Optymalizacja aplikacji OLAP pracujących w oparciu o SZBD Oracle9i

Witold Świerzy

wswier@sgh.waw.pl

Szkoła Główna Handlowa, Warszawa

Streszczenie

Artykuł zwraca uwagę na specyfikę strojenia aplikacji przetwarzania analitycznego (OLAP) i wspomagania podejmowania decyzji (DSS). System zarządzania bazą danych Oracle9i oferuje szereg zaawansowanych mechanizmów optymalizacji zapytań występujących w tego typu aplikacjach. Przedstawiono mechanizmy oraz zastosowania m.in.: rozszerzeń partycjonowania (In-List), optymalizacji wykonywania zapytań gwiaździstych (Star Query, Star Transformation) oraz automatycznego przepisywania zapytań (Query Rewrite) w celu wykorzystania perspektyw materializowanych.

Wprowadzenie

Wraz z rozwojem technologii baz danych, ilością gromadzonych danych, zmienia się również rola aplikacji pracujących w oparciu o Systemy Zarządzania Bazami Danych (SZBD). Tradycyjne aplikacje zorientowane są na gromadzenie informacji niezbędnych do poprawnego funkcjonowania instytucji. Równocześnie rośnie rola, jaką odgrywają analizy zebranych informacji - analizy takowe mogą dostarczyć wręcz niezbędnych informacji na temat sposobu funkcjonowania instytucji gromadzącej informacje lub pozwalać na przewidywanie przyszłych trendów w rozwoju dziedziny, jaką dana instytucja się zajmuje. Dopóki ilość informacji gromadzonych jest stosunkowo niewielka implementacja systemu analitycznego nie nastręcza większych trudności - system takowy może funkcjonować w oparciu o te same zbiory informacji (bazy danych), które są wykorzystywane w systemie je gromadzącym (OLTP). Jednakże, gdy ilość gromadzonych informacji gwałtownie przyrasta rosną również trudności w prawidłowej i wydajnej implementacji systemu je analizującego.

Jest to główna przyczyna powstania nowej gałęzi technologii baz danych - hurtowni danych. Bazy danych wykorzystywane jako hurtownie służą do składowania informacji wykorzystywanych do analizy. Na bazach takich pracują aplikacje OLAP (On Line Analytical Processing) - służą one do analizy zebranych informacji. Aplikacje te, oraz bazy danych na których pracują charakteryzują się następującymi własnościami:

1. Aplikacje OLAP głównie CZYTAJĄ dane, prawie nigdy ich nie modyfikują - baza danych będąca hurtownią informacji dla takiej aplikacji jest zorientowana na realizację długich i złożonych odczytów.
2. Na hurtowni danych nie pracuje jednocześnie zbyt duża liczba użytkowników, stąd nie występują tu charakterystyczne dla systemów OLTP problemy związane z wielodostępem.
3. Hurtownie danych podlegają okresowym załadunkom dużych ilości danych - załadunki te są w praktyce jedynymi okresami wysokiej aktywności DML takiej bazy danych.
4. W okresach pomiędzy załadunkami danych zmienność danych jest praktycznie zerowa - wpływa to na strategię archiwizacji i odtwarzania
5. O ile w systemach transakcyjnych bardzo ważny jest odpowiedni stopień normalizacji danych (redukuje to ilość redundancji i tym samym wpływa korzystnie na łatwość implementacji i wydajność modyfikacji danych), to podczas załadunku do hurtowni informacje bardzo często podlegają procesowi odwrotnemu - składujemy je w postaci wstępnie zagregowanej, aby zminimalizować ilość niezbędnych złączeń i sortowań.
6. O ile w systemach OLTP nie ma zazwyczaj kłopotu z ustaleniem wydajnego planu wykonania polecenia SQL (charakteryzują się one niejako z definicji ogromną selektywnością, stąd wykorzystanie klasycznych indeksów b-drzewowych jest wręcz normą), o tyle praca aplikacji OLAP jest bardziej złożona - plan wykonania odpowiednio dużego odczytu powinien być wręcz uzależniony od aktualnego rozkładu danych.

Powyższa charakterystyka aplikacji OLAP i hurtowni danych, na których aplikacje te pracują wymusza wręcz zupełnie inne niż w przypadku aplikacji OLTP podejście do strojenia wydajności całego systemu. W poniższej tabelce przedstawiamy krótkie porównanie wymagań tradycyjnych aplikacji OLTP do wymagań aplikacji OLAP pracujących w oparciu o SZRBD Oracle.

	Aplikacja OLTP	**Aplikacja OLAP**
Bloki bazy danych	dostosowane do rozmiaru transakcji (a więc zazwyczaj małe)	największe możliwe - duże bloki poprawiają wydajność odczytów
Obszar dzielony	Wysoka wydajność ze względu na konieczność dzielenia kodu SQL	brak wymagań
Bufory danych	Wysoka wydajność - użytkownicy takiej aplikacji odwołują się zazwyczaj do niewielkiego podzbioru bloków bazy danych, który jesteśmy w stanie efektywnie buforować	brak wymagań - większość operacji to pełne przeglądy tabel, nie jesteśmy w stanie efektywnie buforować wszystkich potrzebnych bloków bazy danych
Bufor dziennika powtórzeń	Wysoka wydajność - użytkownicy wykonują dużą ilość operacji DML, Bufor ten jest więc intensywnie wykorzystywany	Wysoka wydajność jest potrzebna jedynie okresowo, podczas załadunku danych, o ile nie odbywa się on z pominięciem bufora dziennika powtórzeń
Segmenty wycofania	Istotne jest osiągnięcie stanu, w którym nie pojawiają się oczekiwania na nagłówki segmentów przy rozpoczynaniu każdej kolejnej transakcji, a więc dbamy o odp. ich ilość	Ważne jest, aby podczas załadunku danych nie pojawiały się problemy z brakiem miejsca w segmentach wycofania, a więc dbamy o odp. rozmiar
Archiwizacja	Tryb ARCHIVELOG, kopie archiwalne ONLINE, odtwarzanie do momentu wystąpienia awarii	Tryb NOARCHIVELOG, wystarczą zimne kopie bezpieczeństwa tworzone każdorazowo po załadunku danych
Tryb optymalizacji	najczęściej regułowy	przeważnie kosztowy - uzależnia on plan wykonania poleceń SQL od konkretnego rozkładu danych

Wobec zwiększającego się ciągle zainteresowania aplikacjami OLAP w ciągu dalszym tej pracy zajmiemy się problematyką strojenia takich właśnie systemów opartych o SZRBD Oracle9i.

Nowości optymalizatora kosztowego Oracle9i

Jak już wykazaliśmy poprzednio właściwym dla aplikacji OLAP-owych trybem optymalizacji jest optymalizacja kosztowa. Zależy nam bowiem na najefektywniejszych planach poleceń SQL o bardzo różnej charakterystyce. Efektywne plany w takich sytuacjach są uzależnione od konkretnego rozkładu danych w przeszukiwanych tabelach. W części tej omówimy nowe elementy optymalizacji kosztowej SZRBD Oracle9i.

Pierwszą z nich jest nowy sposób specyfikowania optymalizacji kosztowej typu FIRST_ROWS. Optymalizacja ta w poprzednich wersjach Oracle opierała się częściowo na pewnych dość sztywnych założeniach (np. użyj indeksu, jeśli to jest tylko możliwe), co skutkowało nieraz bardzo złymi jej wynikami. W wersji 9i usunięto wszystkie sztywne założenia dotyczące optymalizacji tego typu, mamy ponadto, dzięki nowym dopuszczalnym wartościom parametru OPTIMIZER_MODE, możliwość podawania rozmiaru badanej próbki danych. Parametr ten w najnowszej wersji serwera dopuszcza dodatkowo następujące wartości: FIRST_ROWS_1 (optymalizacja w oparciu o jeden wiersz), FIRST_ROWS_10 (optymalizacja w oparciu o 10 wierszy), FIRST_ROWS_100 (100 wierszy), FIRST_ROWS_1000 (1000 wierszy). Oczywiście stare, znane z poprzednich wersji serwera są również dopuszczalne, przy czym wartość FIRST_ROWS jest również dopuszczalna - została ona zachowana dla zachowania zgodności wstecz i powoduje przełączenie optymalizatora w "klasyczny" tryb optymalizacji FIRST_ROWS. Parametr ten możemy oczywiście odpowiednio modyfikować dynamiczne w sesji (dopuszczalne jest np. polecenie ALTER SESSION SET OPTIMIZER_MODE=FIRST_ROWS_100). Dodatkowo w wersji 9i pojawiła się nowa wskazówka do optymalizatora: FIRST_ROWS(*n*), gdzie *n* jest *dowolną* liczbą całkowitą :

```
SELECT /*+ FIRST_ROWS(14) */
FROM EMP
WHERE ENAME LIKE 'S%';
```

Inną nowością optymalizatora kosztowego w Oracle9i jest nowa wartość parametru inicjalizacyjnego CURSOR_SHARING. Parametr ten wprowadzony w wersji serwera Oracle8i umożliwia współdzielenie planów wykonania poleceń SQL nawet wtedy, gdy różnią się one literałami:

1. Wartość EXACT. Jest to wartość domyślna powodująca wyłączenie zastępowania literałów. Buforowany w obszarze dzielonym plan wykonania może być użyty jedynie wtedy, gdy tekst odpowiadającego mu polecenia *w całości* pokrywa się z tekstem polecenia analizowanego.
2. Wartość FORCE. Wartość ta powoduje zastępowanie każdego wystąpienia literału w tekście analizowanego polecenia SQL. Buforowany w obszarze dzielonym plan wykonania będzie użyty nawet wtedy, gdy polecenie to odwołuje się w odpowiednich miejscach do zupełnie innych wartości.
3. Wartość SIMILAR. Wartość ta jest właśnie ową nowością Oracle9i. Umożliwia ona zastępowanie wystąpień literałów jedynie w tzw. przypadkach bezpiecznych. W celu sprawdzenia, czy mamy do czynienia właśnie z taką sytuacją system analizuje informacje statystyczne dotyczące odp. tabel i dostępnych indeksów.

Parametr ten może być ponadto modyfikowany dynamicznie zarówno dla całego systemu:

```
ALTER SYSTEM
SET CURSOR_SHARING = SIMILAR;
```

lub dla sesji :

```
ALTER SESSION
SET CURSOR_SHARING = SIMILAR;
```

Podstawą efektywnej pracy optymalizatora kosztowego są statystyki. Ich sporządzanie pociąga jednakże za sobą pewne koszty (system wykonuje przeglądy nieraz bardzo dużych tabel, sortuje zebrane informacje). Szczególnie kłopotliwy jest przypadek, w którym

analizowane tabele są duże i silnie zmienne. Ich zmienność wymusza bowiem regularne sporządzanie statystyk. Zazwyczaj redukujemy w takich sytuacjach ilość potrzebnych zasobów tworząc statystyki przybliżone, jednakże zawsze w takich sytuacjach stoimy przed problemem odpowiedniego oszacowania rozmiaru analizowanej próbki wierszy. Rozwiązaniem tego problemu mogą okazać się pewne nowości pakietu DBMS_STATS w Oracle9i. W pakiecie tym wprowadzono stałą DBMS_STATS.AUTO_SIZE zmuszającą procedury analizujące do samodzielnego oszacowania rozmiaru próbki analizowanych danych. Inną nowością tego pakietu są nowe opcje parametru METHOD_OPT (jest to parametr procedur analizujących w tym pakiecie). Opcje te to:

1. REPEAT - opcja ta użyta przy powtórnym tworzeniu histogramu dotyczącego pewnej kolumny spowoduje jego utworzenie z dokładnie taką samą dokładnością (podziałem). Jeżeli natomiast analizowana kolumna nie posiadała dotychczas histogramu, to nowy *nie zostanie w ogóle wygenerowany!*
2. AUTO - serwer Oracle sam decyduje o dokładności histogramu bazując nie tylko na informacjach o nierównomiernie rozłożonych wartościach, ale również na informacjach o sposobie korzystania z danej kolumny przez aplikację
3. SKEWONLY - serwer Oracle sam decyduje o dokładności histogramu opierając się jedynie o informacje o nierównomiernie rozłożonych wartościach, pomijając sposób wykorzystania danej kolumny przez aplikację. Opcja ta jest użyteczna wtedy, gdy np. sporządzamy histogram tuż po otwarciu bazy danych

Oto przykład wykorzystania nowości pakietu DBMS_STATS:

```
SQL> EXECUTE  DBMS_STATS  (  OWNNAME => 'SCOTT' ,  -
 2           ESTIMATE_PERCENT => DBMS_STATS.AUTO_SAMPLE_SIZE ,
                        -
 3         METHOT_OPT  => 'FOR ALL COLUMNS SIZE AUTO' )
```

Wybór odpowiedniej strategii indeksowania

Indeksy są jednym z podstawowych sposobów na przyspieszenie dostępu do danych, sprawdzają się jednakże tylko w określonych sytuacjach. Dwie podstawowe kategorie indeksów dzielimy je tu na dwie kategorie ze względu na ich wewnętrzną budowę): b-drzewowe i bitmapowe mają, jak się okazuje zupełnie przeciwstawne zastosowania. Przypomnijmy w tym miejscu, że indeksy b-drzewowe, jak sama nazwa wskazuje, stanowią grupę bloków danych zorganizowaną w b-drzewo, w liściach takiego indeksu znajdziemy informacje o poszukiwanych wartościach klucza i ROWID poszukiwanych wierszy. Indeksy bitmapowe są też zorganizowane w b-drzewa, jednakże w liściach takiego indeksu znajdziemy segmenty map bitowych. Każdy z takich segmentów poświęcony jest pojedynczej wartości klucza, bity w jego mapie bitowej są ustawione na 1, jeżeli odp. wiersz posiada poszukiwaną wartość, 0 gdy jej nie posiada. Bloki indeksu są zawsze odczytywane pojedynczo. Poniższa tabelka przypomina zastosowania obydwu typów indeksów.

Indeks b-drzewowy	**Indeks bitmapowy**
Odp. dla kolumn o wysokiej kardynalności	Odp. dla kolumn o niskiej kardynalności
Modyfikacje klucza mało kosztowne	Modyfikacje klucza bardzo kosztowne
Nieefektywne dla selekcji z alternatywą	Efektywne dla dow. warunków selekcji

Systemy OLAP pracujące na hurtowniach danych charakteryzują się dużą ilością poleceń SQL o niewielkiej selektywności, istnieją jednakże pojedyncze polecenia odrzucające nieraz znaczną ilość danych. Nad wyborem właściwego sposobu wykonania polecenia czuwa optymalizator kosztowy, który na szczęście nie jest sztywno oparty o składnię, lecz warto się zastanowić nad dwoma problemami :

1. Jakie indeksy (bitmapowe czy b-drzewowe) powinniśmy tworzyć na konkretnych tabelach?
2. Jak w praktyce sprawdzić, czy indeks jest w ogóle wykorzystywany (być może nie jest i można go po prostu usunąć)?

Rozwiązanie problemu z punktu 1 bazuje na powyższej tabelce - indeksy b-drzewowe tworzymy w tych nielicznych w systemach OLAP przypadkach, w których

a) polecenia charakteryzują się wysoką selektywnością (musimy więc znać aplikację)
b) przeszukiwane kolumny charakteryzują się wysoką kardynalnością

W pozostałych sytuacjach zastanawiamy się nad utworzeniem indeksów bitmapowych.
Problem nr 2 jest znacznie łatwiejszy do rozwiązania, o ile aplikacja pracuje w oparciu o serwer Oracle w wersji 9i. W tej wersji serwera możemy bowiem uaktywnić monitorowanie indeksów, a następnie sprawdzić, czy indeksy te były w ogóle wykorzystywane. Cała procedura postępowania przedstawiona jest poniżej, przy założeniu, że dysponujemy "klasyczną" szkoleniową tabelą EMP:

1. Tworzymy monitorowany indeks

```
SQL> CREATE INDEX I_EMP_EMPNO ON EMP(EMPNO) MONITORING USAGE;
```

Monitorowanie możemy również włączyć dla istniejących indeksów:

```
SQL> ALTER INDEX I_EMP_ENAME MONITORING USAGE;
```

2. Zezwalamy na normalną pracę aplikacji …

3. Po pewnym czasie wyłączamy monitoring

```
SQL> ALTER INDEX I_EMP_EMPNO NOMONITORING USAGE;
```

4. W perspektywie V$OBJECT_USAGE sprawdzamy, czy indeks był wykorzystywany:

```
SQL> SELECT INDEX_NAME, TABLE_NAME, USED
  2> WHERE INDEX_NAME = 'I_EMP_EMPNO';
```

Kolumna USED posiada dwie dopuszczalne wartości: 'YES', jeżeli indeks w okresie monitoringu został wykorzystany przynajmniej raz, oraz 'NO' w przeciwnym przypadku.

Nie należy zapominać, że indeksy są również strukturami pozwalającymi nieraz na poprawę efektywności złączeń, co w systemach OLAP odgrywa bardzo dużą rolę. Przy omawianiu tej funkcji indeksów nie wolno pominąć ważnej nowości serwera Oracle9i - mianowicie indeksów bitmapowych połączeniowych. Indeks taki, w przeciwieństwie do zwykłego zawiera informacje o wartościach kluczy nie z tabeli w oparciu o którą został utworzony lecz z tabeli, którą łączymy z tabelą indeksowaną. Dzięki temu możliwa jest optymalizacja złączenia dwóch tabel. Poniżej prezentujemy przykład utworzenia takowego indeksu:

```
SQL> create bitmap index i_emp_deptno on emp(d.deptno)
  2  from emp e, dept d
  3  where d.deptno=e.deptno;
```

Indeksy bitmapowe złączeniowe posiadają następującą charakterystykę:
1. Oferują dobrą efektywność złączeń
2. Nie zajmują zbyt wiele przestrzeni
3. Są użyteczne dla dużych tabel wymiarów w schematach gwiaździstych (a z takowych najczęściej korzystają aplikacje OLAP).

Wybór odpowiedniej strategii partycjonowania danych

Aplikacje OLAP pracują zazwyczaj na bardzo dużych zbiorach danych. Pojawiające się w takich przypadkach problemy są charakterystyczne dla wszystkich baz VLDB (Very Large Databases - Bardzo dużych baz danych) i obejmują kłopoty z konserwacją ONLINE oraz problemy z wydajnością poleceń SQL. Problemom tym możemy częściowo zaradzić dzieląc duże tabele na mniejsze segmenty zwane partycjami. Opcja partycjonowania Oracle jest dostępna od wersji 8.0, w wersji 9i oferuje możliwości partycjonowania tabel i indeksów według następujących schematów :
1. Zakresowego - tabela lub indeks są dzielone na partycje w oparciu o zakres wartości w kolumnach klucza partycji
2. Haszowego - tabela lub indeks są dzielone na partycje w oparciu o wartości funkcji mieszającej obliczanej dla kolumn klucza partycji
3. Złożonego - zakresowo-haszowego - w tym schemacie tabela lub indeks są dzielone na partycje w oparciu o zakresy wartości kolumn klucza, w ramach partycji głównych tworzymy podpartycje w oparciu o schemat haszowany
4. LIST - pozwala na podział tabeli lub indeksu w oparciu o listy dopuszczalnych wartości kolumn kluczowych dla każdej partycji

Ostatnia z wymienionych możliwości jest nowością w wersji 9i, stąd też poniżej przytaczamy przykład polecenia CREATE TABLE tworzącego tabelę w ten sposób podzieloną na partycje:

```
SQL> CREATE TABLE CUSTOMERS
  2  (CUST_ID NUMBER(5),
  3   CUST_NAME VARCHAR2(200),
  4   CUST_CITY VARCHAR2(200),
  5   CUST_STATE VARCHAR2(200) )
  6  PARTITION BY LIST (CUST_STATE)
  7  ( PARTITION P1 VALUES ('CALIFORNIA','TEXAS'),
  8    PARTITION P2 VALUES ('NEW YORK','OKLAHOMA'),
  9    PARTITION P3 VALUES ('NEW MEXICO','ALABAMA') );
```

Odpowiednio partycjonowane tabele i indeksy umożliwiają :
1. Zwiększenie dostępności danych podczas konserwacji struktur (konserwacja w danej chwili może obejmować tylko jedną lub kilka partycji zamiast całej tabeli lub indeksu)
2. Zwiększenie dostępności danych podczas awarii poprzez rozmieszczenie różnych partycji w różnych przestrzeniach tabel

3. Zwiększenie wydajności odczytu danych poprzez redukcję przeszukiwanych partycji tabeli
4. Zwiększenie wydajności odczytów indeksowych poprzez redukcję przeszukiwanych partycji indeksu
5. Zwiększenie efektywności złączeń poprzez stosowanie tabel równopartycjonowanych (tak samo podzielonych na partycje)
6. Zwiększenie efektywności załadunku danych (na tabelach partycjonowanych możemy wykonywać polecenia DML równolegle).

Ważne jest wobec tego wybranie odpowiedniej do zastosowania strategii partycjonowania tabel i indeksów.

Przyjmuje się, że partycjonowanie zakresowe odpowiednie jest dla tabel w których dane łatwo można podzielić na "historyczne" i "bieżące". Najczęściej wykorzystywanymi kolumnami klucza są więc w takich przypadkach kolumny typu DATE.

Partycjonowanie haszowe odpowiednie jest dla tabel w których nie można zdefiniować łatwo podziału na dane historyczne i bieżące, wartości klucza partycji są równomiernie rozłożone oraz spełniony jest przynajmniej jeden z poniższych warunków:

1. Dane są często wyszukiwane przy użyciu operatora "=" lub IN
2. Zależy nam na równoległym wykonywaniu poleceń DML

Partycjonowanie złożone stosujemy, gdy dane można łatwo podzielić wg klasycznego kryterium aktualności, a ponadto spełnione są następujące warunki:

1. Jedna z tabel jest partycjonowana wg schematu haszowanego
2. Tabela ta jest często łączona z tabelą, którą właśnie tworzymy - tworzona właśnie tabela powinna być wtedy "równopartycjonowana" - umożliwia to znaczną redukcję kosztów złączeń poprzez redukcję ilości łączonych ze sobą danych.

Partycjonowanie LIST stosujemy, gdy danych nie można łatwo podzielić na "bieżące" i "historyczne" oraz rozkład danych jest nieodpowiedni dla partycjonowania haszowanego.

Kolejną kategorią problemów jest właściwy wybór strategii partycjonowania indeksów. Przypomnijmy, że partycjonowane indeksy dzielimy na różne kategorie wg dwóch kryteriów :

1. Indeksy lokalne - globalne. Indeks nazywamy *lokalnym*, jeżeli istnieje powiązanie 1-1 pomiędzy jego partycjami a partycjami tabeli bazowej. W przeciwnym przypadku indeks nazywamy indeksem *globalnym.*
2. Indeksy z prefiksem - bez prefiksu. Indeks nazywamy *indeksem z prefiksem*, jeżeli jego podklucz jest kluczem partycji. W przeciwnym przypadku indeks nazywamy *indeksem bez prefiksu.* Np. indeks oparty o klucz (a,b,c,d) będzie indeksem z prefiksem, jeżeli podzielimy go na partycje wg kluczy (a),(a,b),(a,b,c). Ten sam indeks będzie indeksem bez prefiksu, jeżeli go podzielimy na partycje wg np. klucza (b,c)

Wobec tego teoretycznie istnieją cztery możliwości tworzenia indeksów partycjonowanych :

1. lokalny z prefiksem
2. lokalny bez prefiksu
3. globalny z prefiksem
4. globalny bez prefiksu

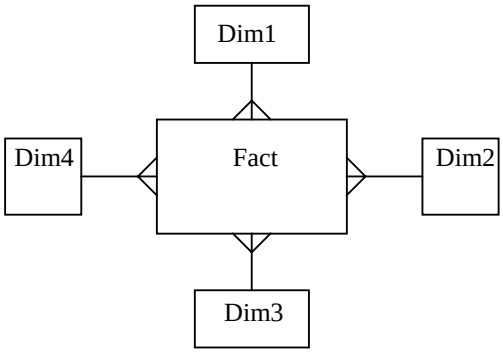
Możliwość czwarta jest czysto teoretyczna - indeks takowy nie dawałby żadnych korzyści zarówno od strony administracyjnej, jak i wydajnościowej - jest więc formalnie

niedozwolony. Dla pierwszych trzech kategorii przyjmuje się natomiast następującą metodę wyboru odpowiedniej strategii partycjonowania:

1. Jeżeli to tylko możliwe stosujemy indeksy lokalne z prefiksem. Umożliwiają one redukcję partycji przy przeszukiwaniach indeksowych (prefiks) i ułatwiają konserwację tabeli (lokalność)
2. W pozostałych przypadkach, jeżeli zależy nam bardziej na efektywności poleceń SQL niż ułatwieniach administracyjnych stosujemy indeksy globalne z prefiksem, jeżeli natomiast przeciwnie - stosujemy indeksy lokalne bez prefiksu (brak prefiksu oznacza znacznie mniejszą liczbę sytuacji w których można wyeliminować partycje indeksu podczas przeszukiwania indeksowego)

Zapytania gwiaździste i transformacja gwiaździsta

Jak się okazuje, większość aplikacji OLAP pracuje w oparciu o pewien specyficzny schemat encji zwany gwiazdą. Przykład takiego schematu prezentujemy poniżej:



Tabele Dim1,…,Dim4 są tzw. tabelami wymiarów - zazwyczaj są to słowniki (np. klienci, produkty, sprzedawcy itp.). Tabela Fact jest tzw. tabelą faktów - zawiera ona główne dane (np. informacje o sprzedaży klientom poszczególnych produktów przez poszczególnych sprzedawców). Aplikacja OLAP analizuje zawarte w tabeli faktów dane łącząc ją z poszczególnymi tabelami wymiarów. Przy zastosowaniu "klasycznych" algorytmów złączeń okazuje się zazwyczaj, że koszty takiej operacji są ogromne, gdyż system wykonuje taką operację niejako "po kolei":

1. złączenie fact-dim1
2. złączenie (fakt-dim1)-dim2
3. złączenie (fakt-dim1-dim2)-dim3
4. złączenie (fakt-dim1-dim2-dim3)-dim4

Mamy więc cztery operacje złączenia, każda następna jest zazwyczaj bardziej pracochłonna od poprzedniej (łączymy coraz to większe ilości danych). Zapytanie gwiaździste umożliwia redukcję ilości i kosztów złączeń, przy następujących założeniach:

1. Tabela faktów jest dostatecznie "gęsta", tzn. np. nie ma zbyt dużej liczby klientów, którzy nie nabywali zbyt wielu towarów sprzedawanych im przez pewną liczbę sprzedawców.
2. Tabele wymiarów są względnie małe i jest ich niezbyt wiele.

Algorytm takiego złączenia wygląda następująco:

1. Utwórz iloczyn kartezjański tabel Dim1xDim2xDim3xDim4
2. Utworzony iloczyn kartezjański złącz z centralną tabelą faktów

Okazuje się w wielu przypadkach, że koszt takiego złączenia (przy spełnieniu wcześniej wymienionych założeń) jest mniejszy, niż serii złączeń realizowanych metodami klasycznymi.
Złączenie takie jest możliwe jedynie w trybie optymalizacji kosztowej, możemy je ponadto wymusić stosując wskazówkę optymalizatora `/*+ STAR */`.

W niektórych przypadkach zapytanie gwiaździste może jednakże okazać się mało efektywną metodą złączeń. Przypadki te obejmują sytuacje w których:
1. Tabela faktów jest rzadka
2. Łączymy ją jednocześnie z dużą liczbą tabel wymiarów
3. Tabele wymiarów są duże
4. Na danych wybieranych z łączonych tabel dokonujemy dodatkowej selekcji

W sytuacjach takich czasem wygodnie jest dokonać *transformacji gwiaździstej* zapytania tak, aby nie wykonywać złączenia gwiaździstego. Rozpatrzmy następujący przykład :
1. Tabelą faktów jest tabela CLASSES
2. Łączymy ją z następującymi tabelami wymiarów : LOCATIONS, EMPLOYEES i CURSES
3. Na kolumnach LOC_ID, INSTR_ID, CRS_ID tabeli faktów, które służą do złączeń zostały założone indeksy bitmapowe
4. Wykonujemy następujące zapytanie :

```
SELECT  CL.START_DATE ,  L.CITY ,  CO.DESCRIPTION
FROM  CLASSES  CL ,  LOCATIONS  L ,   EMPLOYEES  E ,
      COURSES  CO
WHERE  CL.LOC_ID      =  L.LOC_ID
AND    CL.INSTR_ID  =  E.EMP_ID
AND    CL.CRS_ID      =  CO.CRS_ID
AND    L.STATE          IN  ( 'NY' , 'CA' , 'TX' )
AND    E.JOB               =  'SRPR_INSTRUCTOR'
AND    CO.DAYS        <  3;
```

Zwróćmy uwagę na fakt, że zapytanie w powyższej postaci nie skorzysta z indeksów bitmapowych założonych na kolumnach złączeniowych tabeli faktów. Dopiero transformacja gwiaździsta przetworzy jego tekst tak, aby owe indeksy mogły zostać wykorzystane. Poniżej prezentujemy przekonwertowaną przy pomocy transformacji gwiaździstej wersję tego polecenia :

```
SELECT  CL.START_DATE ,  L.CITY ,  CO.DESCRIPTION
FROM   CLASSES  CL ,  LOCATIONS  L ,  COURSES  CO
WHERE  CL.LOC_ID  =  L.LOC_ID
AND    CL.CRS_ID  =  CO.CRS_ID
AND    L.STATE     IN  ( 'NY' , 'CA' , 'TX' )
AND    CO.DAYS     <  3
AND    CL.LOC_ID   IN  (SELECT  LOC_ID  FROM  LOCATIONS
                          WHERE STATE IN
                          ('NY' , 'CA' , 'TX'))
AND    CL.INSTR_ID IN  (SELECT  EMP_ID  FROM  EMPLOYEES
                         WHERE  JOB  =  'SRPR_INSTRUCTOR')
AND    CL.CRS_ID   IN  (SELECT  CRS_ID  FROM  COURSES
                         WHERE  DAYS  <  3 );
```

Tak przetworzone polecenie będzie już mogło skorzystać z odp. indeksów bitmapowych dokonując selekcji potrzebnych danych.

Aby uaktywnić transformację gwiaździstą należy ustawić parametr inicjalizacyjny STAR_TRANSFORMATION_ENABLED na wartość TRUE. Parametr ten można również modyfikować dynamicznie na poziomie sesji. Nie ma jednakże sposobu na wymuszenie transformacji gwiaździstej - w systemie istnieje szereg ograniczeń na jej stosowanie. Często też optymalizator jej nie zastosuje, gdyż będzie dysponował lepszym planem wykonania.

Materializowane perspektywy, przepisywanie zapytań

Materializowane perspektywy wraz z mechanizmem przepisywania zapytań są jedną z poważniejszych nowości w serwerze Oracle8i. Umożliwiają one zdecydowaną poprawę wydajności systemu produkcyjnego, który np. nie został zaprojektowany pod kątem złożonych analiz dużych ilości danych. Implementację takiego systemu trudno jest zazwyczaj poprawić, tak, aby złożone raporty mogły być wykonywane na zbiorach wstępnie zagregowanych danych - wymagało by to znacznej ingerencji w kodzie źródłowym aplikacji. Mechanizm ten umożliwia przepisywanie poleceń SQL przez optymalizator “na gorąco” tak, aby odczytywały dane z tabel przechowujących je w odpowiedniejszej postaci. Aby go uaktywnić, należy spełnić następujące wymagania :

1. Należy dysponować zmaterializowanymi perspektywami, które przechowują dane odpowiadające wymaganiom systemu OLAP.
2. Oprócz tego należy uaktywnić optymalizację kosztową, gdyż przepisywanie zapytań działa tylko w tym trybie
3. Ponadto należy uaktywnić sam mechanizm - dokonujemy tego poprzez ustawienie następujących parametrów inicjalizacyjnych:
    a) QUERY_REWRITE_ENABLED - parametr logiczny; aby uaktywnić przepisywanie zapytań należy ustawić go na wartość TRUE
    b) QUERY_REWRITE_INTEGRITY - parametr sterujący minimalnym wymaganym poziomem spójności danych pobieranych przez nadpisane zapytanie; dopuszczalne wartości to :
        - ENFORCED - jest to wartość domyślna; zapytanie zostanie nadpisane na zmaterializowaną perspektywę tylko I wyłącznie wtedy, gdy optymalizator będzie w stanie zagwarantować spójność danych; w celu sprawdzenia spójności optymalizator posługuje się więzami integralności (zazwyczaj są to klucze obce) pracującymi w trybie ENABLE VALIDATE oraz świeżo zaktualizowanymi zmaterializowanymi perspektywami
        - TRUSTED - zapytanie zostanie nadpisane na zmaterializowaną perspektywę również jedynie wtedy, gdy optymalizator będzie w stanie zagwarantować spójność danych; w celu jej sprawdzenia posługuje się dodatkowo (oprócz struktur wykorzystywanych przy wartości FORCE) wszystkimi adekwatnymi w danej sytuacji zmaterializowanymi perspektywami, więzami integralności pracującymi w dowolnym trybie, lecz z ustawioną flagą RELY oraz wymiarami.
        - STALE_TOLERATED - zapytanie zostanie przepisane na zmaterializowaną perspektywę nawet wtedy, gdy nie zawiera ona aktualnych danych - spójność I aktualność danych nie jest więc kontrolowana

W celu lepszego zapoznania się z tym mechanizmem prześledźmy konkretny przykład: w aplikacji zaszyte jest następujące polecenie opierające się o dużą tabelę faktów CLASSES i tabelę wymiarów LOCATIONS:

```
SELECT  L.STATE, COUNT(*)  NUMBER_OF_OCCURRENCES
FROM CLASSES C, LOCATIONS L
WHERE C.LOC_ID = L.LOC_ID
GROUP BY L.STATE;
```

Ponieważ polecenie to za każdym razem pochłania zbyt dużo zasobów przy łączeniu tabeli CLASSES i LOCATIONS, a także podczas agregacji, to decydujemy się na zastosowanie mechanizmu przepisywania zapytań. W tym celu:

1. Tworzymy zmaterializowaną perspektywę:

```
CREATE MATERIALIZED VIEW CL_LOC_MV
BUILD IMMEDIATE
REFRESH FAST
ENABLE QUERY REWRITE
AS
      SELECT  L.STATE, COUNT(*)  NUMBER_OF_OCCURRENCES
      FROM CLASSES C, LOCATIONS L
      WHERE C.LOC_ID = L.LOC_ID
      GROUP BY L.STATE;
```

2. Uaktywniamy mechanizm przepisywania zapytań tak, jak zostało to opisane powyżej

3. Badamy plan wykonania wcześniej przedstawionego polecenia SQL. Jeżeli plan ten będzie zawierał przegląd zmaterializowanej perspektywy zamiast przeglądu tabel CLASSES i LOCATIONS to znaczy że zapytanie zostało nadpisane

Z mechanizmem przepisywania zapytań wiążą się ponadto pewne struktury dokumentujące w słowniku danych hierarchie i powiązania pomiędzy danymi zawartymi w hurtowniach danych. Dokumentacja taka pozwala optymalizatorowi kosztowemu na nadpisanie zapytania w dodatkowych sytuacjach, gdy parametr QUERY_REWRITE_INTEGRITY jest ustawiony na wartość TRUSTED. Ponadto istnieją również wskazówki do optymalizacji `/*+REWRITE*/` i `/*+NOREWRITE*/` pozwalające odpowiednio na przepisanie lub nie przepisanie polecenia SQL.

Systemy hybrydowe

Utrzymywanie niezależnego systemu dla potrzeb aplikacji OLAP jest niewątpliwie korzystne z wydajnościowego punktu widzenia, powoduje jednakże znaczne zwiększenie zapotrzebowania na środki finansowe niezbędne do utrzymania i konserwacji systemów informatycznych pracujących w naszej instytucji. Z tego też powodu często decydujemy się na budowę systemu hybrydowego, który obsługuje zarówno aplikacje transakcyjne jak i analityczne. Z punktu widzenia osoby odpowiedzialnej za strojenie wydajności takiego systemu jest to pewien (zazwyczaj znaczny) kompromis - jak mieliśmy okazję się wcześniej przekonać aplikacje OLTP mają zupełnie inne wymagania niż aplikacje OLAP. W ostatniej części prezentujemy kilka wskazówek umożliwiających poradzenie sobie z tą sytuacją.

1. Aplikacje OLTP powinny wykonywać polecenia sekwencyjnie, aplikacje OLAP w miarę możliwości równolegle. Osiągnąć to możemy stosując np. RESOURCE_MANAGERA - możemy utworzyć dwie grupy użytkowników - nazwijmy je OLTP i OLAP. Grupie OLAP przypisujemy odpowiednio wysoki stopień zrównoleglenia poleceń SQL, grupie OLTP - niski.
2. Aplikacje OLAP nie powinny łączyć się z bazą danych poprzez połączenia wielokanałowe - powoduje to zazwyczaj znaczący spadek wydajności ich pracy
3. Na tabelach wykorzystywanych przez aplikację OLTP nie powinny być zakładane indeksy bitmapowe - jeżeli mogłyby one poprawić wydajność aplikacji OLAP, to umożliwiamy jej pracę na innych tabelach (np. poprzez zmaterializowane perspektywy)
4. Aplikacje OLTP bardzo często "dobrze się zachowują" w trybie optymalizacji regułowej - często posługują się zmiennymi wiązanymi aby zwiększyć stopień dzielenia kodu SQL. W takich przypadkach należy uzależnić tryb optymalizacji od kategorii użytkownika. W tym celu możemy zastosować np. wyzwalacze ON LOGON
5. Aplikacje OLTP charakteryzują się zazwyczaj niewielkimi sortowaniami - już względnie małe wartości parametru SORT_AREA_SIZE umożliwiają wykonywanie większości sortowań w pamięci. Aplikacje OLAP przeciwnie - zachowują się lepiej z dużymi wartościami tego parametru. Aby oszczędzić pamięć możemy go modyfikować w zależności od kategorii podłączającego się użytkownika np. w wyzwalaczu ON LOGON. Z drugiej strony rozmiar ekstentu w tymczasowej przestrzeni tabel obliczamy wg wzoru K*SORT_AREA_SIZE - często oznacza to konieczność utworzenia dwóch różnych tymczasowych przestrzeni tabel dla dwóch różnych kategorii użytkowników.
6. Dane przetwarzane przez aplikacje OLAP winny być trzymane w blokach danych o dużym rozmiarze, z drugiej strony rozmiar bloku danych dla aplikacji OLTP uzależniamy od rozmiaru (zazwyczaj niewielkiego) pojedync zej transakcji. W Oracle9i możemy tworzyć w ramach pojedynczej bazy danych przestrzenie tabel z różnymi blokami danych - dane OLAP umieszczamy w przestrzeniach z dużymi blokami danych, segmenty wycofania wykorzystywane przez aplikację OLTP przeciwnie. Poniżej prezentujemy przykład polecenia tworzącego przestrzeń tabel o zadanym w nim rozmiarze bloku danych :

```
CREATE TABLESPACE OLAP_DATA
DATAFILE '/u01/olap_data01.dbf' SIZE 200M
BLOCKSIZE 32K;
```