

KBD2 — Komercyjne bazy danych

dr inż. Tomasz Traczyk

9. Rozwiązania dla hurtowni danych i VLDB

Listopad 2007

Copyright © Tomasz Traczyk
Instytut Automatyki i Informatyki Stosowanej Politechniki Warszawskiej
Materiały dydaktyczne przeznaczone są wyłącznie do indywidualnego użytku studiujących.
Rozpowszechnianie kopii bez pisemnej zgody autora jest zabronione.

Tabele zorganizowane indeksowo

Idea Index Organized Tables (IOTs)

- Przechowują dane w strukturze indeksu (nie ma osobnego segmentu tabeli, jak w *heap-organized tables*)
- Muszą mieć klucz główny
- Zapewniają szybszy dostęp do danych przy wyszukiwaniu wg klucza głównego

Zastosowanie

- Tabele, w których składniki klucza głównego zajmują duży procent wiersza
- Typowe zastosowania: GIS, hurtownie (wymiary), struktury generyczne (metadane) i in.

Tworzenie

```
CREATE TABLE nazwa_tabeli (kolumny) CONSTRAINT nazwa_pk PRIMARY KEY (klucz)  
ORGANIZATION INDEX TABLESPACE przestrzeń_indeksu [OVERFLOW TABLESPACE przestrzeń_danych]
```

Właściwości

- Nie ma zwykłego (fizycznego) ROWID
- Istnieje odpowiednik, tzw. logiczny ROWID (typu UROWID)
- IOTs nie mogą należeć do gron
- Partycjonowanie musi być wg. kolumn należących do klucza głównego
- Jeśli część wiersza poza kluczem głównym jest duża, można użyć klauzuli *OVERFLOW* w celu umieszczenia tych danych w osobnym segmencie

Kompresja indeksów

- Stosowana dla indeksów złożonych, gdy wartości początkowych kolumn indeksu (*prefix entries*) powtarzają się wielokrotnie
- Poważnie zmniejsza wielkość indeksu i operacje dyskowe, ale nieco zwiększa użycie CPU
- Polega na zastąpieniu wielokrotnych powtórzeń początkowych kolumn klucza jednym zapisem i odpowiednimi powiązaniem
- Tworząc indeks określa się liczbę grupujących („kompresowanych”) początkowych kolumn klucza
 - domyślnie wszystkie kolumny oprócz ostatniej
 - maksymalnie
 - * wszystkie kolumny dla indeksów nieunikalnych
 - * wszystkie kolumny oprócz ostatniej dla indeksów unikalnych
- Podobnej „kompresji” można poddać tabele zorganizowane indeksowo (IOTs)

Tworzenie

```
CREATE INDEX ... ON ... COMPRESS [liczba_kolumn]
```

Kompresja tabel

- Stosowana w bazach analitycznych do tabel *heap organized*, zwykle partycjonowanych
- Zmniejsza użycie dysku i pamięci, ale zwiększa użycie CPU
- Wskazana, gdy w tabeli jest dużo powtórzeń (np. liczne powtórzenia wartości długich kluczy obcych)
- Niewskazana, gdy przewiduje się dużo operacji DML
- Kompresowana może być cała tabela lub wybrane partycje

Tworzenie

```
CREATE TABLE ... COMPRESS
```

Partycjonowanie

Idea

- Pojedyncza tabela może być podzielona na partycje
- Podział na partycje może zależeć od wartości wybranych kolumn
- Optymalizator zapytań umie wykorzystywać tę informację do otwierania tylko niezbędnych partycji (*partition pruning*, *partition-wise joins*), np. na podstawie warunków w zapytaniu
- Operacje administracyjne mogą być wykonywane na każdej partycji osobno
- Możliwe jest zrównoleglenie operacji na partycjach tej samej tabeli
- Umiejętne zastosowanie partycjonowania może niemal uniezależnić wydajność zapytań od przyrostu danych
- Zalecane dla tabel powyżej 2GB
- Partycjonowanie jest przezroczyste dla aplikacji

Typy partycjonowania

- *By range* – wg zakresów wartości klucza partycjonowania
- *By list* – wg wartości wskazanych enumeratywnie (umożliwia grupowanie wartości niekolejnych)
- *By hash* – z użyciem funkcji mieszającej (odpowiednie gdy nie umiemy znaleźć kryterium równomiernego podziału)
- *Composite* – dwupoziomowe, najczęściej *range – hash* lub *range – list*

Partycjonowanie – przykłady

```
CREATE TABLE sprzedaz ...
PARTITION BY RANGE(data_sprzedazy)
(
  PARTITION sp2005 VALUES LESS THAN (TO_DATE(2006, 'YYYY')) TABLESPACE tsp2005,
  PARTITION sp2006 VALUES LESS THAN (TO_DATE(2007, 'YYYY')) TABLESPACE tsp2006,
  PARTITION sp_rest VALUES LESS THAN (MAXVALUE) TABLESPACE tsp_rest
)
```

```
CREATE TABLE sprzedaz ...
PARTITION BY LIST(id_koloru)
  PARTITION sp1 VALUES ('zielony', 'niebieski'),
  PARTITION sp2 VALUES ('czerwony', 'żółty'),
  PARTITION sp0 VALUES (DEFAULT)
```

```
CREATE TABLE sprzedaz ...
PARTITION BY HASH(id_sprzedazy)
  PARTITIONS 4 STORE IN (sp1, sp2, sp3, sp4)
```

Partycjonowanie indeksów

Indeksy partycjonowane lokalnie

- Partycje indeksu są zgodne z partycjami tabeli (*equipartitioning*)
- Jest to rozwiązanie łatwe do zarządzania i utrzymania przez DBMS
- Klucz partycjonowania nie musi być początkową częścią klucza indeksu (*nonprefixed index*)
- Należy dążyć do tego typu partycjonowania indeksów, zwłaszcza w bazach analitycznych

Indeksy partycjonowane globalnie

- Partycje indeksu nie są zgodne z partycjami tabeli
- Dostępne partycjonowanie *by range* i *by hash*
- Klucz partycjonowania musi być początkową częścią klucza indeksu (*prefixed index*)
- Jest to rozwiązanie bardziej uniwersalne: indeksy mogą być partycjonowane inaczej od tabel; stosowane raczej w OLTP

Chociaż partycjonowanie jest przezroczyste dla aplikacji, powinno być przewidywane już w czasie projektowania struktur, gdyż dopiero umiejętne zaprojektowanie kluczy głównych i indeksów umożliwi poprawny i efektywny podział na partycje.

Zmaterializowane perspektywy

- *Materialized views*, dawniej *snapshots*
- Zastosowanie:
 - replikacja asymetryczna w bazach rozproszonych
 - materializacja agregatów
- Tworzenie: `CREATE MATERIALIZED VIEW...`
- Czas odświeżania
 - synchronicznie: `REFRESH ON COMMIT`
 - asynchronicznie: `REFRESH NEXT sysdate+...`
- Tryb odświeżania
 - pełny
 - szybki: z dziennikiem migawki
 - * (`CREATE MATERIALIZED VIEW LOG` po stronie oryginału)
 - * tylko proste zapytania
 - forsowany: jak się da najlepiej
- Przepisywanie zapytań z udziałem zmaterializowanych perspektyw
 - automatycznie zastępuje wyliczenia agregatów sięgnięciem po ich materializację
 - potrzebne powiązanie zmaterializowanej perspektywy z tabelami za pomocą kluczy obcych typu `RELY`
 - potrzebne odpowiednie przywileje i włączenie parametru bazy lub sesji

Indeksy bitowe (Bitmap Indexes)

- Warunki efektywnego działania indeksów bitowych
 - klucz zawiera niewiele wartości o dowolnej częstotliwości
 - tabela jest duża
 - równościowe warunki z udziałem klucza w WHERE (wiele takich warunków)
- Warunki, gdy indeksy nie działają efektywnie
 - klucz zawiera wiele wartości
 - tabela jest mała
 - warunki nierównościowe (zakresy)
- Utrzymywanie indeksów
 - automatyczne
 - bardzo spowalnia DML
- Wykorzystanie indeksów
 - decyduje optymalizator zapytań
 - wyszukiwanie przez operacje logiczne na bitach kodujących wartości kluczy
- Zastosowanie
 - w zasadzie wyłącznie w bazach analitycznych
 - hurtownie – indeksowanie tabel faktów po wymiarach

Tworzenie

```
CREATE BITMAP INDEX wykladowcy_bidx ON wykladowcy ( tytul )  
TABLESPACE user_index;                               tytul przybiera niewiele wartości
```

Indeksy złączeniowe (Bitmap Join Indexes)

- Przeznaczone do użycia w bazach analitycznych (struktury gwiazdzone)
- Indeksujące tabelę faktów wartościami z tabeli wymiarów
- Stosowane, gdy wyszukiwania dotyczą deskryptora, a nie identyfikatora wymiaru
- W indeksie mogą być wyłącznie kolumny wymiarów
- Warunek złączenia musi dotyczyć klucza głównego lub unikalnego (całego)

Tworzenie

```
CREATE BITMAP INDEX sprzedaz_kolor_idx ON sprzedaz(kolory.nazwa)  
FROM sprzedaz, kolory  
WHERE sprzedaz.id_koloru = kolory.id_koloru;
```

Optymalizacja zapytań gwiazdzystych

Warunki

- Hurtownia danych – struktura gwiazdy
- Każdy klucz obcy do wymiarów w tabeli faktów zaindeksowany indeksem bitowym lub istnienie odpowiedniego indeksu złączeniowego
- Włączony parametr bazy `STAR_TRANSFORMATION_ENABLED`
- Zapytanie do tabeli faktów ze złączeniami do tabel wymiarów

Działanie

- *Star transformation* – automatyczne przepisywanie zapytań gwiazdzystych
- Fazy
 - podzapytanie do tabeli faktów z użyciem indeksów bitowych (bez złączenia!)
 - złączenie (*semijoin*) wyniku tego podzapytania z tabelami faktów (małymi)
- Dla dużych zbiorów danych uniknięcie „prawdziwego” złączenia daje znaczny wzrost wydajności

Tabele zewnętrzne (external tables)

- Zastosowanie
 - ładowanie danych zewnętrznych, np. procesy ETL w bazach analitycznych
 - sporadyczny dostęp do wielkich mało zmiennych zbiorów danych
- Działanie
 - zewnętrzny plik danych jest „podłączany” do bazy jak tabela
 - dane z pliku są dostępne przez zwykłe zdania SQL
 - wydajność zapytań jest oczywiście znacznie gorsza
- Sposób definiowania
 - `CREATE TABLE ... ORGANIZATION EXTERNAL`
 - określenie struktury pliku podobne jak dla narzędzia SQL*Loader

Przenośne przestrzenie tabel

Zastosowanie

- Najszybsza migracja dużych porcji danych między bazami Oracle, np. w procesach ETL

Warunki

- Bazy Oracle w dostatecznie nowej wersji i ta sama strona kodowa
- Ten sam system operacyjny lub konieczność dodatkowych manipulacji przy przenoszeniu

Realizacja

1. Umieszczenie przenoszonych danych w wydzielonej przestrzeni tabel
 - kopiowanie danych, np. `CREATE TABLE AS SELECT`
 - `ALTER TABLESPACE przestrzeń READ ONLY;`
2. Eksport metadanych
`exp TRANSPORT_TABLESPACE=y TABLESPACES=przestrzeń FILE=plik.dmp`
3. Kopiowanie plików danych przestrzeni przestrzeń i pliku plik.dmp do systemu docelowego
4. Import metadanych
`imp TRANSPORT_TABLESPACE=y DATAFILES='pliki_danych' TABLESPACES=przestrzeń FILE=plik.dmp`

Data Pump

Idea

- Narzędzia do szybkiego przenoszenia danych między bazami Oracle
- Szybsze od tradycyjnych `exp` i `imp` oraz mające większe możliwości sterowania procesem
- Ze wsparciem dla nowych możliwości baz danych Oracle10g (poza XML)
- Z możliwością zmiany przypisania przestrzeni tabel

Składniki

- Specjalne narzędzia do eksportu i importu danych: `expdp` i `impdp` (niekompatybilne z `exp` i `imp`!)
- Pakiet `DBMS_DATAPUMP` zawierający *Data Pump API*
- Pakiet `DBMS_METADATA` umożliwiający operacje na metadanych ze słownika DBMS

Działanie

- Pliki danych i logi są tworzone na serwerze
- Przepływ danych odbywa się przez *direct path load/unload* lub mechanizm tabel zewnętrznych