

Oracle Technet: [Ask Tom](#)

## On Partitioning and Pipelining

By Tom Kyte

Traduzione di Enrico Cirillo

Ho un Database OLTP con un alto numero di transazioni in cui ho creato delle partizioni per facilitare l'eliminazione dei dati obsoleti. Tutti gli indici non relativi a date sono *global nonpartitioned*. Poiché per aggiornare gli indici globali occorre molto tempo (le mie tabelle hanno una dimensione da 4 a 25 gigabytes), sono preoccupato riguardo al Locking e alla concorrenza sulla tabella quando provo ad eseguire DROP PARTITION ... UPDATE GLOBAL INDEXES, perché contemporaneamente vengono eseguite istruzioni di UPDATE e INSERT con alti tassi di DML (60 – 80 al secondo per la tabella).

Il problema non è la dimensione aggregata delle tabelle, ma piuttosto il formato specifico delle partizioni. Se si divide la tabella in partizioni abbastanza piccole, ogni DROP PARTITION non sarà più così grande e terminerà rapidamente.

In risposta alla domanda sul locking, finché la partizione da eliminare non ha transazioni attive su di essa (durante l'esecuzione del comando DROP, si otterrà un errore ORA-54 se la partizione ha transazioni attive), il comando DROP potrà essere eseguito – perfino con comandi concurrent/outstanding DML attivi su altre partizioni. Il comando UPDATE GLOBAL INDEXES può essere eseguito attraverso comandi tipo DML (come se fosse invocato il comando di DELETE) e terminerà con successo contemporaneamente ad altre transazioni.

### UNION o UNION ALL

Sto effettuando il tuning di una query in cui c'è una clausola WHERE EXISTS, come questa: WHERE EXISTS (SELECT 'x' FROM t1 WHERE ... UNION SELECT 'x' FROM t2 WHERE ... UNION SELECT 'x' FROM t3 WHERE ... ). La clausola WHERE EXISTS impiega un tempo non fissato. C'è una maniera migliore di eseguire la Query?

Certamente. In giro si vede frequentemente questo: l'uso di UNION quando UNION ALL sarebbe molto più efficiente. Il problema è che eseguendo una clausola UNION Oracle cerca tutti i record che soddisfano la condizione e li elimina dal risultato. Per vedere cosa si intende, si può confrontare semplicemente le seguenti queries:

```
SQL> select * from dual
      2 union
      3 select * from dual;
```

```
D
-
X
```

```
SQL> select * from dual
      2 union ALL
      3 select * from dual;
```

```
D
-
X
X
```

Si noti come la prima query restituisca solo un record, mentre la seconda due. Una UNION forza un ordinamento e una rimozione di valori duplicati. Nella maggior parte dei casi è completamente inutile. Per vedere come ciò potrebbe essere importante, verranno usate le tabelle del dizionario di dati per eseguire una query con la clausola WHERE EXISTS che usi UNION e UNION ALL e verranno paragonati i risultati con TKPROF. I risultati sono drammaticamente diversi.

Per prima la query con la clausola UNION:

```
SQL> select *
      2 from dual
      3 where exists
      4 (select null from all_objects
      5 union
      6 select null from dba_objects
      7 union
      8 select null from all_users);
```

call	cnt	cpu	ela	query
Parse	1	0.01	0.00	0
Execute	1	2.78	2.75	192234
Fetch	2	0.00	0.00	3
total	4	2.79	2.76	192237

Come si può vedere, c'è stato moltissimo lavoro - più di 192000 I/O solo per vedere se si deve prendere o no la riga dalla tabella DUAL. Adesso si usi la clausola UNION ALL:

```
SQL> select *
      2 from dual
      3 where exists
      4 (select null from all_objects
      5 union all
      6 select null from dba_objects
      7 union all
      8 select null from all_users);
```

call	cnt	cpu	ela	query
-----	-----	-----	-----	-----
Parse	1	0.00	0.00	0
Execute	1	0.01	0.00	9
Fetch	2	0.00	0.00	3
-----	-----	-----	-----	-----
total	4	0.01	0.00	12

Che cambiamento! Quello che è accaduto è che la clausola WHERE EXISTS ha terminato l'esecuzione della subquery quando ha trovato la prima riga, e, per il fatto che il DB non ha dovuto perdere tempo ad eliminare le righe superflue, ottenere la prima riga è stato effettivamente molto veloce.

N.B.: Se si può usare la clausola UNION ALL, occorre usarla al posto di UNION per evitare operazioni non necessarie probabilmente nella maggior parte delle volte.

## **FUNZIONI PIPELINED**

**Si può illustrare l'uso delle funzioni pipelined con (EMP, DEPT) con un esempio semplice? In che circostanze l'uso di una funzione pipelined può essere efficace?**

Le funzioni *pipelined* sono semplicemente del codice che si può far finta che sia una tabella del database. Le funzioni *pipelined* danno la capacità di usare SELECT \* FROM <PLSQL\_FUNCTION>;.

In qualsiasi momento si pensi di avere la capacità di usare SELECT \* da una funzione anziché da una tabella, una funzione *pipelined* potrebbe essere utile. Si consideri il seguente processo extract/trasform/load (ETL), per cui un *file flat* è inserito in una funzione PL/SQL che lo trasforma, ed i dati trasformati siano usati per aggiornare i dati attuali della tabella. Verranno dimostrate alcune caratteristiche del DB, compreso le tabelle esterne, le funzioni *pipelined* e l'operazione MERGE.

Per generare ed usare una tabella esterna, devo usare un oggetto directory. Si inizi a crearlo in */tmp*:

```
SQL> create or replace
  2  directory data_dir as '/tmp/'
  3  /
Directory created.
```

Quindi si crei una tabella esterna, la cui definizione sembra un *controlfile*, poichè, in effetti, parte della creazione di una tabella esterna crea un *controlfile*:

```

SQL> create table external_table
  2  (EMPNO NUMBER(4) ,
  3    ENAME VARCHAR2(10),
  4    JOB VARCHAR2(9),
  5    MGR NUMBER(4),
  6    HIREDATE DATE,
  7    SAL NUMBER(7, 2),
  8    COMM NUMBER(7, 2),
  9    DEPTNO NUMBER(2)
 10 )
 11 ORGANIZATION EXTERNAL
 12 (type oracle_loader
 13   default directory data_dir
 14   access parameters
 15   (fields terminated by ',')
 16   location ('emp.dat')
 17 )
 18 /

```

Table created.

Adesso si userà l'utility *flat* (che si scarica da: <http://asktom.oracle.com/~tkyte/flat>) per creare un *file flat* dalla tabella EMP.

```

SQL> host flat scott/tiger -
  > emp > /tmp/emp.dat

```

Siamo pronti a testare la tabella esterna; il *file flat* appena creato funziona come una tabella DB

```

SQL> select empno, ename, hiredate
  2  from external_table
  3  where ename like '%A%'
  4  /

```

EMPNO	ENAME	HIREDATE
7499	ALLEN	20-FEB-81
7521	WARD	22-FEB-81
7654	MARTIN	28-SEP-81
7698	BLAKE	01-MAY-81
7782	CLARK	09-JUN-81
7876	ADAMS	12-JAN-83
7900	JAMES	03-DEC-81

7 rows selected.

Verrà creata un procedura PL/SQL ETL per alimentare il *file flat* e per inserire o unire i dati veri.

Una funzione *pipelined* deve restituire un tipo *collection*, e in questo caso si vuole restituire un oggetto che assomigli alla tabella EMP; per far ciò genero il tipo oggetto scalare e quindi una tabella di quel tipo.

```

SQL> create or replace type
  2 emp_scalar_type as object
  3 (EMPNO NUMBER(4) ,
  4  ENAME VARCHAR2(10),
  5  JOB VARCHAR2(9),
  6  MGR NUMBER(4),
  7  HIREDATE DATE,
  8  SAL NUMBER(7, 2),
  9  COMM NUMBER(7, 2),
 10  DEPTNO NUMBER(2)
 11 )
 12 /
Type created.

```

```

SQL> create or replace type
  2 emp_table_type as table
  3 of emp_scalar_type
  4 /
Type created.

```

Adesso si può creare la funzione *pipelined*. La funzione ETL è molto semplice: modifica la colonna *ename*, ma la logica può essere complessa a piacimento, come includere il logging degli errori e così via;

```

create or replace function emp_etl
(p_cursor in sys_refcursor)
return emp_table_type
PIPELINED
as
  l_rec external_table%rowtype;
begin
  loop
    fetch p_cursor into l_rec;
    exit when (p_cursor%notfound);
    -- validation routine
    -- log bad rows elsewhere
    -- lookup some value
    -- perform conversion
    pipe row(
emp_scalar_type(l_rec.empno,
  LOWER(l_rec.ename),
  l_rec.job,
  l_rec.mgr,
  l_rec.hiredate,
  l_rec.sal,
  l_rec.comm,
  l_rec.deptno) );
    end loop;
  return;

```

```
end;  
/  
Function created.
```

La funzione *pipelined* emp\_etl funziona come se fosse una tabella. La query seguente seleziona due colonne, empno e ename, dalla funzione, e la funzione è legata alla tabella external.

```
SQL> select empno, ename  
2      from TABLE(emp_etl(  
3          cursor(select *  
4                  from external_table  
5                  ) ) )  
6      where ename like '%a%';
```

EMPNO	ENAME
7499	allen
7521	ward
7654	martin
7698	blake
7782	clark
7876	adams
7900	james

7 rows selected.

Notare l'uso della parola chiave PIPELINED nella definizione della funzione; la parola chiave è obbligatoria nella creazione della funzione *pipelined*. Notare anche l'uso della direttiva PL/SQL pipe row – cosa che rende veramente interessante la funzione *pipelined*. La direttiva pipe row restituisce immediatamente i dati al cliente, cioè l'output della funzione arriva prima che la funzione arrivi all'ultimo record dei dati. Se il cursore che utilizzo con questa funzione restituisce 1.000.000 di righe, io non dovrò aspettare che il PL/SQL processi tutte le righe per ottenere la prima; i dati cominciano ad essere restituiti non appena sono pronti. Ecco perché queste sono denominate funzioni *pipelined*: i dati fluiscono come se fossero in un grande tubo (pipe) – dal cursore alla funzione di PL/SQL e poi al programma chiamante.

Ora, per finire il lavoro, genererò una tabella dei dati che vorrei aggiornare dal sistema sorgente, che mi trasmette il *file flat* che ho prodotto sopra. La logica è la seguente: se il record esiste già nella mia base di dati, si effettua l'UPDATE dei campi ename e sal; se il record non esiste, se ne effettua l'INSERT.

Si inizia con un pò di dati dalla tabella di EMP.

```
SQL> create table emp as  
2      select * from scott.emp  
3      where mod(empno,2) = 0;
```

Table created.

Ed ecco la funzione MERGE che gestisce i dati dal *file flat*, attraverso ETL, verso la tabella, senza occupare il disco fisico con *files* intermedi:

```
SQL> merge into EMP e1
  2  using (select *
  3          from TABLE
  4          (emp_etl(
  5            cursor(select *
  6                  from external_table))
  7          )
  8  ) e2
  9  on (e2.empno = e1.empno)
 10  when matched then
 11    update set e1.sal = e2.sal,
 12             e1.ename = e2.ename
 13  when not matched then
 14    insert (empno, ename, job, mgr,
 15           hiredate, sal, comm, deptno)
 16  values (e2.empno, e2.ename,
 17         e2.job, e2.mgr,
 18         e2.hiredate, e2.sal,
 19         e2.comm, e2.deptno)
 20  /
14 rows merged.
```

## **L'ULTIMO SABATO DI OGNI MESE**

**Ho bisogno di creare una query che restituisca la data dell'ultimo sabato di un mese qualsiasi. Come?**

E' abbastanza semplice con le funzioni NEXT\_DAY e LAST\_DAY. Si può prendere l'ultimo giorno del mese con LAST\_DAY, sottrarre 7 giorni, e cercare il primo sabato successivo. Creerò due queries, una che restituisce l'ultimo sabato per ogni mese dell'anno corrente, e un'altra che accetta come parametro un mese e ne restituisce l'ultimo sabato.

Il primo listato è relativo alla prima query:

```
SQL> select next_day(
  2  last_day(
  3    add_months(trunc( sysdate, 'y' ), rownum-1 ) )-7,
  4    to_char(to_date('29-01-1927', 'dd-mm-yyyy'), 'DAY' ))
  5  from all_objects
  6  where rownum <= 12;
```

```

NEXT_DAY(
-----
31-JAN-04
28-FEB-04
27-MAR-04
24-APR-04
29-MAY-04
26-JUN-04
31-JUL-04
28-AUG-04
25-SEP-04
30-OCT-04
27-NOV-04
25-DEC-04

```

12 rows selected.

Qualcuno si chiederà cosa significa la data 29-01-1927 che è inserita nella query. E' solo una data qualsiasi che cadeva di sabato. Andava bene qualsiasi data che cadeva di sabato. Ho fatto così, invece di inserire l'abbreviazione del nome del giorno della settimana 'SAT', perché è legato alla lingua usata. Ad esempio in italiano sarebbe stato 'SAB'. Invece così la query è esportabile ovunque.

Il secondo listato è relativo alla query parametrica da eseguire in SQL\*Plus:

```

SQL> select next_day(
      2      last_day(to_date('&YOUR_MONTH', 'MM')) -7,
      3      to_char(to_date('29-01-1927', 'dd-mm-yyyy'), 'DAY'))
      4  from dual;

```

Enter value for your\_month: 01

```

old 2:      last_day( to_date( '&YOUR_MONTH', 'MM' ))-7,
new 2:      last_day( to_date( '01', 'MM' ))-7,

```

```

NEXT_DAY(
-----
31-JAN-04

```

Questa query funziona con l'anno corrente. E' semplice estendere l'uso della funzione, modificando il valore del parametro MM a ciò che si desidera (ad esempio YYYYMM).

Un lettore ("Ant" da New York) ha trovato un miglioramento alla prima query, sfruttando la nuova clausola SQL MODEL di Oracle 10g, in maniera da restituire le righe senza effettuare alcun I/O nel database. Riportiamo la sua soluzione nel listato seguente:

```

SQL> select
      1 next_day(last_day(add_months(trunc(sysdate, 'y'), cell))-7,
      2 to_char( to_date('29-jan-1927', 'dd-mon-yyyy'), 'DAY'))

```

```
3 from dual
4 model return all rows
5 dimension by (0 attr)
6 measures (0 cell)
7 rules iterate (12) (
8 cell[iteration_number] = iteration_number);
```

#### Execution Plan

```
-----
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=1)
1  0    SQL MODEL (ORDERED FAST)
2  1    FAST DUAL (Cost=2 Card=1)
```

#### Statistics

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

Per approfondire la clausola MODEL e per poterla usare al meglio, date un'occhiata all'eccellente [articolo](#) di Jonathan Gennick nel numero di Gennaio/Febbraio 2004 di Oracle Magazine.

### **CONFUSIONE SU STATSPACK**

Sto lavorando con la versione Oracle Database 9.2.0.4 su ALPHA OPEN VMS con due CPU. Ho bisogno di aiuto per interpretare i risultati di un report prodotto da STATSPACK. La finestra di snapshot dura 10 minuti. In 10 minuti ci sono 600 secondi. Quando però vedo la classifica Top 5 degli eventi che durano di più, vedo che ci sono tempi d'attesa ben oltre i 900 secondi per `db_file_sequential_read` – e il totale di tutti gli eventi di attesa va ben oltre i 1000 secondi. Come può essere? La mia finestra temporale dura solo 600 secondi. Inoltre osservo solo 54 secondi di CPU per 27 sessioni. Sto cercando di calcolare un tempo medio di servizio di STATSPACK e sono un po' confuso.

Bene, se hai 1000 persone che aspettano per un secondo simultaneamente, hai un tempo di attesa di 1000 secondi in quel secondo.

L'unica cosa che quel particolare *report* di STATSPACK dice è che il tempo di attesa cumulativo per `db_file_sequential_read` (dovuto ad abuso di indici, secondo me) è di 906 secondi.

Supponiamo di avere 27 sessioni. Significa che il tempo medio di attesa per sessione è 33 secondi ( $27 \times 33 = 891$  secondi) durante la tua finestra temporale di 10 minuti (600 secondi).

Per la CPU, potrebbe significare che ogni sessione ha usato una media di due secondi di tempo CPU (ripeto) nel periodo di osservazione. Di nuovo, potrebbe non essere vero. Il problema è che in 26 sessioni è stata chiamata una Stored procedure giusto prima dell'inizio, e l'esecuzione della procedura era ancora attiva (cioè il controllo non era ancora ritornato al client) quando la finestra temporale è terminata e quella procedura consuma CPU – non faceva I/O, ma solo usava pesantemente la CPU – e queste procedure contribuiscono con tempo CPU zero al report, poiché quel tempo viene calcolato solo alla fine dell'esecuzione delle procedure, e nel tuo caso non erano ancora completate.

Così una procedura la cui durata, ad esempio, è di 10 ore, in esecuzione prima che iniziasse la finestra temporale, e che finisce all'interno della finestra, contribuisce al calcolo del tempo CPU.

Viceversa, la stessa procedura che inizia all'interno della finestra e non finisce prima della fine, non contribuisce.

Per cui occorre porre molta attenzione al tempo CPU; a volte è fuorviante nei sistemi batch, e assolutamente non funzionante per i sistemi transazionali. Tutte queste considerazioni rendono fisicamente impossibile calcolare il tempo medio di un servizio tramite i report di STATSPACK; i dati da utilizzare sono troppo aggregati. Non si può in nessuna maniera calcolare il tempo medio di un servizio tramite STATSPACK – ignora qualsiasi cosa che dica il contrario.

L'unica maniera di calcolare in maniera corretta i tempi di un servizio è di avere delle applicazioni client che registrano l'attività in qualche posto.

### **Trucchetto**

Volete rendere il vostro codice JDBC più efficiente? Date un'occhiata a <http://oracle.com/technology/products/oracle9i/daily/jun24.html>, e abilitate la nuova caratteristica di JDBC che lo fa funzionare come PL/SQL.

---

Tom Kyte ([thomas.kyte@oracle.com](mailto:thomas.kyte@oracle.com)) è in Oracle dal 1993. Kyte è vicepresidente di Oracle Government, Education and Healthcare e l'autore di *Effective Oracle by Design* (Oracle Press, 2003) e di *Export One-on-One* (Apress, 2003).

Copyright © 1999-2005, Oracle Corporation. All Rights Reserved.