TEHNICI DE OPTIMIZARE SQL

Suport Laborator Saptămâna 07

FEAA Master SIA/SDBIS

Cuprins

Tehnici de optimizare SQL

La laboratorul de data trecută am început să discutăm despre tehnici de optimizare ale server-ului Oracle. Am acoperit câteva astfel de tehnici ce ţin de instanţa Oracle, cum ar fi configurarea zonelor de memorie, crearea de noi grupuri redolog, dar si obţinerea raportului AWR.

Optimizarea SQL, contrar a ceea ce s-ar putea crede, nu intră doar în sarcina DBA-ului, ci mai ales a dezvoltatorilor de aplicaţii. În general, cei care implementează logică în aplicaţie, bazată pe SQL, ar trebui să aibă cunoştinţe solide legate de optimizarea acestor fraze SQL.

Pregătirea platformei de test

Schimbăm puţin registrul, de la aplicaţia de tip Facebook cu care ne-am jucat laboratorul trecut, la o aplicaţie de gestiune a vânzărilor. Pentru a crea şi popula schema VANZARI, va trebui să descărcaţi fişierul *perf_setup.sql* şi să-l rulaţi într-o fereastră de SqlPlus.

```
SQL> @C:\Users\talek\perf_setup.sql
Connecting as SYS...
Connected.

Cleanup VANZARI environment...

Create VANZARI schema.
The password is "Vanzari123"
Grant rights to VANZARI.

Connecting as VANZARI...
Connected.

Creating JUDETE table...
Creating CODURI_POSTALE table...
Creating CLIENTI table...
Creating PERSOANE table...
Creating PERSCLIENTI table...
Creating PRODUSE table...
Creating FACTURI table...
Creating LINIIFACT table...
Creating INCASARI table...
Creating INCASFACT table...
Creating WORKLOAD_PRODUCER procedure...

Populating VANZARI tables: 1000 clients and 1000 products...

Done.
```

Planul de execuţie

Planul de execuţie este cel care ne spune felul în care o anumită frază SQL este executată de optimizor.

Există două tipuri de planuri de execuţie: cel *estimat*, atunci când instrucţiunea SQL poate să nu fie neaparat executată şi planul real care implică rularea instrucţiunii. De preferat ar fi ca

întodeauna să ne ghidăm după planul real, dat fiind că cel estimat s-ar putea să fie diferit de cel real.

Iată un exemplu. Să zicem că avem nevoie de o nouă tabelă LINIIFACT2, cu acelaşi conţinut ca LINIIFACT, dar cu o coloană nouă denumită ID, cu valori de la 1 la N şi cu rol de cheie primară. Simplu! Putem face asa.

```
SQL> create table liniifact2 as select to_char(rownum) id, liniifact.* from liniifact;

Table created.

SQL> alter table liniifact2 add constraint liniifact2_pk primary key (id);

Table altered.
```

Mai departe, declarăm o variabilă, o iniţializăm şi încercăm să obţinem planul de execuţie estimat pentru o interogare simplă cu COUNT(*).

```
SQL> variable n number
SQL> exec :n := 5;

PL/SQL procedure successfully completed.

SQL> set autotrace traceonly
SQL> select count(*) from liniifact2 where id = :n;


Execution Plan
----------------------------------------------------------
Plan hash value: 3483566290

-------------------------------------------------------------------------------------------
| Id  | Operation          | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |               |     1 |     7 |     1   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE    |               |     1 |     7 |            |          |
|*  2 |   INDEX UNIQUE SCAN| LINIIFACT2_PK |     1 |     7 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("ID"=:N)


Statistics
----------------------------------------------------------
         27  recursive calls
          0  db block gets
        406  consistent gets
        384  physical reads
          0  redo size
        542  bytes sent via SQL*Net to client
        551  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
```

```
          2  sorts (memory)
          0  sorts (disk)
          1  rows processed

SQL> set autotrace off
```

Iată, conform planului estimat de mai sus, optimizorul ar face o operaţie de tip *INDEX UNIQUE SCAN* pe indexul *LINIIFACT2_PK*.

Să vedem acum planul de execuţie real:

```
SQL> set serveroutput off
SQL> select count(*) from liniifact2 where id = :n;

  COUNT(*)
----------
         1

SQL> select * from table(dbms_xplan.display_cursor);

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------
SQL_ID  6fzk8p8mz1pz8, child number 0
-------------------------------------
select count(*) from liniifact2 where id = :n

Plan hash value: 3771462659

-------------------------------------------------------------------------------------------
| Id  | Operation             | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |               |       |       |   106 (100)|          |
|   1 |  SORT AGGREGATE       |               |     1 |     7 |            |          |
|*  2 |   INDEX FAST FULL SCAN| LINIIFACT2_PK |     1 |     7 |   106   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(TO_NUMBER("ID")=:N)
```

Conform planului de execuţie real, s-ar părea că optimizorul alege o operaţie de tip *INDEX FAST FULL SCAN* şi nicidecum *INDEX UNIQUE SCAN*. Observaţi că a fost necesar să rulăm efectiv instrucţiunea SQL şi abia apoi să folosim *DBMS_XPLAN* pentru a obţine planul de execuţie real.

1. *Care este semnificaţia coloanei "Cost" din planul de execuţie?*
2. *De ce cele două planuri de execuţie nu sunt identice? Ce a estimat greşit optimizorul în primul caz?*

Planuri de execuție eficiente vs. planuri ineficiente

Știm deja cum obținem un plan de execuție: fie folosind pachetul *DBMS_XPLAN,* fie din fișierul "trace", dacă a fost activată generarea unui astfel de fișier (am văzut la laboratorul de data trecută cum facem asta).

Problema care se pune e dacă planul de excuție pe care l-am obținut este eficient sau nu. În general, ne uităm pe acele SQL-uri care au timpi mari de execuție, eventual pe acele operații de care se plâng utilizatorii că se "mișcă" ardelenește.

Să luăm în considerare următorul SQL, cu planul de execuție aferent:

```
VANZARI@SQL> set serveroutput off
VANZARI@SQL> select /*+ gather_plan_statistics */ nrfact from liniifact where codpr=2000;

    NRFACT
----------
      4207
      4016
      4019

...

181 rows selected.

VANZARI@SQL> select * from table(dbms_xplan.display_cursor(format => 'iostats last'));

PLAN_TABLE_OUTPUT
--------------------
SQL_ID  g5rjyba2tqbgs, child number 0
-------------------------------------
select /*+ gather_plan_statistics */ nrfact from liniifact where
codpr=2000

Plan hash value: 770896586

-----------------------------------------------------------------------------------------------------
| Id  | Operation         | Name      | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |      1 |        |    181 |00:00:00.01 |     581 |    560 |
|*  1 |  TABLE ACCESS FULL| LINIIFACT |      1 |    169 |    181 |00:00:00.01 |     581 |    560 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("CODPR"=2000)
```

Ce-i cu notația aceea ciudată /+ ... */ din fraza select?*

Prima etapa ar fi să "descifrăm" acest plan de execuție. Avem un *SQL_ID* și un "*child number*" cu ajutorul cărora putem identifica, unic, un plan de execuție în zona de memorie "*library cache*".

	*Unde se află zona de memorie "library cache"?*

Apoi, *DBMS_XPLAN*-ul, ne oferă şi textul frazei SQL. De multe ori, doar uitându-ne la interogare ne putem da seama ce se vrea a se obţine din punctul de vedere al logicii aplicaţiei. Pentru alte interogări "stufoase" cu zeci de "join"-uri s-ar putea să ne declarăm repede învinşi de sistem. În cazul de faţă e clar că intenţia ar fi de a "trage" toate facturile pe care apare produsul ce are codul 2000.

În sfârşit, planul de execuţie ne apare ca o înlanţuire ierarhică de operaţii pe care optimizorul le efectuează. În cazul de faţă, lucurile sunt simple. Avem operaţia rădăcină care nu e altceva decât SELECT-ul nostru. Oricum, de una singură, operaţia asta nu are ce să facă pentru că aşteptă date care sa-i fie trasnferate din operaţia copil, cea cu identificatorul 1. Aceasta este una de tip "TABLE ACCESS FULL", adică va parcurge toate liniile din LINIIFACT şi va returna părintelui (operaţia 0) doar pe acelea care îndeplinesc condiţia din predicat.

	*Care e semnificaţia coloanelor: E-Rows, A-Rows şi Buffers?*

Este eficient acest plan de execuţie? Depinde cum definim această eficienţă, dar în principiu, eficient ar trebui să însemne că executăm instrucţiunea SQL folosind cât mai puţine resurse: CPU, disk, memorie şi reţea. Dacă vrem să fim foarte riguroşi, ar trebui să le luăm pe toate în calcul. În practică, ar fi mult prea costisitor şi ne-ar lua prea mult timp să facem o astfel de analiză. Sigur, am putea lua în considerare costul planului de execuţie. Problema e că un astfel de cost nu ne spune mare lucru: e doar o cifra şi nu putem compara aceste costuri pentru fraze SQL distincte, ci doar dacă fraza SQL rămâne nemodificată, iar optimizarea se referă la crearea de structuri de optimizare adiacente de tipul indecşilor, view-urilor materializate şi aşa mai departe. În plus, s-ar putea să avem un cost mic al planului de execuţie dar SQL-ul să se execute foarte ineficient. Este un caz frecvent atunci când avem distribuţii neuniforme ale datelor iar predicatele din fraza SELECT conţin variabile de tip "bind". Dacă nu aţi înţeles mare lucru din această explicaţie, nu-i bai. Ce trebuie să reţineţi e că optimizarea bazată doar pe costurile din planurile de execuţie nu e întotdeauna fezabilă.

Prin urmare, în practică folosim optimizarea bazată pe numărul de citiri logice (de blocuri) efectuate de un SQL. Cu cât îs mai puţine cu atât mai bine. De ce? Simplu: un SQL care vâjâie mai puţine blocuri e mai scalabil în medii puternic concurenţiale (accesul la blocul Oracle este serializat), o citire logică implică treabă făcută de CPU, deci reducând citirile logice eliberăm şi din "stress"-ul pe procesor şi, desigur, citirile logice pot avea în spate citiri fizice direct de pe disc, ceea ce înseamnă că mai puţine citiri logice implică o probabilitate mai mică de a efectua citiri lente de pe disc. În plus, obţinerea indicatorului de citiri logice la nivel de SQL este relativ uşor de obţinut. E vorba de coloana "buffers" din planul de execuţie afişat de DBMS_XPLAN.

Iată, pentru interogarea noastră, pentru a afla toate facturile pe care apare produsul cu identificatorul 2000, s-au efectuat 581 de citiri logice pentru blocurile oracle care conţin datele din LINIIFACT. Acum, corelând cu numărul de linii accesate, ajungem la o rată de 581/181=3.2 blocuri logice per înregistrare returnată, ceea ce nu e rău. Din practică, ce-i sub 5 citiri logice per

inregistrare este considerată o rată buna, ce-i până în 15 este considerat acceptabil, iar peste deja considerăm că e loc pentru a fi optimizat. Sigur, acestea sunt cifre orientative. De la caz la caz, discuţia privind aceste "praguri" poate fi nuanţată.

- *Pe exemplul nostru, în ce condiţii rata de blocuri logice citite per înregistrare returnată ar putea să devină o problemă?*
- *Ce se întâmplă cu această rată dacă dorim să aflăm pe câte facturi se află produsul care are codul 2000? Este vreo problemă? Explicaţi!*

Factori care influențează numărul de citiri logice

Ori de câte ori estimăm eficienţa unei fraze SQL folosind numărul de blocuri logice citite trebuie să luăm în considerare doi factori ce ar putea conduce la o "inflamare" artificială a numărului de astfel de citiri.

Citiri consistente

Dacă blocurile citite sunt modificate de alte sesiuni, atunci Oracle va reconstitui datele consistent cu nivelul de izolare a tranzacţiei, folosind segmentele UNDO.

Pentru a exemplifica, deschideţi două ferestre CMD din care rulaţi câte un sqlplus, conectându-vă la schema VANZARI. Parola, în cazul în care aţi uitat-o, e "Vanzari123". Din una din sesiuni rulaţi următoarele comenzi:

```
VANZARI@SQL> set serveroutput off
SQL> select /*+ gather_plan_statistics */ count(nrfact) from liniifact where codpr=2000;

COUNT(NRFACT)
-------------
          181

VANZARI@SQL> select * from table(dbms_xplan.display_cursor(format => 'basic iostats last -cost'));

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------
EXPLAINED SQL STATEMENT:
------------------------
select /*+ gather_plan_statistics */ count(nrfact) from liniifact where
codpr=2000

Plan hash value: 1601902421

-----------------------------------------------------------------------------------------------
| Id  | Operation          | Name      | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |      1 |        |      1 |00:00:00.02 |     568 |    560 |
|   1 |  SORT AGGREGATE    |           |      1 |      1 |      1 |00:00:00.02 |     568 |    560 |
|*  2 |   TABLE ACCESS FULL| LINIIFACT |      1 |    169 |    181 |00:00:00.01 |     568 |    560 |
-----------------------------------------------------------------------------------------------
```

Nimic spectaculos. S-au "vâjât" 568 de blocuri, cam la fel cu în primul exemplu. Acum, din ce-a de-a doua fereastra/sesiune rulaţi următoarea comandă:

```
VANZARI@SQL> update liniifact set cantitate = 10;
```

```
168520 rows updated.
```

Am efectuat practic un UPDATE de sus până jos. Revenim acum la prima sesiune, rulăm din nou SELECT-ul cu pricina şi tragem o ochiadă pe planul de execuţie.

```
VANZARI@SQL> select /*+ gather_plan_statistics */ count(nrfact) from liniifact where codpr=2000;

COUNT(NRFACT)
-------------
          181

VANZARI@SQL> select * from table(dbms_xplan.display_cursor(format => 'basic iostats last -cost'));

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------
EXPLAINED SQL STATEMENT:
------------------------
select /*+ gather_plan_statistics */ count(nrfact) from liniifact where
codpr=2000

Plan hash value: 1601902421

-------------------------------------------------------------------------------------------
| Id  | Operation          | Name      | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |      1 |        |      1 |00:00:00.16 |     178K|
|   1 |  SORT AGGREGATE    |           |      1 |      1 |      1 |00:00:00.16 |     178K|
|*  2 |   TABLE ACCESS FULL| LINIIFACT |      1 |    169 |    181 |00:00:00.04 |     178K|
-------------------------------------------------------------------------------------------
```

Deşi avem de-a face cu acelaşi SQL, numărul de citiri logice este cu mult mai mare. Deci, ori de câte ori obţineţi un număr nefiresc de mare de citiri logice, e bine să vă asiguraţi că nu aţi dat fix peste speţa aceasta, când datele citite sunt de fapt modificate semnificativ din alte sesiuni.

Configurarea operației FETCH

Operaţia prin care înregistrările sunt efectiv "pasate" spre client poartă denumirea de FETCH. Sigur, dacă interogarea returnează foarte multe înregistrări este necesară invocarea de mai multe ori a operaţiei FETCH, până când toate înregistrările au fost returnate. În sqlplus acest lucru se întâmplă transparent, dar in PL/SQL sau într-un alt limbaj care foloseşte librăriile client Oracle se poate controla exact cum să se facă operaţiile FETCH. În general, printr-o singură operaţie FETCH sunt returnate mai multe înregistrări odată. Totuşi acest aspect este configurabil şi se poate specifica numărul de linii de împachetat într-o operaţie FETCH. Acest lucru afecteză numărul de citiri logice, aşa cum vom exemplifica mai jos. Conectaţi-vă la schema VANZARI şi rulaţi următoarele comenzi:

```
VANZARI@SQL> set serveroutput off
VANZARI@SQL> show arraysize
arraysize 15
VANZARI@SQL> select /*+ gather_plan_statistics */ denpr from produse;

…

Produs2821
PrestServicii2822
PrestServicii2823
PrestServicii2824
PrestServicii2825
```

```
1000 rows selected.

SQL> select * from table(dbms_xplan.display_cursor(format => 'basic iostats last -cost'));

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------
EXPLAINED SQL STATEMENT:
------------------------
select /*+ gather_plan_statistics */ denpr from produse

Plan hash value: 3588227231

-----------------------------------------------------------------------------------------
| Id  | Operation         | Name    | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |         |      1 |        |   1000 |00:00:00.01 |      73 |
|   1 |  TABLE ACCESS FULL| PRODUSE |      1 |   1000 |   1000 |00:00:00.01 |      73 |

-----------------------------------------------------------------------------------------
```

Numărul de înregistrări returnate printr-o operaţie FETCH în sqlplus e dat de parametrul "*arraysize*". Implicit, este 15. Vom seta acest parametru pe valoarea 1 şi vom analiza impactul asupra numărului de blocuri logice citite:

```
VANZARI@SQL> set arraysize 1
VANZARI@SQL> select /*+ gather_plan_statistics */ denpr from produse;

…

Produs2821
PrestServicii2822
PrestServicii2823
PrestServicii2824
PrestServicii2825

1000 rows selected.

SQL> select * from table(dbms_xplan.display_cursor(format => 'basic iostats last -cost'));

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------
EXPLAINED SQL STATEMENT:
------------------------
select /*+ gather_plan_statistics */ denpr from produse

Plan hash value: 3588227231

-----------------------------------------------------------------------------------------
| Id  | Operation         | Name    | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |         |      1 |        |   1000 |00:00:00.01 |     506 |
|   1 |  TABLE ACCESS FULL| PRODUSE |      1 |   1000 |   1000 |00:00:00.01 |     506 |
-----------------------------------------------------------------------------------------
```

Observăm că atunci când optăm pentru o dimensiune mică a numărului de linii returnate printr-o operaţie FETCH, constatăm o "inflaţie" de blocuri logice citite. E important ca atunci când se investighează probleme de performanţă cu diferite SQL-uri executate de o aplicaţie, testele să fie făcute cu aceeaşi valoarea a parametrului "arraysize" ca cea configurată în aplicaţie.

Optimizarea interogărilor cu selectivitate slabă

Ori de câte ori avem de-a face cu un predicat care selectează o pondere mare de înregistrări din total, spunem că respectivul predicat are o slectivitate slabă.

Operația FULL TABLE SCAN

Deja ne-am întâlnit cu această operație în planurile de execuție cu care ne-am jucat până acum. E vorba de: *TABLE ACCESS FULL*. Prin această operație, optimizorul alege să parcurgă întreaga tabelă, bloc cu bloc și înregistrare cu înregistrare. Compară condiția din predicat și returnează doar acele linii care satisfac acea condiție.

Cum putem optimiza o astfel de operație de tip FULL SCAN? Nu avem prea multe opțiuni, dar putem încerca să aplicăm câteva "șmecherii".

Parametrul db_file_multiblock_read_count

Acest parametru indică numărul de blocuri pe care Oracle ar trebui să le citească dintr-odată de pe disc, ori de câte ori intervine o operație de tip FULL SCAN (poate fi aplicată și la indecși).

Care este valoarea acestui parametru pe baza de date Oracle cu care lucrați acum?

Oracle nu garantează că va și reuși să citească dintr-odată numărul de blocuri configurat prin acest parametru, dar nu intrăm în detalii acum.

Păstrarea datelor în memorie

Știm deja că in SGA (System/Shared Global Area) există o zonă de memorie denumită "DB Buffer Cache" în care, implicit, sunt ținute blocurile citite din fișierele de date, blocuri care sunt frecvent accesate. Ei bine, pe lângă această structură de memorie Oracle mai oferă zona KEEP și zona de memorie RECYCLE. În KEEP ar trebui să fie puse acele tabele relativ mici, dar care dorim să fie permanent în memorie, chiar dacă sunt citite destul de rar. În zona RECYCLE ar trebui puse în general tabele folosite rar și care au multe înregistrări. Spre exemplu o tabela istoric cu facturile de acum 10 ani.

- *Cum pot fi configurate zonele KEEP și RECYCLE?*
- *De ce tabelele mari, folosite rar/excepțional ar trebui să fie configurate cu clauza RECYCLE?*

Configurăm zona de memorie în care datele să fie duse la nivelul segmentului, prin intermediul atributului BUFFER_POOL din clauza STORAGE și a clauzei CACHE. Spre exemplificare,

să zicem că vrem ca tabela CLIENTI să o păstrăm cât mai mult în memorie. Putem face asta la nivelul segmentului CLIENTI.

```
VANZARI@SQL> alter table clienti storage (buffer_pool keep) cache;

Table altered.
```

Ce face în plus setarea CACHE față de atributul BUFFER_POOL?

Pentru a aduce datele în memorie e suficient să citim toată tabela CLIENTI.

```
VANZARI@SQL> alter system flush buffer_cache;

System altered.

VANZARI@SQL> set autotrace traceonly statistics
VANZARI@SQL> select * from clienti;

1000 rows selected.


Statistics
----------------------------------------------------------
         73  recursive calls
          0  db block gets
        198  consistent gets
         22  physical reads
          0  redo size
      52723  bytes sent via SQL*Net to client
       1277  bytes received via SQL*Net from client
         68  SQL*Net roundtrips to/from client
          6  sorts (memory)
          0  sorts (disk)
       1000  rows processed
```

Prima instrucţiune invalidează/şterge tot ce se afla în "buffer cache". Apoi se poate observa că o citire de sus până jos a tabelei CLIENTI a necesitat 22 citiri fizice de pe disc.

Putem confirma că datele au fost "cache"-uite folosind comenzile de mai jos (va trebui să descărcaţi scriptul "*clienti_buffer_stats.sql*" de pe portal):

```
VANZARI@SQL> @C:\Users\talek\lab_04\clienti_buffer_stats.sql

BUFFER_ OWNER      OBJECT_NAM OBJECT_BLOCKS CACHED_BLOCKS
------- ---------- ---------- ------------- -------------
KEEP    VANZARI    CLIENTI               16            14
```

De ce avem mai puţine blocuri "cache"-uite decât numărul total de blocuri alocat segmentului CLIENTI?

Partiționarea datelor

Oracle permite partiţionarea tabelelor şi a indecşilor după diferite criterii. Asta înseamnă ca unei tabele ar putea să-i corespună mai multe segmente, câte unul pentru fiecare partiţie. Pentru aplicaţii, partiţionarea este transparentă. Tabela partiţionată arată de la exterior ca o tabelă obişnuită. În general partiţionarea este folosită pentru mentenanţa tabelelor mari, dar ea poate avea un rol important şi în optimizare. Dacă o interograre pe o tabelă partiţiontă are un predicat care filtrează după cheia de partiţionare, doar partiţiile relvante vor fi accesate de respectiva interogare. Să exemplificăm pe schema VANZARI. Mai întâi să cream o tabelă partiţionată pe modelul celei de FACTURI.

```
CREATE TABLE facturi_rdf_part
PARTITION BY RANGE (datafact)
(
  PARTITION part_facturi_1
    values less than (to_date('01/09/2015','dd/mm/yyyy'))
    TABLESPACE USERS STORAGE(BUFFER_POOL RECYCLE),
  PARTITION part_facturi_2
    VALUES LESS THAN (to_date('01/10/2015','dd/mm/yyyy'))
    TABLESPACE USERS STORAGE(BUFFER_POOL RECYCLE),
  PARTITION part_facturi_3
    VALUES LESS THAN (to_date('01/11/2015','dd/mm/yyyy'))
    TABLESPACE USERS STORAGE(BUFFER_POOL RECYCLE),
  PARTITION part_facturi_4
    VALUES LESS THAN (to_date('01/12/2015','dd/mm/yyyy'))
    TABLESPACE USERS STORAGE(BUFFER_POOL DEFAULT),
  PARTITION part_facturi_5
    VALUES LESS THAN (to_date('01/01/2016','dd/mm/yyyy'))
    TABLESPACE USERS STORAGE(BUFFER_POOL KEEP)
)
AS SELECT * FROM facturi;
```

Am folosit o partiţionare de tip interval după data facturii. Practic pentru fiecare din lunile august până în decembrie 2015 am alocat câte o partiţie.

Ce alte tipuri de partiţionare mai cunoaşteţi?

Cu trimitere la capitolul precedent despre păstrarea datelor în memorie, observaţi că zonele de memorie pot fi configurate la nivel de partiţie.

Pentru a fi siguri că optimizorul va decide corect asupra planurilor de execuţie pentru SQL-urile ce vizează tabela FACTURI_RDF_PART, e o idee bună să colectăm statistici:

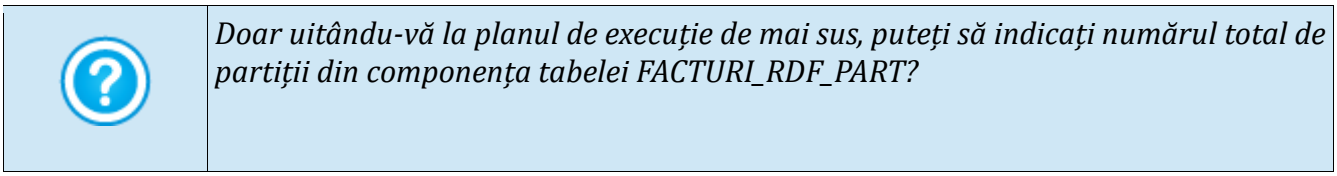
```
VANZARI@SQL> exec dbms_stats.gather_table_stats(ownname => user, tabname => 'facturi_rdf_part');

PL/SQL procedure successfully completed.
```

Să rulăm mai întâi un SELECT care accesează toate datele din tabela FACTURI_RDF_PART:

```
VANZARI@SQL> set linesize 120
VANZARI@SQL> set autotrace traceonly
VANZARI@SQL> select * from facturi_rdf_part where datafact between to_date('01/08/2015', 'dd/mm/yyyy') and
to_date('01/01/2016', 'dd/mm/yyyy');
```

```
24000 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 1944855953

-------------------------------------------------------------------------------------------------
| Id  | Operation           | Name             | Rows  | Bytes | Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |                  | 24000 |   398K| 00:00:01 |       |       |
|   1 |  PARTITION RANGE ALL|                  | 24000 |   398K| 00:00:01 |     1 |     5 |
|*  2 |   TABLE ACCESS FULL | FACTURI_RDF_PART | 24000 |   398K| 00:00:01 |     1 |     5 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("DATAFACT">=TO_DATE(' 2015-08-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND
              "DATAFACT"<=TO_DATE(' 2016-01-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))


Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       1677  consistent gets
          0  physical reads
          0  redo size
     619933  bytes sent via SQL*Net to client
      18140  bytes received via SQL*Net from client
       1601  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      24000  rows processed
```

De data aceasta am folosit opţiunea *AUTOTRACE* a sqlplus-ului. Sigur, puteam să mergem în continuare cu *DBMS_XPLAN*, dar e bine să ne amintim şi de celelalte metode de obţinere a planului de execuţie. Observăm calea de access aleasă de optimizor, şi anume "*PARTITION RANGE ALL*". Practic, toate partiţiile sunt accesate, ceea ce e normal ţinând cont de predicatul folosit. Observaţi şi numărul de citiri logice dat de indicatorul "consistent gets".



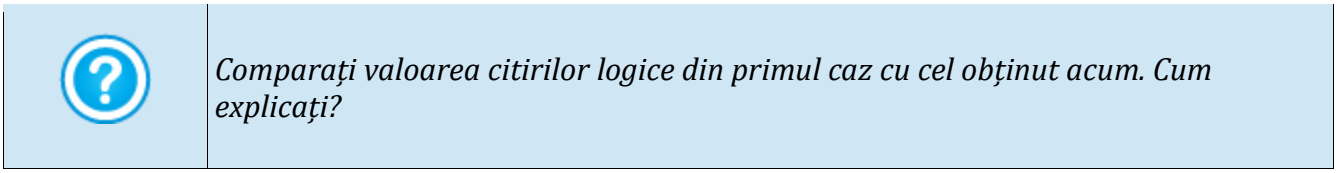
Doar uitându-vă la planul de execuţie de mai sus, puteţi să indicaţi numărul total de partiţii din componenţa tabelei FACTURI_RDF_PART?

Ce se întâmplă daca venim cu un predicat mai restrictiv. Sa zicem ca ne interesează doar facturile din august şi septembrie.

```
VANZARI@SQL> set linesize 120
VANZARI@SQL> set autotrace traceonly
VANZARI@SQL> select * from facturi_rdf_part where datafact between to_date('01/08/2015', 'dd/mm/yyyy') and
to_date('30/09/2015', 'dd/mm/yyyy');

2833 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 289346008
```

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                | Name             | Rows  | Bytes | Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |                  |  2833 | 48161 | 00:00:01 |       |       |
|   1 |  PARTITION RANGE ITERATOR|                  |  2833 | 48161 | 00:00:01 |     1 |     2 |
|*  2 |   TABLE ACCESS FULL      | FACTURI_RDF_PART |  2833 | 48161 | 00:00:01 |     1 |     2 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("DATAFACT"<=TO_DATE(' 2015-09-30 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND
              "DATAFACT">=TO_DATE(' 2015-08-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))


Statistics
----------------------------------------------------------
         43  recursive calls
          0  db block gets
        235  consistent gets
          1  physical reads
          0  redo size
      93090  bytes sent via SQL*Net to client
       2619  bytes received via SQL*Net from client
        190  SQL*Net roundtrips to/from client
          5  sorts (memory)
          0  sorts (disk)
       2833  rows processed
```

Observaţi noua cale de acces aleasă de optimizor: "*PARTITION RANGE ITERATOR*". Coloanele *Pstart* şi *Pstop* ne indică clar că doar prima şi a doua partiţie au fost accesate.



Comparaţi valoarea citirilor logice din primul caz cu cel obţinut acum. Cum explicaţi?

În sfârşit, ar fi interesant să vedem ce se întâmplă cu un predicat şi mai restrictiv: doar facturile din prima jumătate a lunii noiembrie.

```
VANZARI@SQL> set linesize 120
VANZARI@SQL> set autotrace traceonly
VANZARI@SQL> select * from facturi_rdf_part where datafact between to_date('01/11/2015', 'dd/mm/yyyy') and
to_date('15/11/2015', 'dd/mm/yyyy');

480 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 2488175927

-------------------------------------------------------------------------------------------------
| Id  | Operation              | Name             | Rows  | Bytes | Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                  |   480 |  8160 | 00:00:01 |       |       |
|   1 |  PARTITION RANGE SINGLE|                  |   480 |  8160 | 00:00:01 |     4 |     4 |
|*  2 |   TABLE ACCESS FULL    | FACTURI_RDF_PART |   480 |  8160 | 00:00:01 |     4 |     4 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
```

```
   2 - filter("DATAFACT"<=TO_DATE(' 2015-11-15 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))


Statistics
----------------------------------------------------------
         10  recursive calls
          0  db block gets
         63  consistent gets
          0  physical reads
          0  redo size
      17138  bytes sent via SQL*Net to client
        892  bytes received via SQL*Net from client
         33  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
        480  rows processed
```

Observaţi cum doar o singură partiţie a fost accesată.

Compresia datelor

Tabelele sau partiţiile care nu sunt actualizate frecvent cum ar fi de pildă tabelele de tip istoric sau cele de audit pot fi comprimate. Ele vor ocupa mai putin spaţiu pe disc, prim urmare vor fi necesare mai putine blocuri Oracle care să ţină respectivele date, prin urmare mai puţine citiri fizice/logice. Pe de altă parte, preţul plătit este în gradul de încărcare al procesorului, dat fiind că înregistrările vor trebui decompresate la citire.

Pentru a exemplifica, să facem o tabelă de backup pentru FACTURI.

```
VANZARI@SQL> create table facturi_bak compress as select * from facturi;

Table created.

VANZARI@SQL> set arraysize 300
VANZARI@SQL> set autotrace traceonly statistics
VANZARI@SQL> select * from facturi;

24000 rows selected.


Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        156  consistent gets
          0  physical reads
          0  redo size
     537891  bytes sent via SQL*Net to client
       1420  bytes received via SQL*Net from client
         81  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      24000  rows processed

SQL> select * from facturi_bak;

24000 rows selected.


Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        124  consistent gets
          0  physical reads
```

```
          0  redo size
     316458  bytes sent via SQL*Net to client
       1420  bytes received via SQL*Net from client
         81  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      24000  rows processed
```

Observaţi diferenţa la categoria "*consistent gets*".

Paralelism

O altă metoda de a optimiza interogările ce au predicate cu selectivitate slabă o reprezintă folosirea paralelismului. Practic, mai multe procese server lucrează simultan pentru a returna rezultatul interogării. În general folosim paralelismul pe sisteme de tip depozite de date sau "data-mining". Îl putem folosi şi în alte scenarii, dar trebuie să ţinem cont de faptul că o interogare executată cu funcţionalitatea de paralelism va "mânca" mult mai multe resurse pe server.

Iată cum ar arăta un plan de execuţie pentru o astfel de interogare:

```
VANZARI@SQL> set linesize 120
VANZARI@SQL> set autotrace traceonly
VANZARI@SQL> select /*+ parallel(t) cardinality(t 1e6) */ * from facturi t;

24000 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 2453027608

-------------------------------------------------------------------------------------------------------
| Id  | Operation            | Name     | Rows  | Bytes | Time     |    TQ  |IN-OUT| PQ Distrib |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |          |  1000K|    16M| 00:00:01 |        |      |         |
|   1 |  PX COORDINATOR      |          |       |       |          |        |      |         |
|   2 |   PX SEND QC (RANDOM)| :TQ10000 |  1000K|    16M| 00:00:01 |  Q1,00 | P->S | QC (RAND)  |
|   3 |    PX BLOCK ITERATOR |          |  1000K|    16M| 00:00:01 |  Q1,00 | PCWC |         |
|   4 |     TABLE ACCESS FULL| FACTURI  |  1000K|    16M| 00:00:01 |  Q1,00 | PCWP |         |
-------------------------------------------------------------------------------------------------------

Note
-----
   - Degree of Parallelism is 2 because of table property


Statistics
----------------------------------------------------------
          6  recursive calls
          0  db block gets
        153  consistent gets
          0  physical reads
          0  redo size
     408342  bytes sent via SQL*Net to client
       1420  bytes received via SQL*Net from client
         81  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      24000  rows processed
```

Putem controla gradul de paralelism la nivelul segmentului, folosind clauza *PARALLEL* sau prin intermediul unui "hint", la nivelul interogării.

Folosirea unui index pentru a ocoli accesul la tabelă

Poate părea ciudată folosirea unui index pentru a optimiza interogări cu predicate ce au selectivitate slabă. Totuşi, un index ce conţine doar o parte din coloanele din tabela pe care e definit va ocupa mai puţin spaţiu pe disc, iar daca interogarea are în lista de coloane solicitate pe cele conţinute de index, Oracle ar trebui să fie suficient de deştept să acceseze doar structura indexului. Prin urmare, mai puţine citiri logice, posibil un plan de execuţie mai eficient.

Să exemplificăm:

```
VANZARI@SQL> set arraysize 300
VANZARI@SQL> set autotrace traceonly
VANZARI@SQL> select nrfact, codpr, cantitate from liniifact;

168520 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 770896586

--------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |   168K|  1974K| 00:00:01 |
|   1 |  TABLE ACCESS FULL| LINIIFACT |   168K|  1974K| 00:00:01 |
--------------------------------------------------------------------------


Statistics
----------------------------------------------------------
          7  recursive calls
          0  db block gets
       1139  consistent gets
        565  physical reads
          0  redo size
    2255996  bytes sent via SQL*Net to client
       6722  bytes received via SQL*Net from client
        563  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
     168520  rows processed

VANZARI@SQL> create index ix_linii_fact_test on liniifact(nrfact, codpr, cantitate);

Index created.

VANZARI@SQL> select nrfact, codpr, cantitate from liniifact;

168520 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 2840895513

-------------------------------------------------------------------------------------
| Id  | Operation            | Name               | Rows  | Bytes | Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                    |   168K|  1974K| 00:00:01 |
|   1 |  INDEX FAST FULL SCAN| IX_LINII_FACT_TEST |   168K|  1974K| 00:00:01 |
-------------------------------------------------------------------------------------


Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
```

```
      1102  consistent gets
       535  physical reads
         0  redo size
   2255996  bytes sent via SQL*Net to client
      6722  bytes received via SQL*Net from client
       563  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
    168520  rows processed
```

În cazul de faţă nu e o scădere drastică a numărului de "consistent gets", dar pe volume mai mari de date e posibil să se ajungă la o diferenţă semnificativă.

Optimizarea interogărilor cu selectivitate bună

Un predicat cu selectivitate buna va selecta doar o mică fracţiune din totalul de înregistrări dintr-o tabelă. O interogare cu un predicat de egalitate după coloanele din cheia primară sau dintr-o cheie unică este un predicat cu selectivitate buna. Va selecta zero sau maximum o înregistrare.

Accesul la înregistrări după ROWID

Ştim deja că fiecare înregistrare poate fi localizată după o adresă denumită ROWID. Acest ROWID oferă cel mai rapid acces spre înregistrarea corespunzătoare. Dacă ştim această adresă atunci nu mai avem nevoie de nici un index.

Iată, înregistrarea aferentă clientului 1997 are următorul ROWID:

```
VANZARI@SQL> select rowid from clienti where dencl = 'Client 1997';

ROWID
------------------
AAAWpBAAGAAAAUkABC
```

Având acum acest ROWID pot ajunge oricând, foarte rapid, fix la înregistrarea corespunzătoare clientului 1997:

```
VANZARI@SQL> set linesize 120
VANZARI@SQL> set autotrace on
VANZARI@SQL> select * from clienti where rowid = 'AAAWpBAAGAAAAUkABC';

     CODCL DENCL                          CODFISCAL ADRESA                          CODPOS TELEFON
---------- ------------------------------ --------- ------------------------------ ------ ----------
      1997 Client 1997                    R1997                                    11001


Execution Plan
----------------------------------------------------------
Plan hash value: 764057726

-------------------------------------------------------------------------------------------
| Id  | Operation                   | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |         |     1 |    37 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY USER ROWID| CLIENTI |     1 |    37 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------


Statistics
----------------------------------------------------------
          0  recursive calls
```

```
          0  db block gets
          1  consistent gets
          0  physical reads
          0  redo size
        915  bytes sent via SQL*Net to client
        551  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Vom obține un singur "consistent get" doar dacă nu avem "recursive calls", deoarece interogările pe dicționar fac și ele la rândul lor citiri logice. Pentru a obține rezultatul de mai sus ar trebui să executați respectivul SELECT de mai multe ori.

Să zicem că proiectați o aplicație de salarizare, iar unul din module se referă la nomenclatorul de salariați. Acesta vă permite să vizualizați salariații unul câte unul (card view) și să modificați câmpurile corespunzătoare fiecărui angajat. Cum ați putea integra conceptul de ROWID în proiectarea acestui modul, astfel încat actualizarea câmpurilor unui angajat să fie foarte eficientă?

Accesul prin index

Este cea mai frecventă metodă de optimizare a interogărilor cu selectivitate bună. De altfel, fiecare constrângere de cheie primară sau de unicitate are în spate un index creat automat de către Oracle. Indecșii sunt practic structuri redundante care conțin o cheie de indexare ce corespund unei coloane sau mai multor coloane din tabela pe care indexul este creat, iar pentru fiecare cheie, în structura indexului, este stocat ROWID-ul spre înregistrarea referită. În funcție de tipul de index, structura acestuia este diferită. Ce e important de reținut este faptul că Oracle implementează algoritmi eficienti de parcurgere a acestor indecși, astfel încât să poată face localizarea rapidă a înregistrărilor selectate printr-un predicat cu selectivitate bună.

Pe lângă aspectele ce țin de optimizare, indecșii pot fi folosiți și pentru a indexa cheile străine din tabelele copil. Acest lucru e recomandat deoarece un LOCK pe înregistrarea parinte va bloca în cascadă doar liniile corespunzătoare din tabela copil atunci când există un index definit pe cheia străină.

Cel mai comun tip de index este cel de tip BTree. Să zicem că vrem să selectăm clienții și după denumire, ca mai jos:

```
VANZARI@SQL> set autotrace on
VANZARI@SQL> select * from clienti where dencl = 'Client 1997';

     CODCL DENCL                          CODFISCAL ADRESA                          CODPOS TELEFON
---------- ------------------------------ --------- ------------------------------ ------ ----------
      1997 Client 1997                    R1997                                    11001
```

```
Execution Plan
----------------------------------------------------------
Plan hash value: 1121005057

--------------------------------------------------------------------------------
| Id  | Operation         | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |         |     1 |    37 |     5   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| CLIENTI |     1 |    37 |     5   (0)| 00:00:01 |
--------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("DENCL"='Client 1997')


Statistics
----------------------------------------------------------
         93  recursive calls
          0  db block gets
        144  consistent gets
          0  physical reads
          0  redo size
        915  bytes sent via SQL*Net to client
        551  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          6  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Observăm că avem o rata de citiri logice per linie returnată foarte mare. E şi normal dat fiind că pentru o singură înregistrare a trebuit să se facă o parcurgere totală a tabelei. Ar merge în acest caz să creăm un index b-tree. Folosim comanda de mai jos:

```
VANZARI@SQL> create index ix_clienti_dencl on clienti(dencl);

Index created.
```

Dacă rulăm aceeaşi interogare, dar cu indexul tocmai creat, obţinem:

```
VANZARI@SQL> set autotrace on
VANZARI@SQL> select * from clienti where dencl = 'Client 1997';

     CODCL DENCL                          CODFISCAL ADRESA                       CODPOS TELEFON
---------- ------------------------------ --------- ---------------------------- ------ ----------
      1997 Client 1997                    R1997                                  11001


Execution Plan
----------------------------------------------------------
Plan hash value: 3395442693

-----------------------------------------------------------------------------------------------
| Id  | Operation                           | Name             | Rows  | Bytes | Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                  |     1 |    37 | 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| CLIENTI          |     1 |    37 | 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | IX_CLIENTI_DENCL |     1 |       | 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("DENCL"='Client 1997')
```

```
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads
          0  redo size
        918  bytes sent via SQL*Net to client
        551  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Acest index se comportă bine şi cu predicate LIKE, atât timp cât furnizăm în masca de potrivire un şablon cu prefix:

```
VANZARI@SQL> set autotrace on explain
VANZARI@SQL> select * from clienti where dencl like 'Client 199%';

     CODCL DENCL                          CODFISCAL ADRESA   CODPOS TELEFON
---------- ------------------------------ --------- -------- ------ ----------
      1990 Client 1990                    R1990              11001
      1991 Client 1991                    R1991              11001
      1992 Client 1992                    R1992              11001
      1993 Client 1993                    R1993              11001
      1994 Client 1994                    R1994              11001
      1995 Client 1995                    R1995              11001
      1996 Client 1996                    R1996              11001
      1997 Client 1997                    R1997              11001
      1998 Client 1998                    R1998              11001
      1999 Client 1999                    R1999              11001

10 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 3395442693

-----------------------------------------------------------------------------------------------
| Id  | Operation                           | Name             | Rows  | Bytes | Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                  |     1 |    37 | 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| CLIENTI          |     1 |    37 | 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | IX_CLIENTI_DENCL |     1 |       | 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("DENCL" LIKE 'Client 199%')
       filter("DENCL" LIKE 'Client 199%')
```

Totuşi, o mască LIKE neprefixată nu funcţionează cum ne-am aştepta:

```
VANZARI@SQL> set autotrace on explain
VANZARI@SQL> select * from clienti where dencl like '%199%';

     CODCL DENCL                          CODFISCAL ADRESA  CODPOS TELEFON
---------- ------------------------------ --------- ------- ------ ----------
      1199 Client 1199                    R1199             10401
      1990 Client 1990                    R1990             11001
      1991 Client 1991                    R1991             11001
      1992 Client 1992                    R1992             11001
      1993 Client 1993                    R1993             11001
```

```
      1994 Client 1994                  R1994             11001
      1995 Client 1995                  R1995             11001
      1996 Client 1996                  R1996             11001
      1997 Client 1997                  R1997             11001
      1998 Client 1998                  R1998             11001
      1999 Client 1999                  R1999             11001

11 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 1121005057

--------------------------------------------------------------------
| Id  | Operation         | Name    | Rows  | Bytes | Time     |
--------------------------------------------------------------------
|   0 | SELECT STATEMENT  |         |    11 |   407 | 00:00:01 |
|*  1 |  TABLE ACCESS FULL| CLIENTI |    11 |   407 | 00:00:01 |
--------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("DENCL" LIKE '%199%' AND "DENCL" IS NOT NULL)
```

Optimizorul, în acest caz, a ales varianta "*TABLE ACCESS FULL*".

Atenţie sporită trebuie acordată şi funcţiilor aplicate pe coloana indexată. Dacă dorim sa selectăm clientul, dar fără să ne batem capul cu literele mari sau mici din denumire, putem imagina ceva de genul:

```
VANZARI@SQL> set autotrace on explain
VANZARI@SQL> select * from clienti where upper(dencl) = 'CLIENT 1997';

     CODCL DENCL                         CODFISCAL ADRESA   CODPOS TELEFON
---------- ------------------------------ --------- -------- ------ ----------
      1997 Client 1997                    R1997             11001


Execution Plan
----------------------------------------------------------
Plan hash value: 1121005057

--------------------------------------------------------------------
| Id  | Operation         | Name    | Rows  | Bytes | Time     |
--------------------------------------------------------------------
|   0 | SELECT STATEMENT  |         |    10 |   370 | 00:00:01 |
|*  1 |  TABLE ACCESS FULL| CLIENTI |    10 |   370 | 00:00:01 |
--------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(UPPER("DENCL")='CLIENT 1997')
```

Se poate observa că, în acest caz, funcţia *UPPER* aplicată pe cloana "*dencl*" a invalidat folosirea indexului. Putem rezolva problema relativ simplu prin adăugarea unui index funcţional:

```
VANZARI@SQL> set autotrace on explain
VANZARI@SQL> create index fix_clienti_upper_dencl on clienti(upper(dencl));

Index created.

SQL> select * from clienti where upper(dencl) = 'CLIENT 1997';
```

```
     CODCL DENCL                          CODFISCAL ADRESA  CODPOS TELEFON
---------- ------------------------------ --------- ------- ------ ----------
      1997 Client 1997                    R1997             11001


Execution Plan
----------------------------------------------------------
Plan hash value: 309075056

-------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name                    | Rows  | Bytes | Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                         |    10 |   370 | 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| CLIENTI                 |    10 |   370 | 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | FIX_CLIENTI_UPPER_DENCL |     4 |       | 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access(UPPER("DENCL")='CLIENT 1997')
```

Atenție sporită trebuie acordată și valorilor nule atunci când folosim indecși de tip b-tree. Asta pentru că un astfel de index nu stochează valorile nule. Spre exemplificare, să spunem că avem un client fără denumire. Vom simula acest lucru prin următorul UPDATE:

```
VANZARI@SQL> update clienti set dencl = null where codcl = 1997;

1 row updated.

VANZARI@SQL> set autotrace on explain
VANZARI@SQL> select * from clienti where dencl is null;

     CODCL DENCL                          CODFISCAL ADRESA    CODPOS TELEFON
---------- ------------------------------ --------- --------- ------ ----------
      1997                                R1997               11001


Execution Plan
----------------------------------------------------------
Plan hash value: 1121005057

--------------------------------------------------------------------
| Id  | Operation         | Name    | Rows  | Bytes | Time     |
--------------------------------------------------------------------
|   0 | SELECT STATEMENT  |         |     1 |    37 | 00:00:01 |
|*  1 |  TABLE ACCESS FULL| CLIENTI |     1 |    37 | 00:00:01 |
--------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("DENCL" IS NULL)
```

Se poate observa că deși avem index pe coloana "dencl", acesta nu este folosit pentru a detecta valorile nule. Prin urmare ajungem să facem un "full table scan". Am putea rezolva acest neajuns printr-un index compus de tip b-tree sau printr-un index bitmap. Mai întâi varianta indexului compus:

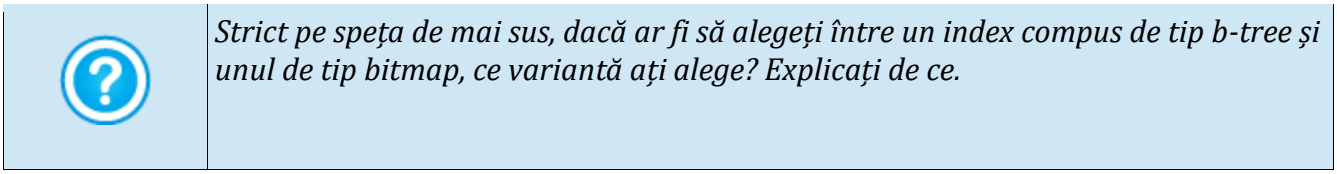
```
VANZARI@SQL> create index ix_clienti_compus on clienti(codcl, dencl);

Index created.

VANZARI@SQL> set autotrace on explain
```

```
VANZARI@SQL> select * from clienti where dencl is null;

     CODCL DENCL                          CODFISCAL ADRESA   CODPOS TELEFON
---------- ------------------------------ --------- -------- ------ ----------
      1997                                R1997              11001


Execution Plan
----------------------------------------------------------
Plan hash value: 1677021668

-------------------------------------------------------------------------------------
| Id  | Operation                           | Name              | Rows  | Bytes |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                   |     1 |    37 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| CLIENTI           |     1 |    37 |
|*  2 |   INDEX SKIP SCAN                   | IX_CLIENTI_COMPUS |     1 |       |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("DENCL" IS NULL)
       filter("DENCL" IS NULL)
```

Stergem indecşii de pe coloana "*dencl*" şi încercăm cu un index de tip bitmap:

```
VANZARI@SQL> drop index ix_clienti_dencl;

Index dropped.

VANZARI@SQL> create bitmap index bix_clienti_dencl on clienti(dencl);

Index created.

VANZARI@SQL> set autotrace on explain
VANZARI@SQL> select * from clienti where dencl is null;

     CODCL DENCL                          CODFISCAL ADRESA  CODPOS TELEFON
---------- ------------------------------ --------- ------- ------ ----------
      1997                                R1997             11001


Execution Plan
----------------------------------------------------------
Plan hash value: 1762749066

-------------------------------------------------------------------------------
| Id  | Operation                           | Name              | Rows  |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                   |     1 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| CLIENTI           |     1 |
|   2 |   BITMAP CONVERSION TO ROWIDS       |                   |       |
|*  3 |    BITMAP INDEX SINGLE VALUE        | BIX_CLIENTI_DENCL |       |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("DENCL" IS NULL)
```

Se poate observa că index-ul de tip bitmap este folosit, asta deoarece un index bitmap stochează în structura sa şi valorile nule.



Strict pe speța de mai sus, dacă ar fi să alegeți între un index compus de tip b-tree și unul de tip bitmap, ce variantă ați alege? Explicați de ce.

Tabele IOT

Tabelele IOT (Index Organized Table) sunt un fel de struțo-cămile. O combinație de index și tabelă. Un IOT are structura de index, doar că, în dreptul cheii de indexare, în locul ROWID-ului sunt stocate efectiv celelalte coloane. Avantajul e că nu mai este necesar încă un drum la tabelă prin ROWID, dat fiind că toate datele sunt deja acolo.

Să exemplificăm pe modelul tabelei CLIENTI. Comanda de creare a acestui IOT este:

```
CREATE TABLE clienti_iot (
    codcl NUMBER(6) CONSTRAINT pk_clienti_iot PRIMARY KEY,
    dencl VARCHAR2(30),
    codfiscal CHAR(9),
    adresa VARCHAR2(40),
    codpost CHAR(6),
    telefon VARCHAR2(10)
) organization index;
```

O tabelă de tip IOT este obligatoriu să aibă o cheie primară. Apoi populăm folosind datele din tabela CLIENTI:

```
VANZARI@SQL> insert into clienti_iot select * from clienti;

1000 rows created.
```

Să facem o interogare acum după cheia primară:

```
VANZARI@SQL> set autotrace on explain
VANZARI@SQL> select * from clienti_iot where codcl = 2000;

...


Execution Plan
----------------------------------------------------------
Plan hash value: 2173914887

--------------------------------------------------------------------------------
| Id  | Operation         | Name           | Rows  | Bytes | Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                |     1 |    78 | 00:00:01 |
|*  1 |  INDEX UNIQUE SCAN| PK_CLIENTI_IOT |     1 |    78 | 00:00:01 |
--------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("CODCL"=1997)

SQL> select * from clienti_iot where dencl = 'Client 2000';

...


Execution Plan
```

```
------------------------------------------------------------
Plan hash value: 2969006588

-------------------------------------------------------------------------------
| Id  | Operation            | Name           | Rows  | Bytes | Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                |     1 |    78 | 00:00:01 |
|*  1 |  INDEX FAST FULL SCAN| PK_CLIENTI_IOT |     1 |    78 | 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("DENCL"='Client 2000')
```

Se poate observa că accesul se face doar în contextul index-ului din spatele IOT-ului, asta chiar şi pentru acele predicate care nu fac neaparat referire la cheia primară.

Ce dezavantaje vedeţi pentru folosirea IOT-urilor?

Tabele clusterizate de tip "Single HASH"

Tabelele clusterizate de tip hash sunt ceva mai exotice şi nu sunt foarte des utilizate, desi ele pot aduce îmbunătăţiri seminificative ale performanţei pentru anumite interogări, mai ales pentru aşa-numitele "lookup tables". În principiu, dacă avem o cheie comună şi ştim dinainte cam câte chei distincte vom avea, putem crea un astfel de cluster în care vom stoca înregistrările care partajează aceeaşi cheie, grupat, în aceleaşi blocuri. Spre exemplu, s-ar putea să avem cazul în care clienţii sunt selectaţi din tabela CLIENTI după codul postal. Putem şti dinainte câte coduri poştale distincte vom avea? Da. Conform nomenclatorului de coduri postale. Din raţiuni didactice, în tabela noastră de coduri poştale sunt maximum 10 coduri postale distincte. Am putea crea lejer un cluster după următorul model:

```
create cluster coduri_postale_hash (
  codpost char(6)
)
hashkeys 10
size 8192;

create table clienti_cluster (
  codcl number(6),
  dencl varchar2(30),
  codfiscal char(9),
  adresa varchar2(40),
  codpost char(6),
  telefon varchar2(10)
) cluster coduri_postale_hash(codpost);

insert into clienti_cluster select * from clienti;
```

Să vedem ce se întâmplă atunci când interogăm acest cluster după cheia de clusterizare:

```
VANZARI@SQL> set autotrace on explain
VANZARI@SQL> select * from clienti_cluster where codpost = '10101';
```

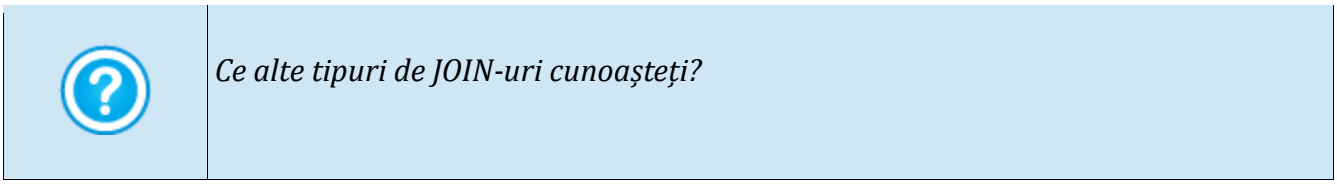
```
     CODCL DENCL                          CODFISCAL ADRESA       CODPOS TELEFON
---------- ------------------------------ --------- ------------ ------ ----------
      1001 Client 1001                    R1001                  10101
      1002 Client 1002                    R1002                  10101
      1003 Client 1003                    R1003                  10101
      1004 Client 1004                    R1004                  10101
      1005 Client 1005                    R1005                  10101
      1006 Client 1006                    R1006                  10101
      1007 Client 1007                    R1007                  10101
      1008 Client 1008                    R1008                  10101
      1009 Client 1009                    R1009                  10101
      1010 Client 1010                    R1010                  10101
      1011 Client 1011                    R1011                  10101

     CODCL DENCL                          CODFISCAL ADRESA       CODPOS TELEFON
---------- ------------------------------ --------- ------------ ------ ----------
      1012 Client 1012                    R1012                  10101
      1013 Client 1013                    R1013                  10101

13 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 1797436792

-------------------------------------------------------------------------------
| Id  | Operation         | Name            | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                 |    13 |  1014 |     0   (0)|
|*  1 |  TABLE ACCESS HASH| CLIENTI_CLUSTER |    13 |  1014 |            |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("CODPOST"='10101')

Note
-----
   - dynamic statistics used: dynamic sampling (level=2)


Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          2  consistent gets
          0  physical reads
          0  redo size
       1342  bytes sent via SQL*Net to client
        551  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
         13  rows processed
```

Observaţi noua cale de acces din planul de execuţie, dar şi numărul mic de blocuri logice citite. Şi asta fără să avem vreun index creat pe această structură.

- *Rulaţi acelaşi SELECT dar pe tabela neclusterizată clienţi şi aflaţi numărul de blocuri citite (consistent gets). Cum explicaţi diferenţa?*
- *Ce se întâmplă dacă înlocuiţi predicatul de mai sus cu o condiţie de between: WHERE codpost between '10000' and '20000'?*

Optimizarea JOIN-urilor

Cu excepţia *CROSS JOIN*-ului, celelalte tipuri de joncţiuni necesită de obicei o condiţie după care să se efectueze JOIN-ul. În general, o strategie de indexare a coloanelor după care se face joncţionarea ar putea îmbunătăţi performanţa / eficienţa interogării.



Ce alte tipuri de JOIN-uri cunoaşteţi?

Din punctul de vedere al optimizorului, cele mai cunoscute metode de joncţionare sunt: *NESTED LOOP*, *HASH JOIN* şi *SORT MERGE*. Le vom exemplifca pe fiecare în parte, dar vom "forţa" optimizorul să aleagă aceste metode de JOIN prin intermediul unor hint-uri.

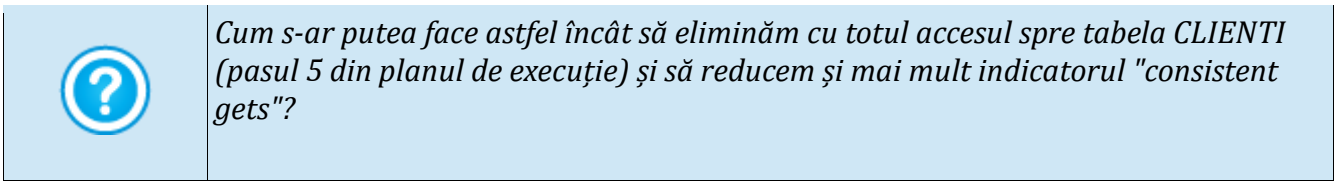
Iată un prim exemplu care foloseşte metoda NESTED LOOP. Vom joncţiona tabela CLIENTI cu cea în care sunt stocate codurile poştale.

```
VANZARI@SQL> set lines 120
VANZARI@SQL> set autotrace traceonly
VANZARI@SQL> SELECT /*+ use_nl(c cp) gather_plan_statistics*/ c.codcl, c.dencl, cp.loc FROM coduri_postale cp
INNER JOIN clienti c ON CP.CodPost = C.CodPost;

1000 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 3068962852

-------------------------------------------------------------------
| Id  | Operation          | Name           | Rows  | Time     |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                |  1000 | 00:00:01 |
|   1 |  NESTED LOOPS      |                |  1000 | 00:00:01 |
|   2 |   TABLE ACCESS FULL| CODURI_POSTALE |    10 | 00:00:01 |
|*  3 |   TABLE ACCESS FULL| CLIENTI        |   100 | 00:00:01 |
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("CP"."CODPOST"="C"."CODPOST")


Statistics
----------------------------------------------------------
         58  recursive calls
          0  db block gets
        291  consistent gets
          0  physical reads
          0  redo size
      34437  bytes sent via SQL*Net to client
       1277  bytes received via SQL*Net from client
         68  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
       1000  rows processed
```

Conform planului de mai sus, pentru fiecare cod postal citit din *CODURI_POSTALE*, tabela *CLIENTI* este parcursă de sus până jos. Deoarece accesul în tabela *CLIENTI* se face după coloana *CODPOST*, un index pe această coloană ar fi de folos:

```
VANZARI@SQL> set autotrace traceonly
VANZARI@SQL> create index ix_clienti_codpost on clienti(codpost);

Index created.

VANZARI@SQL> SELECT /*+ use_nl(c cp) gather_plan_statistics*/ c.codcl, c.dencl, cp.loc FROM coduri_postale cp
INNER JOIN clienti c ON CP.CodPost = C.CodPost;

1000 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 2049414977

-------------------------------------------------------------------------
| Id  | Operation                    | Name               | Rows  |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                    |  1000 |
|   1 |  NESTED LOOPS                |                    |  1000 |
|   2 |   NESTED LOOPS               |                    |  1000 |
|   3 |    TABLE ACCESS FULL         | CODURI_POSTALE     |    10 |
|*  4 |    INDEX RANGE SCAN          | IX_CLIENTI_CODPOST |   100 |
|   5 |   TABLE ACCESS BY INDEX ROWID| CLIENTI            |   100 |
-------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("CP"."CODPOST"="C"."CODPOST")


Statistics
----------------------------------------------------------
         39  recursive calls
          0  db block gets
        204  consistent gets
          3  physical reads
          0  redo size
      34437  bytes sent via SQL*Net to client
       1277  bytes received via SQL*Net from client
         68  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
       1000  rows processed
```

Observaţi faptul că noul index este folosit de optimizor şi că indicatorul "consistent gets" a scăzut.



Cum s-ar putea face astfel încât să eliminăm cu totul accesul spre tabela CLIENTI (pasul 5 din planul de execuţie) şi să reducem şi mai mult indicatorul "consistent gets"?

Un exemplu de HASH_JOIN ar putea fi cel de mai jos, în care joncţionăm patru tabele:

```
VANZARI@SQL> set lines 120
```

```
VANZARI@SQL> set autotrace traceonly
VANZARI@SQL> SELECT /*+ gather_plan_statistics use_hash(c f l p) */ c.dencl, f.nrfact, f.datafact, p.denpr,
l.cantitate FROM clienti c INNER JOIN facturi f on c.codcl = f.codcl INNER JOIN liniifact l on f.nrfact =
l.nrfact INNER JOIN produse p on l.codpr = p.codpr;

168520 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 310653162

-------------------------------------------------------------------
| Id  | Operation               | Name               | Rows  |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT        |                    |   167K|
|*  1 |  HASH JOIN              |                    |   167K|
|   2 |   TABLE ACCESS FULL     | PRODUSE            |  1000 |
|*  3 |   HASH JOIN             |                    |   167K|
|   4 |    TABLE ACCESS FULL    | CLIENTI            |  1000 |
|*  5 |    HASH JOIN            |                    |   167K|
|*  6 |     TABLE ACCESS FULL   | FACTURI            | 24000 |
|*  7 |     INDEX FAST FULL SCAN| IX_LINII_FACT_TEST |   168K|
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("L"."CODPR"="P"."CODPR")
   3 - access("C"."CODCL"="F"."CODCL")
   5 - access("F"."NRFACT"="L"."NRFACT")
   6 - filter("F"."CODCL">1000)
   7 - filter("L"."CODPR">0)


Statistics
----------------------------------------------------------
        238  recursive calls
          0  db block gets
      12198  consistent gets
        619  physical reads
          0  redo size
    6820073  bytes sent via SQL*Net to client
     124125  bytes received via SQL*Net from client
      11236  SQL*Net roundtrips to/from client
         23  sorts (memory)
          0  sorts (disk)
     168520  rows processed
```

În mod asemănător, dar de data acesta cu o clauza ORDER BY, putem merge pe varianta SORT MERGE JOIN:

```
VANZARI@SQL> set lines 120
VANZARI@SQL> set autotrace traceonly
VANZARI@SQL> SELECT /*+ gather_plan_statistics use_merge(c f l p) */ c.dencl, f.nrfact, f.datafact, p.denpr,
l.cantitate FROM clienti c INNER JOIN facturi f on c.codcl = f.codcl INNER JOIN liniifact l on f.nrfact = l.nrfact
INNER JOIN produse p on l.codpr = p.codpr ORDER BY 1, 2, 3;

168520 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 800012184

-------------------------------------------------------------------
| Id  | Operation               | Name               | Rows  |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT        |                    |   167K|
```

```
|   1 |  SORT ORDER BY            |                    |   167K|
|   2 |   MERGE JOIN              |                    |   167K|
|   3 |    SORT JOIN              |                    |   167K|
|   4 |     MERGE JOIN            |                    |   167K|
|   5 |      SORT JOIN            |                    | 24000 |
|   6 |       MERGE JOIN          |                    | 24000 |
|   7 |        SORT JOIN          |                    |  1000 |
|   8 |         TABLE ACCESS FULL | CLIENTI            |  1000 |
|*  9 |        SORT JOIN          |                    | 24000 |
|* 10 |         TABLE ACCESS FULL | FACTURI            | 24000 |
|* 11 |      SORT JOIN            |                    |   168K|
|* 12 |       INDEX FAST FULL SCAN| IX_LINII_FACT_TEST |   168K|
|* 13 |    SORT JOIN              |                    |  1000 |
|  14 |     TABLE ACCESS FULL     | PRODUSE            |  1000 |
---------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   9 - access("C"."CODCL"="F"."CODCL")
       filter("C"."CODCL"="F"."CODCL")
  10 - filter("F"."CODCL">1000)
  11 - access("F"."NRFACT"="L"."NRFACT")
       filter("F"."NRFACT"="L"."NRFACT")
  12 - filter("L"."CODPR">0)
  13 - access("L"."CODPR"="P"."CODPR")
       filter("L"."CODPR"="P"."CODPR")


Statistics
----------------------------------------------------------
        244  recursive calls
          0  db block gets
        957  consistent gets
          0  physical reads
          0  redo size
    6820103  bytes sent via SQL*Net to client
     124125  bytes received via SQL*Net from client
      11236  SQL*Net roundtrips to/from client
         29  sorts (memory)
          0  sorts (disk)
     168520  rows processed
```

Cum s-ar putea optimiza SELECT-urile de mai sus? Ce indecşi ar trebui creaţi?