Luis Marques

ORA-PT

07
JAN 12

10053 Parser - Parke

POR LUÍS MARQUES, ÀS 11:47 | COMENTAR

As promised, i will share the code of Parke (new shiny name), the 10053 trace files Parser for Oracle. It is on github here: https://github.com/lcmarques/Parke

For now, it has only 2 "less than useful" features: hints and explain plans. I will probably improve it next months to make it very complex matching Oracle software.

tags: 10053parser, parke

Email

27
DEZ 11

10053 Parser

POR LUÍS MARQUES, ÀS 17:05 | COMENTAR

Last time i wrote about SYS_DL_CURSOR hint to find out if i can make use of it, but i realized that i rely many times on trace files, mainly 10053 tracefile, so i decided to write a simple parser (in Python) to help me. It's very simple and for now i will not share source code with you until i have a "good" and readeable version of the code :-)

10053Parser has 2 features for now:
1 - Hints
2 - Explain plans

Feature #1 - Hints

10053Parser will (eventually) parse contents to find out how many DML/DDL statements were executed and which hints were used on statements showing the output in a good/fashion way:

```
[oracle@localhost trace]$ ./trace_10053.py --hints testSID_ora_14131_MESSI.trc
Report Hints for [testSID_ora_14131_MESSI.trc] ...
Hint             | Used | Error | Level | SQL FULL TEXT
-------------------------------------------------------------------------------

N/A              | N/A  | N/A   | N/A   | create table t1_abc as select 1 as N1 from du
SYS_DL_CURSOR () | 0    | 0     | 1     | select /*+ full(a) SYS_DL_CURSOR */ N1 from t
FULL ("A")       | 1    | 0     | 3     | select /*+ full(a) SYS_DL_CURSOR */ N1 from t
INDEX ("A")      | 1    | 0     | 3     | select /*+ index(a) */ N1 from t1_abc a
```

As you can see, output is easy to read and you have all information regarding hints, even if the statements uses multiple hints.

Feature #2 - Explain Plans

Instead of trying to remember explain plans for every statement, 10053Parser allows you to output the list of explain plans and corresponding statement:

AUTHOR

Luis Marques

POSTS RECENTES

JANEIRO 2012

DOM	SEG	TER	QUA	QUI	SEX	SAB
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

« Dez

PESQUISAR BLOG

OK

SUBSCREVER FEEDS

Posts
Comentários

SUPPORT ME

Flattr this!

TAGS

10053parser background book btr buffer cache cbo clarifyme concurrency ctas datatype datatype datedirect pathflashback fts hwm index io locks number oracle oracle encryptionparkeperformance oracle encryptionsegmentssintaxe todas as tags

TWITTER

```
[oracle@localhost trace]$ ./trace_10053.py --explain testSID_ora_14131_MESSI.trc
Report Explain for [testSID_ora_14131_MESSI.trc] ...
SQL: create table t1_abc as select 1 as N1 from dual

-----------------------------------------+-----------------------------------+
| Id  | Operation               | Name    | Rows  | Bytes | Cost  | Time      |
-----------------------------------------+-----------------------------------+
| 0   | CREATE TABLE STATEMENT  |         |       |       |     3 |           |
| 1   |  LOAD AS SELECT         |         |       |       |       |           |
| 2   |   FAST DUAL             |         |     1 |       |     2 |  00:00:01 |
-----------------------------------------+-----------------------------------+

SQL: select /*+ full(a) SYS_DL_CURSOR */ N1 from t1_abc a

-------------------------------------+-----------------------------------+
| Id  | Operation          | Name    | Rows  | Bytes | Cost  | Time      |
-------------------------------------+-----------------------------------+
| 0   | SELECT STATEMENT   |         |       |       |     2 |           |
| 1   |  TABLE ACCESS FULL | T1_ABC  |     1 |    13 |     2 |  00:00:01 |
-------------------------------------+-----------------------------------+

SQL: select /*+ index(a) */ N1 from t1_abc a

-------------------------------------+-----------------------------------+
| Id  | Operation          | Name    | Rows  | Bytes | Cost  | Time      |
-------------------------------------+-----------------------------------+
| 0   | SELECT STATEMENT   |         |       |       |     2 |           |
| 1   |  TABLE ACCESS FULL | T1_ABC  |     1 |    13 |     2 |  00:00:01 |
-------------------------------------+-----------------------------------+
```

As soon i have the courage i will release source code :)

tags: 10053parser

Email

17
DEZ 11

SYS_DL_CURSOR - I can't make it

POR LUÍS MARQUES, ÀS 14:20 | COMENTAR

[English]
Some days ago i came across with a pl/sql code where they used (for some reason) an Oracle hint called SYS_DL_CURSOR. I got it when i was looking into V$SQL and i was not familiar with this, maybe the only thing i ("think") know about it is that in some conditions it performs a direct path insert, just like append hint. Hint itself seems to be not official documented by Oracle and i figured out that some aplications (Hi Informatica PowerCenter) uses it for some direct path insert ETL.
This is not by any means a "standard" way to do direct path inserts but if it works good, we need to get into it. So i decided to take a more deep look inside this hint to make sure that i understand what it really does.

A good way to verify if your direct path insert got right is try to query the table segment before your COMMIT or ROLLBACK operation. Let's create a table and try a simple direct path insert:

```
SQL> create table t1_dpi(n1 number, n2 number, CONSTRAINT pk_n UNIQUE(n1));


SQL> insert /*+ append_values */ into t1_dpi VALUES(1,1);
SQL> insert /*+ append_values */ into t1_dpi VALUES(2,2);


SQL> select * from t1_dpi where rownum < 1;
select * from t1_dpi where rownum < 1
              *
ERROR at line 1:
ORA-12838: cannot read/modify an object after modifying it in parallel
```

As you can easily see, we got an ORA-12838 after querying segment when doing a direct path. This happens because the transaction made an attempt to read (or modify) statements on a table and this is not allowed in direct loads. It will prevent data inconsistency [see this].
In this way you ensure that you will use an direct path insert using the common hint /*+ append_values */ (or /*+ append */) however you can use some 10046 trace:

```
SQL> alter session set events='10046 trace name context forever, level 12';
Session altered.


insert /*+ append_values */ into lcmarques.t1_dpi VALUES(2,2)
END OF STMT
PARSE #3:c=1000,e=926,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=1,plh=3581094869,tim=13241253863
WAIT #3: nam='direct path write' ela= 37 file number=4 first dba=10908 block cnt=1 obj#
```

So as everybody expected append_values hint works as advertised :). Now let's try SYS_DL_CURSOR

```
SQL> insert /*+ SYS_DL_CURSOR */ into t1_dpi values (8,8);

1 row created.

SQL> select * from t1_dpi where rownum < 1;

no rows selected
```

It seems that we **can** query the table (no ORA-12838), so probably the HINT is not make direct path insert and Oracle is ignoring it. Let's check explain plan and 10046 trace to make sure:

```
-------------------------------------------+-----------------------------------+
| Id  | Operation                 | Name    | Rows  | Bytes | Cost  | Time      |
-------------------------------------------+-----------------------------------+
| 0   | INSERT STATEMENT          |         |       |       |     1 |           |
| 1   |  LOAD TABLE CONVENTIONAL  |         |       |       |       |           |
-------------------------------------------+-----------------------------------+


insert /*+ SYS_DL_CURSOR */ into t1_dpi values (8,8)
END OF STMT
PARSE #3:c=1000,e=931,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=1,plh=0,tim=1324126029360833
EXEC #3:c=0,e=169,p=0,cr=1,cu=7,mis=0,r=1,dep=0,og=1,plh=0,tim=1324126029361070
STAT #3 id=1 cnt=0 pid=0 pos=1 obj=0 op='LOAD TABLE CONVENTIONAL  (cr=1 pr=0 pw=0 time=
```

No luck here! Next option is now use a 10053 to unsure that CBO is not ignoring silently SYS_DL_CURSOR hint. Dumping Hints section shows even invalid/malformed hints here and for some invalid hints it shows err=1 or used=0. On our case used=0 shows that for some reason this hint is not beeing used:

```
SQL>  alter session set events='10053 trace name context forever, level 1';

Dumping Hints
=============
  atom_hint=(@=0x8cd666f0 err=0 resol=0 used=0 token=914 org=1 lvl=1 txt=SYS_DL_CURSOR
======================== END SQL Statement Dump ======================
```

Another ideia might be trying to use sqlldr (i googled some cases) to generate SYS_DL_CURSOR hint for direct path loading, but no luck here too:

```
[oracle@localhost scripts]$ sqlldr lcmarques/lcmarques@testSID control=loader.ctl log=l
Load completed - logical record count 9.


SELECT /* OPT_DYN_SAMP */ /*+ ALL_ROWS IGNORE_WHERE_CLAUSE NO_PARALLEL(SAMPLESUB
) opt_param('parallel_execution_enabled', 'false') NO_PARALLEL_INDEX(SAMPLESUB)
NO_SQL_TUNE */ NVL(SUM(C1),:"SYS_B_0"), NVL(SUM(C2),:"SYS_B_1") FROM (SELECT /*+
 NO_PARALLEL("T1_DPI") FULL("T1_DPI") NO_PARALLEL_INDEX("T1_DPI") */ :"SYS_B_2"
AS C1, :"SYS_B_3" AS C2 FROM "T1_DPI" "T1_DPI") SAMPLESUB
```

Last thing was to check in v$sql_hint for this hint:

```
SQL> select class from v$sql_hint where class like 'SYS_DL%';

CLASS
----------------------------------------------------------------
SYS_DL_CURSOR
```

Conclusion is simple, i can't make it. For some reason Oracle CBO is ignoring SYS_DL_CURSOR. Maybe this is valid under certain circunstances that i really don't know or is already depreceated for direct path inserts or 11g doesn't really like it.

tags: cbo, direct path

Email

08
DEZ 11

Oracle - Concorrência Simplificada

POR LUIS MARQUES, ÀS 19:44 | COMENTAR

Hoje a matéria é muito simples, é mais uma clarificação "Back to Basics" do que um post elaborado com investigação.
Serve assim de material de base para quando surgirem mais dúvidas deste tipo no meu local de trabalho ;)
A ideia é simples: Aplicar um select _durante_ o update e garantir que existe total consistência de leitura
e que não existe qualquer "bloqueio" em nenhuma sessão dada a arquitectura. Depois do exemplo prático, explicarei como funciona.

```
SQL> create table t2_locks as select rownum N1, dbms_random.string('A', 10) S1 from dua

Table created.

SQL> select sid from v$session where audsid=userenv('SESSIONID');

 SID
----------
 42

SQL> select sid from v$session where audsid=userenv('SESSIONID');

 SID
----------
 32
```

Validações na **SID 42**:

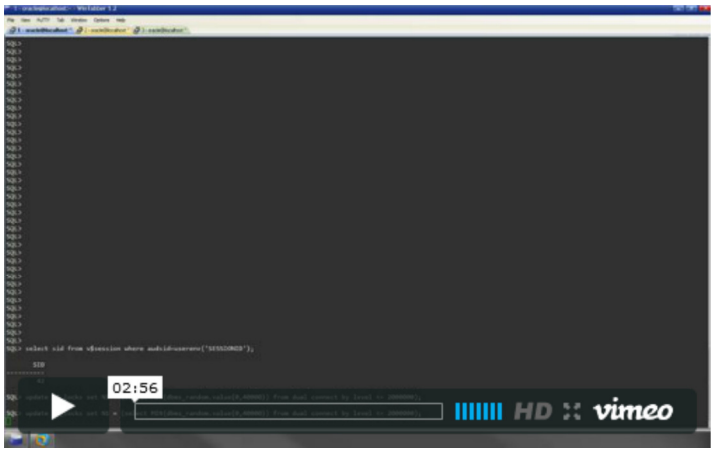
```
SQL> select MIN(N1) from t2_locks;

 MIN(N1)
----------
 1

SQL> select MAX(N1) from t2_locks;

 MAX(N1)
----------
 1000000

SQL> select avg(N1) from t2_locks;

 AVG(N1)
----------
 500000.5
```

Estas três validações irão permitir verificar a consistência dos dados durante o update e após o update (sem COMMIT).

Avançaremos então com o update demorado à SID 42, enquanto na SID 32 faremos as validações enquando decorre o update.
Para facilitar a compreensão, farei um pequeno video (desta vez sem som ;)):

O update será o seguinte:

```
update t2_locks set N1 = (select MIN(dbms_random.value(0,40000)) from dual connect by l
```



Como puderam ver no vídeo e apesar de não ter mostrado os locks feitos na v$lock durante o processo todo, não existe qualquer interferência das sessões entre elas, pois o Oracle possui um mecanismo de consistência de leitura que faz com que o resultado da query venha apenas de um ponto no tempo (na altura que a query começa) para uma ou mais transacções. Quando acontece o update (ou qualquer DML), o Oracle usa os rollback segments para providenciar dados consistentes, ou seja, os rollback segments contem os valores antigos dos dados que vão sendo sujeitos ao update (ou qualquer DML). O select poderá ler parte dos dados do table segment (da tabela em si) e outra parte dos rollback segment enquanto decorre o update ou este não está "commited". É basicamente um snapshot dos data blocks antes da sua modificação. Assim que é feito o commit e executado um select, os data blocks presentes no rollback segments são descartados e os dados do select virão exclusivamente do table segment (ou index segment, se for o caso).

É este rollback segment que permite que aquando de um rollback exista o processo de reversão dos dados sujeitos ao DML.
Explica também porque é que um rollback em média demora o mesmo tempo que a própria operação de DML que o originou, ou seja,
o Oracle tem que ir ao rollback segment lê-lo e repor tudo como estava, ou seja, a operação inversa aquando do DML a nível de
segments.

PS: Acabei por fazer metade em texto, metade em vídeo :-)

tags: clarifyme, concurrency, oracle

Email

04
DEZ 11

Oracle Locks - Parte I - "Isso já eu sabia"

POR LUIS MARQUES, ÀS 20:21 | COMENTAR

Desde o inicio que qualquer Oracle implementa mecanismos de lock para lidar com problemas de concorrência e de consistência na Base de Dados. Aceder a dados simultâneamente enquanto é possível dar a outras sessões dados consistentes, permitindo coerência na leitura e escrita dos dados permanece a função mais importante de uma Base de Dados. Sem estar a querer entrar pelo mundo teórico do modelo ACID, vou apenas demonstrar alguns pontos pertinentes dos mecanismos de Lock no Oracle.

Vou usar 2 sessões e analisar os respectivos locks à medida das operações que vou fazendo. Para facilitar a compreensão vou também explicando:

```
SQL> create table t1_locks as select rownum as N1, dbms_random.string('A',2) S1 from du

Table created.


SQL> select sid from v$session where audsid=userenv('SESSIONID');

 SID
----------
 40

SQL> select sid from v$session where audsid=userenv('SESSIONID');

 SID
----------

 29

SQL> column s1 format a3;
SQL> select * from t1_locks;

        N1 S1
---------- ---
         1 PN
         2 fP
         3 Yo
         4 Ur
         5 KX
```

Na **SID 40** será feito um insert:

```
SQL> insert into t1_locks (N1,S1) values(6,'SL');

1 row created.
```

Sem o commit executado veremos o que vê a sessão com o ID 29:

```
SQL> select * from t1_locks;

N1 S1
---------- ---
1 PN
2 fP
3 Yo
4 Ur
5 KX
```

Naturalmente a SID 29 não verá as alterações feitas pela SID 40, pois não existe commit. Veremos o que se passou no Oracle ao nível do mecanismo de locking:

```
SQL>  select sid,type,id1,lmode,request from v$lock where sid in (40,29);

       SID TY        ID1      LMODE    REQUEST
---------- -- ---------- ---------- ----------
        40 AE        100          4          0
        29 AE        100          4          0
        40 TM      74204          3          0
        40 TX     458771          6          0
```

Os "AE" são para desprezar (significa basicamente Application Edition lock, 11g apenas), mas os outros não. "TX" representa Transaction Lock, ou Transaction Enqueue e "TM" significa DML ou Table lock (DML Enqueue), ou seja, o TX representa a transação, é usado principalmente para prevenir que uma outra transacção modifique o mesmo registo, assim cada vez que um transacção necessita de modificar um registo adquire um TX. O "TM" é usado principalmente para gerir mecanismos de concorrência de operações DDL, como por exemplo, tentar fazer "drop" a uma tabela durante uma operação de insert (ou outro qualquer DML)
A coluna LMODE representa o modo de lock.

Na "TM" o modo de lock será row-exclusive (row-X, mode 3), ou seja, apenas os registos inseridos, enquanto do ponto de vista da "TX" o modo de lock será exclusive, já que o "TX" se refere à propria transacção.

Podemos de uma forma simples verificar se o lock "TM" está aplicado na tabela:

```
SQL> select name from sys.obj$ where obj#=74204;

NAME
------------------------------
T1_LOCKS
```

Confirma-se que existe um lock no objecto T1_LOCKS.
Já vimos que a SID 29 não consegue ver as alterações feitas pela SID 40, pois não houve lugar a commit.
O próximo passo é inserir um registo com a SID 29 e fazer update a um já existente (N1=1). Vejamos:

```
SQL> insert into t1_locks (N1,S1) values(7,'XX');

1 row created.

SQL> update t1_locks set S1='BB' where N1=1;

1 row updated.
```

Não será feito commit. Analisaremos os locks agora referentes às duas sessões:

```
SQL>  select sid,type,id1,lmode,request from v$lock where sid in (40,29);

       SID TY        ID1      LMODE    REQUEST
---------- -- ---------- ---------- ----------
        40 TM      74204          3          0
        29 TM      74204          3          0
        29 TX     393239          6          0
        40 TX     458771          6          0
```

Tudo faz sentido, 1 lock de cada tipo para cada SID diferente. As SID 29 e 40 estão a fazer "Table Lock" com row-S e ambas estão a trazer "Transaction Lock" para os registos inseridos e modificados. Até agora cada sessão consegue apenas visualizar aquilo que inseriu e/ou modificou. Vejamos:

SID 40:

```
SQL> select * from t1_locks; [Inclui INSERT do N1=6]

 N1 S1
---------- ---
 1 PN
 2 fP
 3 Yo
 4 Ur
 5 KX
 6 SL
```

SID 29:

```
SQL> select * from t1_locks; [Inclui INSERT do N1 = 7 e UPDATE do N1=1]

 N1 S1
---------- ---
 1 BB
 2 fP
 3 Yo
 4 Ur
 5 KX
 7 XX
```

Vamos piorar as coisas. Vamos na SID 40 tentar fazer update na tabela para N1=1, tal e qual fizemos na SID 29:

```
SQL> update t1_locks set S1='CC' where N1=1;
```

A sessão está bloqueada, não existiu lugar a update, até que a SID 29 faça commit/rollback das alterações. Podemos ver isto em "directo":

```
SQL>  select event, seconds_in_wait, sid from v$session_wait where sid in (40,29);

EVENT                SECONDS_IN_WAIT        SID
-------------------- --------------- ----------
SQL*Net message from             246         29
 client

enq: TX - row lock c             105         40
ontention
```

A coluna SECONDS_IN_WAIT permite-nos saber À quanto tempo o lock espera, ou seja, o evento "row lock contention" não é mais que uma sessão que espera por um row lock feito por outra sessão. No nosso caso o modo 6 (coluna LMODE) indica mesmo que para resolver este problema deverá ser feito um rollback ou commit dos dados. Na SID 29 faremos o commit:

```
SQL> commit;

Commit complete.
```

Entretanto na **SID 40**:

```
SQL> update t1_locks set S1='CC' where N1=1;

1 row updated.
```

Faremos agora um select * a ambas as tabelas:

SID 40:

```
SQL> select * from t1_locks;

 N1 S1
---------- ---
 1 CC
 2 fP
 3 Yo
 4 Ur
 5 KX
 7 XX
 6 SL

7 rows selected.
```

SID 29:

```
SQL> select * from t1_locks;

 N1 S1
---------- ---
 1 BB
 2 fP
 3 Yo
 4 Ur
 5 KX
 7 XX
```

O resultado é simples de interpretar. Enquanto que a SID 40, vê as alterações efectuadas pela SID 29, o contrário não acontece pois ainda não foi feito qualquer commit na SID 40. De notar que o valor para N1=1 é diferente em ambas as sessões já que na SID 40 foi feito o update para CC mas sem commit, sendo isto apenas visto pela SID 40

e não pela SID 29. De notar ainda que a SID 40 contêm um registo [N1=7] que foi inserido pela SID 29. Para terminar este exemplo bastante simples vamos ver de novo o estado dos locks:

```
SQL>  select sid,type,id1,lmode,request from v$lock where sid in (40,29);

       SID TY        ID1      LMODE    REQUEST
---------- -- ---------- ---------- ----------
        40 TM      74204          3          0
        40 TX     458771          6          0
```

Portanto apenas a SID 40 ainda detêm locks pois não foi feito qualquer rollback ou commit.

tags: locks

Email

19
NOV 11

Leitura de índices B-Tree: Alteração do Clustering Factor - Parte 2 (Reverse Key Index)

POR LUÍS MARQUES, ÀS 17:25 | COMENTAR

Tinha mostrado anteriormente que apenas reorganizando a tabela seria possível alterar o valor do CF e que uma reorganização do índice nada poderia fazer, pois o CF depende directamente da desordem dos blocos na tabela comparada com a organização no índice. No entanto existe uma forma relativamente simples de alterar o valor do CF que é usando reverse key index (não sei o termo em português).

Existem alguns casos onde este tipo de índices é útil, que foram desenhados essencialmente para resolver o problema de contenção nos blocos (index block contention). Basicamente ocorrem em cenários de muita concorrência (DML insert, update ou delete) onde as várias sessões concorrentes precisam de aceder ao mesmo bloco (chamado "hot block") gerando assim contenção ao nível do bloco do índice causando inumeros wait events do tipo "buffer busy waits", por exemplo.

Mas o post não é sobre os Reverse Key Index, mas sim sobre o CF e como este valor pode ser alterado usando um índice reverted. A alteração da ordem no índice leva a que a desordem na tabela seja diferente. Como exemplo simples, se um ID para inserir na tabela for gerado como 112233 será inserido como 332211 no índice. Este tipo de "reverse" permite que os inserts sejam espalhados por toda a estrutura do índice, evitando a contenção em apenas um só leaf bloco (o mais há direita). Com isto fazemos com que as entradas no índice deixem de estar ordenadas da forma natural e como conhecemos, ou seja, o 112234 a seguir ao 112233.
Apesar de parecer resolver alguns problemas nomeadamente em ambientes RAC muito concorridos, cria uma outra panóplia de problemas que não discutiremos neste post.

O código seguinte mostrará uma tabela, um índice normal que depois será convertido para "reverse" e os respectivos valores do CF após cada etapa:

```
SQL> create table t_cf2 as select ceil(dbms_random.value(0,100000)) N1 from dual connec

Table created.

SQL> create index i_cf2 on t_cf2(N1);

Index created.

SQL> exec dbms_stats.gather_table_stats(null,'T_CF2', cascade=>TRUE);

PL/SQL procedure successfully completed.

SQL>  select blevel, leaf_blocks, clustering_factor  from user_indexes where index_name

    BLEVEL LEAF_BLOCKS CLUSTERING_FACTOR
---------- ----------- -----------------
```

```
         1         222             99270

SQL> alter index i_cf2 rebuild reverse;

Index altered.


SQL>  select blevel, leaf_blocks, clustering_factor  from user_indexes where index_name

    BLEVEL LEAF_BLOCKS CLUSTERING_FACTOR
---------- ----------- -----------------
         1         222             99265
```

Temos uma valor diferente no CF, sendo que a diferença é pouca o que significa que o meu exemplo aqui não foi o melhor, no entanto, como nota final não devem de forma alguma usar os RKI sem cuidado, pois a ordem das chaves deixa de ser a natural e os "range scans" deixam de ser possíveis (predicados como BETWEEN, LIKE, > <) e o CBO vai por completo ignorar este tipo de índices. Estes dois posts sobre o CF foram apenas para fazer entender como o CF varia em função das várias ordens seja na tabela ou no índice.

tags: cbo, index

Email

05
NOV 11

Leitura de índices B-Tree: Alteração do Clustering Factor - Parte 1

POR LUÍS MARQUES, ÀS 20:47 | COMENTAR

Como já tinha explicado antes, o CF (vou me referir a partir de agora como CF, para me facilitar) é basicamente uma métrica que compara a ordem no índice com o grau de desordem na tabela, ou de outra forma se quiserem, é forma como os dados estão alinhados na tabela em relação à ordem no índice e o I/O necessário para ler a tabela inteira via full index scan.

Dado um índice é organizado e ordenado, um "rebuild" ao índice nunca (ou em circunstâncias especiais) alterará o valor do CF, pois a ordem das entradas no índice mantêm-se igual após o rebuild tal e qual a ordem dos registos na tabela. Assim é facil entender que para alterar o valor do CF temos que reorganizar a tabela associada e assim alterar o valor do CF.

É bom relembrar ainda que por norma um bom CF é um valor igual (ou abaixo, dado que podem existir blocos vazios abaixo do HWM) ao número de registos da tabela a que se refere o índice.

Vamos então ao exemplo:

```
SQL> create table t_cf as select dbms_random.value(0,500) as N1,
dbms_random.string('A',45) as A1,
dbms_random.string('l',45) as L1 from dual connect by level <= 400000;


Table created.


SQL> create index i1_cf on t_cf(A1);

Index created.


SQL> exec dbms_stats.gather_table_stats('LCMARQUES','T_CF', cascade=>TRUE);

PL/SQL procedure successfully completed.


SQL> select blevel, leaf_blocks, clustering_factor from user_indexes where index_name =

    BLEVEL LEAF_BLOCKS CLUSTERING_FACTOR
---------- ----------- -----------------
         2        3200            399942
```

```
SQL> select num_rows, blocks from user_tables where table_name = 'T_CF';

  NUM_ROWS     BLOCKS
---------- ----------
    400000       6779
```

Temos um índice recém criado (I1_CF) que tem um CF de 3999942, um valor bastante distante do número de blocos da tabela (6779), fazendo dele um índice com um mau CF (relembrar apenas que o CF é apenas um dos critérios escolhidos pelo CBO). Vamos então tentar reorganizar os mesmo dados e obter um CF diferente, um pouco melhor:

```
SQL> create table t2_cf as select * from T_CF;

Table created.

SQL> truncate table T_CF;

Table truncated.

SQL> insert into t_cf select * from t2_cf order by A1;

400000 rows created.

SQL> exec dbms_stats.gather_table_stats('LCMARQUES','T_CF', cascade=>TRUE);

SQL> select blevel, leaf_blocks, clustering_factor from user_indexes where index_name =

    BLEVEL LEAF_BLOCKS CLUSTERING_FACTOR
---------- ----------- -----------------
         2        2858              6558
```

A estratégia foi simples, existem várias formas de reorganizar a tabela e uma delas é usando os CTAS, ou seja criou-se uma tabela auxiliar com os dados e inseriu-se posteriormente na tabela original (T_CF) ordenamente pela coluna que o índice contem. Assim os registos inseridos estão agora organizados exactamente da mesma forma que o índice, levando como é obvio a um valor bastante bom de CF e a um decréscimo do I/O na próxima visita à tabela por índice full scan.

No entanto, existe uma dúvida e é legitima, dado que criou-se uma tabela com 400k registos, depois um índice, levantou-se as estatísticas e o valor do CF saiu péssimo. Isto acontece pois na criação do índice inicialmente os registos na tabela estão desorganizados e na natureza de qualquer índice estes encontram-se devidamente ordenados segundo as colunas presentes, dai que a opção viável para alterar o CF será reordenar a tabela.

Vimos aqui como um índice recém-criado pode não corresponder às espectativas em termos de custo, levando a um excessido consumo de I/O dado que será necessário re-visitar o mesmo bloco "n" vezes dada a aleatoriedade dos blocos (e registos) na tabela, para tal a análise do CF deve ser cuidada, dado que não é o único factor que influencia a decisão do CBO, mas é um bastante importante. Importante também que se a tabela conter mais índices, decidir o critério de organização nem sempre é facil.

Existem outras formas de alterar o valor do CF, entre elas mexendo apenas no índice, mas deixarei isso para a parte 2 deste ponto.

tags: btree, index

Email

22
OUT 11

Leitura de índices B-Tree: A teoria e prática simplificada do Clustering Factor

POR LUÍS MARQUES, ÀS 15:19 | COMENTAR

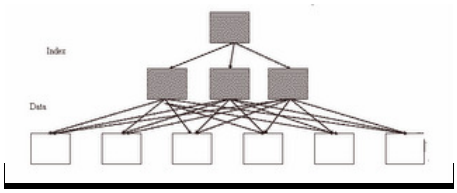
O aviso:

É bastante comum, e má prática no geral dizer que um bom índicetem um baixo clustering_factor e um mau índice tem um alto clustering_factor.

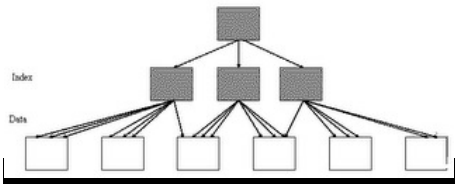
Obviamente que em determinados casos isto acontece realmente e esta assunção é verdadeira. No entanto o valor em si, desprendido de outras comparações é completamente inútil na avaliação da qualidade do índice, pois um valor de por exemplo 1000 no clustering_factor será óptimo para uma tabela com 1000 blocos, provavelmente péssimo para uma tabela com 10 blocos.
Assim, o clustering_factor não é mais que uma métrica que compara a ordem no índice com o grau de desordem na tabela, ou seja, para calcular o clustering_factor o CBO navega na tabela pela ordem do índice(muito importante) e regista quantas vezes salta de um bloco para outro na tabela. Cada vez que existe um salto, o valor do clustering_factor aumenta.
Este contador vai servir ao Oracle para decidir o custo associado ao uso do índice(range scan/full) em comparação com um full table scan.

A representação:

A imagem abaixo representa um **MAU** clustering_factor. A primeira entrada do índiceaponta para o primeiro bloco, a segunda entrada do índiceaponta para para o segundo bloco na tabela, a terceira entrada para o terceiro bloco e etc. Podem assim verificar o quão espanhados estão os dados fisicamente, o que obriga num index range scan ou full index scan, o CBO a saltar de um bloco para o outro e a visitar o mesmo bloco várias vezes pois os dados estão espalhados. Como sabem o I/O é caro e paga-se bastante caro por um clustering_factor alto. Cada vez que há este salto aumenta o VALOR do clustering_factor.



Neste caso, a imagem abaixo representa um **BOM** clustering_factor, numa leitura do índiceo CBO não tem que saltar de bloco em bloco, já que as entradas do índiceapontam para o mesmo bloco.



A interpretação:

Para exemplo prático, vamos usar o índicee tabela criados anteriormente e verificar o clustering_factor associado.

```
SQL> select index_name, clustering_factor from user_indexes where index_name = 'I1';

INDEX_NAME CLUSTERING_FACTOR
------------------------------ -----------------
I1 372


SQL> column segment_name format a14
SQL> select segment_name, blocks from user_segments where segment_name = 'INDX_CON1';

SEGMENT_NAME BLOCKS
-------------- ----------
INDX_CON1 384


SQL> select count(1) from indx_con1;

 COUNT(1)
```

```
----------
 200000
```

Como interpretar isto o clustering_factor:

1 - Normalmente o clustering_factor está entre o número de blocos da tabela e o número de registos de uma tabela, o que
no nosso caso não se verifica. Ver ponto 2)

2 - Um clustering_factor pode ser menor que o número de blocos na tabela se existem blocos vazios na tabela abaixo
da HWM (High Water Mark), ou existem muitos registos null para as colunas indexadas. Vejamos:

```
SQL> select count(*) from indx_con1 where R0 is null;

 COUNT(*)
----------
 0
```

Confirmamos facilmente que as colunas indexadas (R0) não teem valores null, logo assumimos que o valor do clustering_factor é abaixo do número de blocos pois existem blocos vazios abaixo da HWM.

3- O clustering_factor nunca pode ser maior que o número de registos na tabela.

4 - Um **BOM** clustering_factor é igual (ou abaixo, como o caso) ao valor do número de blocos da tabela.
5 - Um **MAU** clustering_factor é igual, ou próximo ao valor do número de registos na tabela.

6 - Caso o clustering_factor seja MAU (próximo do valor do número de registos) é provavel que o índicenão
seja escolhido pelo CBO, a não ser que a selectividade seja muito alta.

7 - Reorganizar um índicenão altera o clustering_factor, apenas a reorganização da tabela o permite, ordenando
os dados conforme a ordem do índice.

8 - É possível ajustar o clustering_factor MANUALMENTE com o uso do package DBMS_STATS.SET_INDEX_STATS.

tags: btree, index

Email

08
OUT 11

Leitura de índices B-Tree - treedump

POR LUIS MARQUES, ÀS 14:52 | COMENTAR

Pegando no exemplo dado no post anterior e usando o mesmo índice, vamos dar uma olhada mais profunda no índice criado anteriormente (i1). O mecanismo aplicado consegue com muito detalhe dar-nos informações interessantes sobre a constituição do índice B-Tree.

O evento associado é o treedump e permite a partir de um object_id que neste caso vai ser um índice obter um ficheiro de trace com o súmario do índice. Vejamos:

```
SQL> select object_id from user_objects where object_name = 'I1';

 OBJECT_ID
----------
     73897

SQL> alter session set events 'immediate trace name treedump level 73897';

Session altered.
```

Viajando até ao directório que contem o trace (ver parametro user_dump_dest) temos:

```
----- begin tree dump
branch: 0x415129 4280617 (0: nrow: 445, level: 1)
   leaf: 0x41512a 4280618 (-1: nrow: 485 rrow: 485)
   leaf: 0x41512b 4280619 (0: nrow: 479 rrow: 479)
   leaf: 0x41512c 4280620 (1: nrow: 479 rrow: 479)
   leaf: 0x41512d 4280621 (2: nrow: 479 rrow: 479)
   leaf: 0x41512e 4280622 (3: nrow: 479 rrow: 479)
    ....
   leaf: 0x41543c 4281404 (441: nrow: 449 rrow: 449)
   leaf: 0x41543d 4281405 (442: nrow: 449 rrow: 449)
   leaf: 0x41543e 4281406 (443: nrow: 16 rrow: 16)
----- end tree dump
```

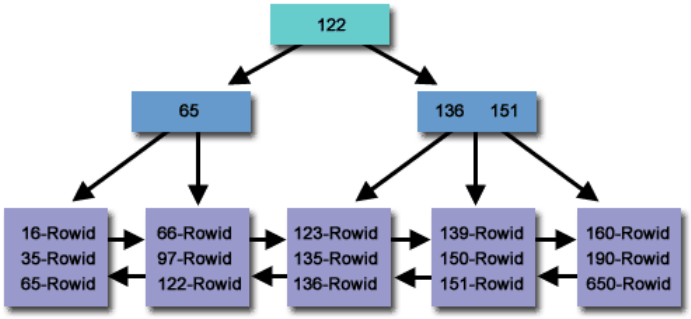
Como foi dito o ficheiro gerado contém a estrutura do índice, nomeadamente uma linha no trace por cada bloco no índice. Estas linhas estão ordenadas conforme a estrutura da árvore e os números hexadecimais/decimais apresentados são o endereço do bloco.

Vamos então tentar entender o resultado deste dump ao índice:

- A primeia linha representa o root block (nó pai) com profundidade 1 (**blevel=1**). De lembrar que este índice é apenas (é pequeno) constituido pelo nó pai e pelos nós filhos.

- O **nrow=445** significa que no root block (nó pai único) existem 445 apontadores para os nós filhos, assim cada vez que for necessário ler o índice, no root block existirá directamente o apontador para o leaf block correspondente.

De notar e muito _importante_ que numa árvore mais complexa (ou seja **blevel > 1**), este valor teria outro significado, ou seja, o nrow do root block seria o número de apontadores para os branch blocks. Usando uma imagem será mais fácil de entender:



Neste exemplo o root block (denominado aqui 122, não importa o valor) teria um **nrows=2**, ou seja teria dois apontadores para os nós filhos que neste caso são branch blocks.

- A segunda linha que começa por um estranho -1 (tanto quanto percebi, a contagem inicia em -1 para denominar o primeiro bloco seja branch ou leaf) representa o primeiro leaf block, notado também pois não há level, pois não faz sentido nos leaf blocks.

- O **rrow** e o **nrow** nos leaf blocks definem respectivamente o número de linhas no bloco e o seu tamanho no block row directory (um dia explico isto, mas basicamente é um conjunto de informação que diz ao Oracle em cada bloco onde cada linha começa e termina).
De notar também que o **rrow = nrow** em todos os blocos significa que não houve block split, pois não foi feito nenhum insert após o índice ter sido criado.

- Por fim, a última linha (**443**) é o último leaf block e como podem ver pelo nrow e pelo rrow não está completo (cheio), pois o valor de 16 é aquém do suportado nos blocos anteriores. Num insert posterior este valor vai concerteza mudar.

Espero que tenha sido claro, e não usem a metodologia "vou-criar-o-índice-e-depois-logo-se-vê-se-melhora-a-performance", caso contrário este post é inútil.

Referências:
- Lista de eventos: http://www.adp-gmbh.ch/ora/tuning/diagnostic_events/list.html

tags: btree, index

Email

25
SET 11

Leitura de índices B-Tree - Considerações

POR LUÍS MARQUES, ÀS 12:08 | COMENTAR

Deve-se fazer apenas considerações (aka "bitaites") sobre o uso de determinados índices se realmente sobermos aquilo que estamos a falar, caso contrário, caímos na especulação e somos o próximo treinador de bancada falhado.

O post de hoje vai ajudar a entender isso, no entanto não esperem conversa para iniciantes.

A abordagem vai ser basicamente um misto de teoria e prática sobre índices Btree.

Para iniciar criaremos uma tabela:

```
SQL>  create table indx_con1 as select rownum as R0, mod(rownum, 200) as R1 from dual c

Table created.

SQL> create index i1 on indx_con1(R0);

Index created.
```

A primeira análise a fazer será na tabela dba_indexes para analisarmos os dados relativos ao índice que acabámos de criar. Vejamos:

```
SQL> select blevel, leaf_blocks, clustering_factor from dba_indexes where index_name =

    BLEVEL LEAF_BLOCKS CLUSTERING_FACTOR
---------- ----------- -----------------
         1         445               372
```

Isto significa que o índice consiste basicamente apenas pelo Root Block (nó pai), informação dada pelo BLEVEL=1 (profundidade da árvore), por 445 leaf blocks (nós filhos). Como tal se necessitarmos de ler uma determinada entrada do índice será necessário ler primeiramente o root block (nó pai) e posteriormente o leaf block especifico (nó filho). No total serão 2 leituras.

Vamos então verificar se esta explicação coincide com aquela que o CBO nos fornece.

```
SQL> set autotrace traceonly explain statistics
SQL> select * from indx_con1 where R0=22;

| Id  | Operation                    | Name      | Rows  | Bytes | Cost (%CPU)| Time
-------------------------------------------------------------------------------------

|   0 | SELECT STATEMENT             |           |     1 |    26 |     2   (0)| 0 0:00:0

|   1 |  TABLE ACCESS BY INDEX ROWID| INDX_CON1 |     1 |    26 |     2   (0)| 0 0:00:0

|*  2 |   INDEX RANGE SCAN           | I1        |     1 |       |     1   (0)| 0 0:00:0
```

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


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

   2 - access("R0"=22)

Note
-----
   - dynamic sampling used for this statement (level=2)


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

Podemos tirar várias ilações sobre os resultados anteriores nomeadamente:

1 - O custo de aceder ao índice (e apenas ao índice) é de 1, significa que o CBO sabe que a profundidade da árvore é 1 (BLEVEL=1), o índice é pequeno logo o root block (nó pai) está em cache e como tal é excluido do cálculo do custo.

2 - O custo TOTAL do plano (e não apenas de ler a entrada no índice) é 2 (ver Cost no Id 0 no plano), ou seja, dado que o root block está excluído do custo, é necessário uma leitura para o leaf block (nó filho) e outra leitura para ler o bloco na tabela, pois neste caso e dada a query executada apenas a leitura do índice não é suficiente pois não contem todas as colunas que pretendemos devolver.

3 - O número estimado de linhas a devolver é 1 (ver coluna Rows).

4 - O número de consistent gets é 4: 1 para ler o root block (nó pai em cache), 1 para ler o leaf block (nó filho), 1 para ler o table block (acesso à tabela) e por fim mais 1 pois o indice não é unique e é necessário perceber se há mais registos a retornar.

Como conclusão, uma análise simples mas essencial para quem pretende saber realmente do que fala ;)

tags: btree, index

Email

« ANTERIOR INÍCIO