze zgromadzonych wcześniej statystyk, stąd pojawia się troska o to, aby zgromadzone statystyki były aktualne.

Do zbierania statystyk służ polecenie ANALYZE o składni:

```
ANALYZE {TABLE | INDEX | CLUSTER} name
   [{COMPUTE STATISTICS |
   ESTIMATE STATISTICS SAMPLE size [ROWS|PERCENT] }
```

```
[FOR {TABLE |
ALL [INDEXED]  COLUMNS [SIZE histogram_size] |
Column_list [SIZE histogram_size] |
ALL [LOCAL] INDEXES ] } …]
```

Polecenie zbiera informacje statystyczne dla tabeli, indeksu lub klastra i zapisuje je w słowniku danych w celu późniejszego użycia ich przy wyznaczaniu ścieżki dostępu. W słowniku zapisywane są:

- dla tabeli (liczba wierszy, liczba użytych i pustych bloków, średnia długość wiersza, średnia zajętość bloków danych)
- dla indeksu (liczba bloków liści, głębokość indeksu B*, ilość jednakowych kluczy)

Chcąc przeliczyć statystyki dla wszystkich obiektów jednego schematu (tabel lub indeksów) znacznie łatwiej posłużyć się procedurą ANALYZE_SCHEMA z pakietu DBMS_UTILITY. Pozwala ono wybrać pomiędzy estymowaniem statystyk a obliczaniem statystyk bazując na całych obiektach.

Niestety statystyki nie wnoszą żadnych informacji o rozkładzie samych danych, co może w wielu sytuacjach być zgubne przy budowanie optymalnej ścieżki dostępu do danych. Weźmy za przykład tabele zawierającą informacje o wieku osób. Tylko bardzo niewiele osób będzie w wieku powyżej 100 lat. Informacja o rozkładzie statystycznym danych mogłaby być bardzo pomocna podczas realizacji zapytań mających wyszukać osoby starsze niż 100 lat. Jeśli tabela z danymi o osobach jest duża, wykonanie na niej full-scana będzie bardzo kosztowne, a przecież Oracle mógłby się posłużyć się w tym przypadku istniejącym indeksem i dotrzeć do danych błyskawicznie.

Histogramy tworzymy następującym poleceniem :

```
ANALYZE TABLE table_name [ESTIMATE…|CALCULATE…]
   {FOR COLUMNS column_list |
     FOR ALL COLUMNS |
     FOR ALL INDEXED COLUMNS }
      SIZE n
```

4. Dostęp do danych w tabeli – tuning

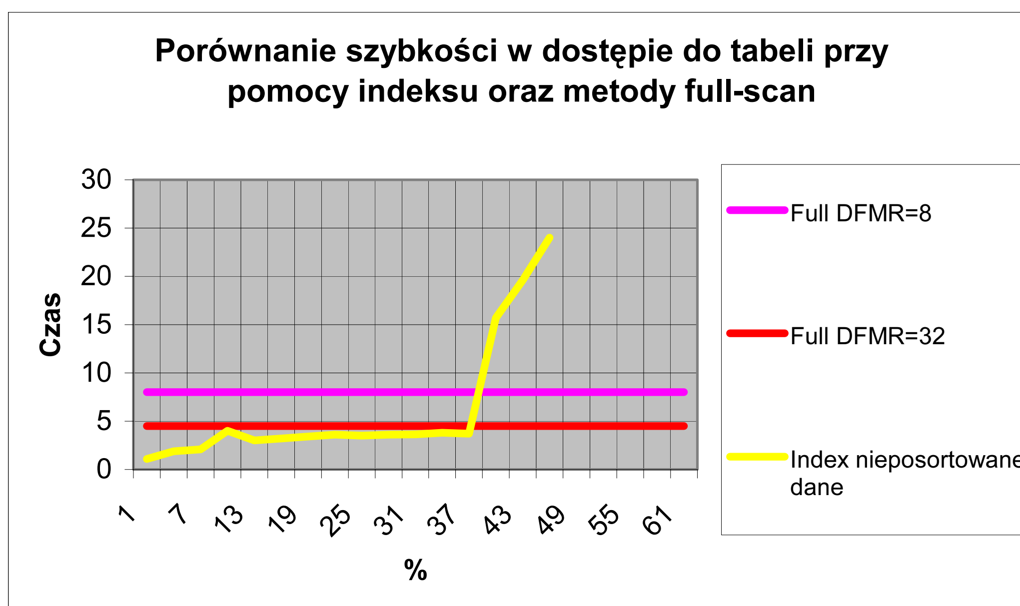
W rozdziale tym przedstawię kilka sposobów dostępu do danych w tabeli. Przedstawię ich porównanie oraz sposoby zwiększenia wydajności dostępu do danych.

Optymalizator nie zawsze ma dyspozycji wszystkie potrzebne informacje, aby wybrać najbardziej optymalny sposób dostępu do danych, co więcej ma na to wielokrotnie mniej czasu niż osoba pisząca zapytanie w języku SQL.

Powszechnie, choć niesłusznie przyjęła się zasada unikania metody full-scan w dostępie do danych. Często full-scan może okazać się metodą mniej kosztowną od dostępu poprzez indeks. Dzieje się tak, choćby i z tej przyczyny, iż dostęp przez indeks wymaga odczytu zarówno bloków indeksu oraz zwykle bloków danych. Co więcej może wiązać się to z naprzemiennym czytaniem bloków danych i indeksu, co jeszcze bardziej zwiększa ruch. W literaturze znajduje się wiele szkół mówiących, przy odczycie ilu procent danych zaczyna się opłacać stosowanie metody full-scan.

Doznania empiryczne przedstawia rysunek 2. Zostały na nim zobrazowane czasy wykonywania zapytań wykonanych przy użyciu indeksów jak i przy użyciu metody full-scan [dla wartości para-

metru DB_FILE_MULTIBBLOCK_READ_COUNT[1] 8 oraz 32] względem procentu odczytywanych danych



Rys. 2. Porównanie szybkości w dostępie do tabeli przy pomocy indeksu oraz metody full-scan

Jak można łatwo odczytać z wykresu już przy odczycie około 40% wszystkich wierszy tabeli zaczyna opłacać się korzystanie z metody full-scan.

Optymalizator kosztowy w porównaniu do optymalizatora regułowego, ma dostęp do informacji statystycznych o tabelach. Tym samym finalna decyzja, jaka metoda dostępu zostanie użyta przy odczycie danych z tabeli zostanie podjęta m. in. na podstawie:

- ilości bloków zaalokowanych przez tabele [poniżej high-water mark]
- rozmiaru bloku danych
- selektywności indeksów
- głębokości indeksu – ilości IO operacji potrzebnych do odczytania bloku danych

Kiedy Oracle użyje FULL SCAN ?

Czasem przygotowane zapytanie wymusza na optymalizatorze wybranie metody full-scan, mimo iż nie musi być ona optymalną metodą do uzyskania pożądanego rezultatu. Poniżej przedstawiam kilka przykładów.

Użycie operatora <>

W przypadku zastosowania operatora <> w zapytaniu powodujemy, iż Oracle zastosuje metodę dostępu full-scan. Wydaje się to nawet sensowne; szybciej jest, bowiem przejrzeć wszystkie krotki (full scan)pominąwszy te, które nie spełniają warunków zapytania

Rozpatrzmy plany wykonań 2 zapytań :

[1] DB_FILE_MULTIBBLOCK_READ_COUNT określa liczbę bloków, która może być odczytana podczas pojedynczej operacji IO

```
SELECT *
  FROM DATA
 WHERE id = 7;
Plan wykonywania
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=3 Card=1 Bytes=74)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'DATA' (Cost=3 Card=1 Bytes=74)
   2    1     INDEX (RANGE SCAN) OF 'DATA_PK' (UNIQUE) (Cost=2 Card=1)


SELECT *
  FROM DATA
 WHERE id <> 7
Plan wykonywania
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=5 Card=2647 Bytes=195878)
   1    0   TABLE ACCESS (FULL) OF 'DATA' (Cost=5 Card=2647 Bytes=195878)
```

Interpretując plan zapytania możemy stwierdzić, iż Oracle zastosował przeglądanie całego indeksu – full scan całego indeksu co dodatkowo wydłużyło tylko czas wykonywania zapytania. Sensownym staje się, więc unikanie operatora <> w sytuacjach, kiedy zapytanie można zredagować inaczej.

Zbiorcze dane przedstawia rysunek 3.

Rys. 3. Koszt wykonań zapytań z użyciem operatorów <> i =

Szukanie wartości null

Załóżmy, iż tabela Data zawiera kolumnę day, która może przyjmować również wartość null.

Dla zapytania postaci :

```
SELECT *
  FROM DATA
 WHERE DAY IS NULL
Plan wykonywania
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=5 Card=2647 Bytes=195878)
   1    0   TABLE ACCESS (FULL) OF 'DATA' (Cost=5 Card=2647 Bytes=195878)
```

Oracle również zastosuje metodę full scan. Obejściem tego jest nadanie kolumnie day zamiast wartości null innej wartości np. „BRAK”. Teraz równoważne zapytanie ma znacznie korzystniejszy plan wykonania.

```
SELECT /*+ index(p) */  *
  FROM DATA
 WHERE DAY = 'BRAK'
Plan wykonywania
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=3 Card=20 Bytes=1480)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'DATA' (Cost=3 Card=20 By-
tes=1480)
   2    1     INDEX (RANGE SCAN) OF 'DATA_I' (NON-UNIQUE) (Cost=2 Card=20)
```

Bardzo często aby wzbogacić proces decyzyjny o dodatkowe informacje zaleca się stworzenie histogramu na kolumnie której szukamy. Przy braku histogramu Oracle może nie wiedzieć, iż krotek z wartością kolumny DAY=BRAK jest np. 5 i lepiej użyć do ich wyznaczenia indeksu niż forsować uzyskanie wyniku metodą full scan.

Ukrywanie indeksu przez funkcję

Załóżmy taką sytuację, iż tabela data zawiera pole text na którym założony jest indeks. Plan wykonania poniższego zapytania jest bardzo dobry. Oracle skorzystał z założonego indeksu i odnalazł wiersze bardzo szybko z jego pomocą.

```
SELECT *
  FROM DATA
 WHERE text = 'Jand';
Plan wykonywania
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=118)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'DATA' (Cost=2 Card=1 Bytes=118)
   2    1     INDEX (RANGE SCAN) OF 'DATA' (NON-UNIQUE) (Cost=1 Card=1)
```

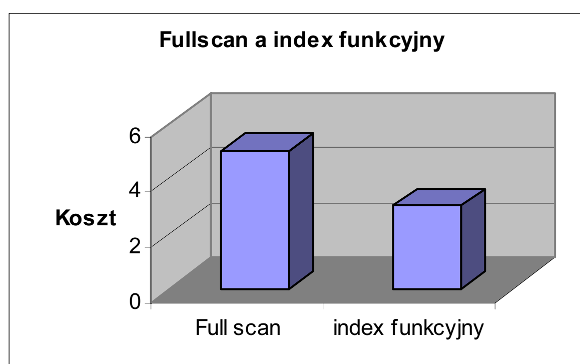
Jednak użyta w takim zapytaniu funkcja upper przysłoni - spowoduje, iż Oracle nie będzie mógł skorzystać z dobrodziejstw indeksu, stąd będzie musiał wykorzystać metodę fullscan – spójrzmy poniżej.

```
SELECT *
  FROM DATA
 WHERE UPPER (text) = 'Jand';
Plan wykonywania
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=5 Card=18 Bytes=2124)
   1    0   TABLE ACCESS (FULL) OF 'DATA' (Cost=5 Card=18 Bytes=2124)
```

Wyjściem z sytuacji jest utworzenie indeksu funkcyjnego :

```
CREATE INDEX idx1 ON DATA (UPPER(text));
```

Różnice w szybkości przy dostępie z wykorzystaniem metody full-scan oraz przy pomocy indeksu funkcyjnego pokazuje rysunek 4



Rys. 4. Full-scan a indeks funkcyjny

Znacznik High Water-Mark a eksport / import

Bardzo często dzieje się tak iż dostęp do tabeli przy metodzie dostępu full-scan z czasem staje się coraz wolniejszy. Co więcej przyczyny nie odnajdujemy w ilości danych, gdyż ilość krotek w tabeli wzrosła nieznacznie lub nawet nie zmieniła się (na tabeli były wykonywane operacje wstawiania, usuwania). Przyczyną takiej sytuacji może być fragmentacja tabeli. Oznacza to, iż przy operacji full-scan Oracle będzie czytał całą tabelę do najodleglejszego miejsca w tabeli w którym kiedykolwiek były dane (high-water mark), do którego kiedykolwiek były wstawiane dane (mimo iż obecnie obszar ten nie musi być wykorzystywany). Sytuację taką przedstwia rysunek 5. Wyjściem z sytuacji jest wyeksportowanie tabeli/schematu i ponowne jej/go zaimportowanie.

Rys. 5. Przeglądanie tabeli aż do High-Water Mark przy metodzie full-scan

5. Strategie indeksowania

Najczęściej stosowane w Oracle metody indeksowania to :

- B* – dokładniejsze informacje jak zbudowany jest B*-tree można znaleźć w [Ora99]. Dla nas ważną informacją powinno być, iż indeks ten nadaje się do indeksowania bardzo dużych ilości danych. Wspomaga zarówno zapytania korzystające z metody index-lookup jak również zapytania typu range queries. Teoretycznie dotarcie do każdej krotki nie powinno zabierać więcej niż 4 operacje IO
- Indeksy bitmapowe, którego budowę można znaleźć w pracy [Ora99] są bardzo przydatne podczas optymalizacji zapytań na kolumnach mających małą liczność – przyjmują wartości z małej dziedziny np. atrybut płeć [kobieta, mężczyzna], kolor [czarny, biały, zielony, niebieski]

Wykorzystywanie indeksów

Jeżeli w zapytaniu w klauzuli WHERE wyspecyfikujemy kilka kolumn naraz, indeks typu B* najbardziej efektywnie będzie się zachowywał, jeżeli jest on indeksem zbudowanych na wszystkich kolumnach podanych w klauzuli WHERE i jeżeli kolumny w warunku WHERE są podane w kolejności identycznej jak występują w indeksie. Najlepiej jest też, jeżeli kolumna o najwyższej selektywności występuje w indeksie (i tym samym w zapytaniu) na pierwszej pozycji

```
SELECT *
  FROM cars
 WHERE plates = 'PSZ R960'
   AND made_by = 'Renault'
   AND prize = 34000
   AND color = 'yellow'
```

Dla tak skonstruowanego zapytania najbardziej efektywnym indeksem byłby indeks zbudowany na kolumnach (ważna kolejność) :

Plates made_by prize color

Zaś zapytanie postaci:

```
SELECT *
  FROM cars
 WHERE made_by = 'Renault' AND prize = 34000;
```

Może nie wykorzystać indeksu. Chciałbym tutaj zwrócić uwagę, iż różne wersje Oracle mogą się różnie zachowywać. Nowsze automatycznie dostosowują kolejność atrybutów do istniejących indeksów, nie występuje w nich problem, iż indeks może zostać użyty niewłaściwie. Tym nie mniej, bardzo ważne jest sprawdzanie planów wykonań dla zapytań wykorzystujących nie wszystkie atrybuty istniejącego indeksu; może bowiem okazać się, iż zapytanie może z opisywanej tu przyczyny wykonywać się wielokrotnie wolniej.

Aby uniknąć tworzenia wielu indeksów dla wielu różnych kombinacji kolumn możemy stworzyć indeksy dla pojedynczych kolumn a następnie wymusić, aby Oracle połączył np. 4 indeksy w jeden.

```
SELECT /*+and_equal(c, plates_i, made_by_i, prize_i, color_i) */    *
  FROM cars c
 WHERE plates = 'PSZ R960'
   AND made_by = 'Renault'
   AND prize = 34000
   AND color = 'yellow'
```

Niestety, przeprowadzone testy wydajnościowe pokazały, iż najlepsze wyniki dają indeksy zbudowane na wszystkich kolumnach występujących w zapytaniu, najmniej korzystny okazał się sztuczny merge wszystkich indeksów.

6. PL/SQL – optymalizacja

Wiele aplikacji zbudowanych w oparciu o rozwiązania firmy Oracle korzysta również z dobrodziejstw programowania proceduralnego tj. z PL/SQLa. Chciałbym również przedstawić kilka metod, których stosowania z pewnością korzystnie wpłynie na szybkość kodu napisanego w PL/SQL.

Właściwe stosowanie VARCHAR2

Optymalizator wirtualnej maszyny PL/SQL zależnie od wielkości zadeklarowanej zmiennej typu VARCHAR2 może przeprowadzać pewne operacje mające wpływać na lepszą wydajniejszą jego pracę. I tak :

- VARCHAR2(2000 i mniej) – optymalizacja względem szybkości
- VARCHAR2(powyżej 2000) – optymalizacja względem wykorzystania

Stosowanie PLS_INTEGER zamiast NUMBER

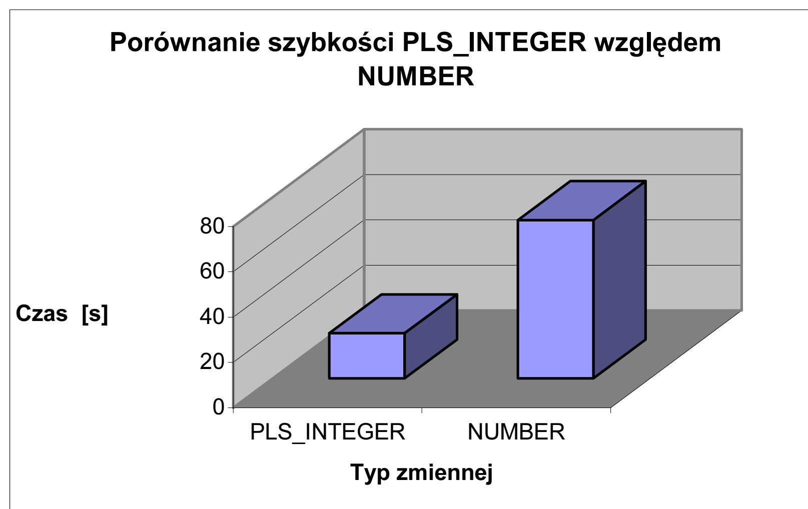
W przypadku wykonywania podstawowych operacji (dodawania, odejmowanie, mnożenie,) na liczbach całkowitych o wiele bardziej efektywne jest użycie zmiennych typu PLS_INETEGER niż 8-bajtowego typu NUMBER. Spójrzymy na czasy realizacji poniższego programu w zależności od tego, czy zmienna i była typu NUMBER czy typu PLS_INTEGER.

Przykład 1

```
DECLARE
 i   PLS_INTEGER;
BEGIN
   i := 10000;
   FOR k IN 1 .. 10000000 LOOP
      i :=   i + k; i :=   i - k;
      i :=   i + k; i :=   i - k;
   END LOOP;
END;
```

Przykład 2

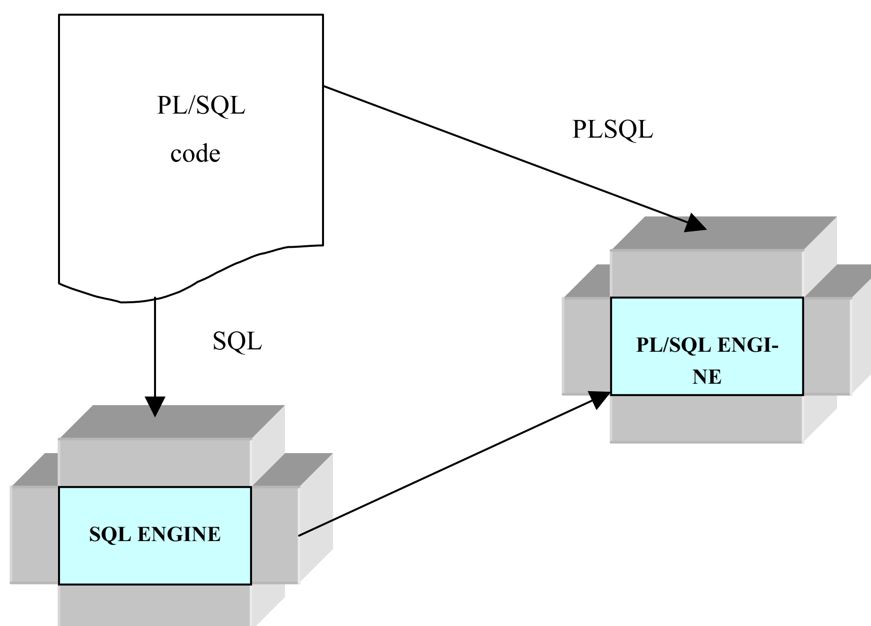
```
DECLARE
i NUMBER;
BEGIN
   i := 10000;
   FOR k IN 1 .. 10000000 LOOP
      i :=   i + k; i :=   i - k;
      i :=   i + k; i :=   i - k;
   END LOOP;
END;
```



Rys. 6. Porównanie szybkości PLS_INTEGER względem NUMBER

Używanie BULKs

Server Oracle ma 3 engine-y wykonawcze : PL/SQLa, SQLa oraz Java-y. W przypadku kiedy np. w pętli PL/SQLa wykonywane jest jakieś polecenie SQL, Oracle musi przełączać kontekst pomiędzy engine-m PL/SQL a enigine-m wykonującym polecenie SQL [engine PLSQLa przekazuje parametry do wykonywanego w engine-ie SQLa polecenia SQL]. Obrazowo przedstwia to rysunek 7.



Rys. 7. Przetwarzanie SQL w pętli bez zastosowania metody wsadowej (bulk)

Taki sposób wykonywania jest o wiele bardziej czasochłonny od tego, który zaraz pokaże na przykładzie instrukcji FORALL.

Znacznie lepiej zastosować jest bowiem przetwarzanie wsadowe polegające na tym, iż engine PL/SQL przysyła wyznaczone wcześniej identyfikatory (zbiorczo) do enigne SQL, a ten wykonuje swoje operacje bez kosztownego przełączania kontekstu. Spójrzymy na poniższe 2 przykłady :

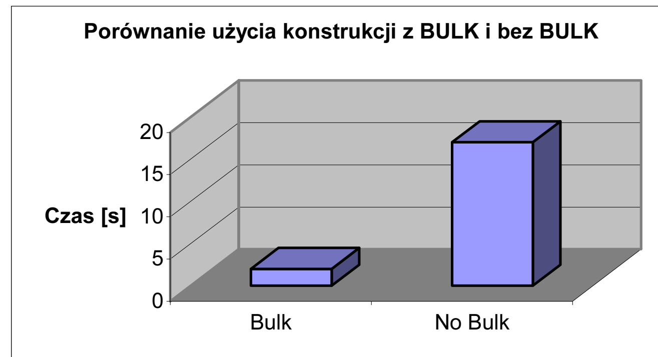
Przykład 1

```
DECLARE
   p   number_array;
BEGIN
   p := number_array ();
   FOR k IN 1 .. 30000  LOOP
      p.EXTEND; p (p.COUNT) := k;
   END LOOP;

   FOR i IN p.FIRST .. p.LAST LOOP
      INSERT INTO a VALUES (p (i), SYSDATE);
   END LOOP;
   COMMIT;
END;
```

Przykład 2

```
DECLARE
   p   number_array;
```

```
BEGIN
   p := number_array ();
   FOR k IN 1 .. 30000   LOOP
      p.EXTEND; p (p.COUNT) := k;
   END LOOP;

   FORALL j IN p.FIRST .. p.LAST
      INSERT INTO a VALUES (p (j), SYSDATE);
   COMMIT;
END;
```

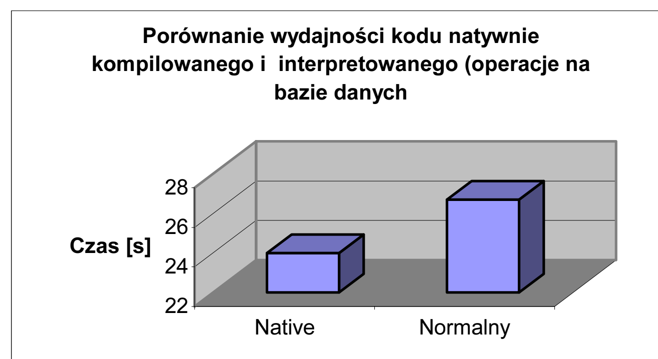


Rys. 8. Porównanie czasowe użycia normalnej pętli oraz pętli z zastosowaniem przetwarzania wsadowego (bulk)

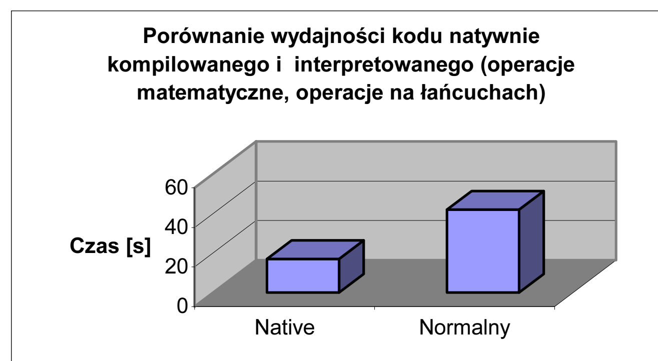
PL/SQL Native compilation

W poprzednich wersjach niż Oracle 9i, kod pakietów składowanych, napisanych w PL/SQL tłumaczony był i przechowywany wewnątrz bazy danych jako byte-code. Wraz z wersją 9i stało sie możliwe tłumaczenia kodu w PL/SQL do natywnej reprezentacji dla danej platformy. Realizowane jest to przez tłumaczenie kodu PL/SQL na język C, a następnie kompilowanie i linkowanie przetranslowanego kodu na platformie na której jest uruchamiany [parametry dot. kompilatora, linkera podawane są jako parametry inicjalizacyjne]. Co ważne, przekompilowane pakiety są przechowywane w systemie plików w postaci bibliotek dynamicznie ładowanych. Zazwyczaj (wykazały to testy) natywna kompilacja pakietu może być kilkukrotnie dłuższa.

Co do wydajności, to testy przeprowadzone przeze mnie wykazały, iż przyspieszenie było wyraźnie widoczne jedynie dla kodu w PL/SQL operującego mało na danych z bazy danych, czy też zawierającego wiele poleceń SQLa [select, insert, update] wiążących się z odczytem/zapisem do bazy danych. Zysk w takich sytuacjach był paru-procentowy (zależny od tego jak wiele było w nim wykonywanych operacji na danych w bazie). Dla kodu zawierającego duże ilości operacji matematycznych, na łańcuchach zysk był faktycznie spory (30 – 400 %).



Rys. 9. Porównanie wydajności kodu natywnie kompilowanego i interpretowanego (operacje na bazie danych)



Rys. 10. Porównanie wydajności kodu natywnie kompilowanego i interpretowanego (operacje matematyczne)

Bibliografia

[Ora99] Oracle 8i Concepts, 1999

[LoTh00] Loney K., Theriault M.: Oracle 8i Podręcznik administratora baz danych, Helion, 2002

[Feu03] Feuerstein S.: Oracle PL/SQL Programming, O'Reilly, 2003

[Urm03] Urman S.: Oracle9i PL/SQL Programming, Oracle Press, 2003

metalink.oracle.com

technet.oracle.com