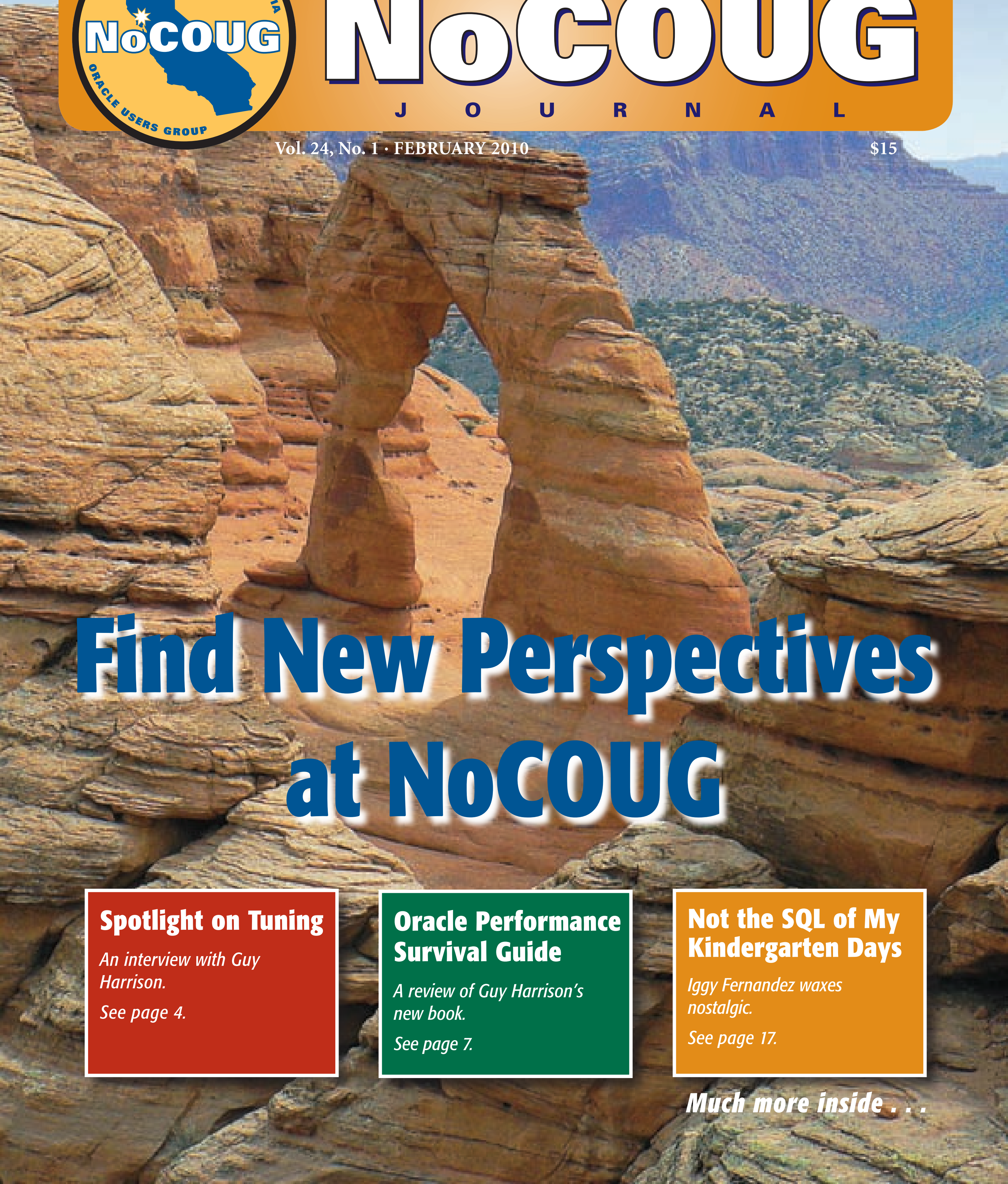
Official Publication of the Northern California Oracle Users Group

NoCOUG JOURNAL

Vol. 24, No. 1 · FEBRUARY 2010 $15



Find New Perspectives at NoCOUG

Spotlight on Tuning

An interview with Guy Harrison.

See page 4.

Oracle Performance Survival Guide

A review of Guy Harrison's new book.

See page 7.

Not the SQL of My Kindergarten Days

Iggy Fernandez waxes nostalgic.

See page 17.

Much more inside . . .

Not the SQL of My Kindergarten Days

by Iggy Fernandez

Iggy Fernandez

I remember, I remember
The fir-trees dark and high;
I used to think their slender tops
Were close against the sky:
It was a childish ignorance,
But now 'tis little joy
To know I'm farther off from heaven
Than when I was a boy.

—Thomas Hood [1799–1845]

The relational model was invented by IBM researcher Edgar Codd when I was in kindergarten. Here's a picture of my kindergarten class; one of those cute little tykes is me.

Codd made proposals for "data base sublanguages" in his paper *Relational Completeness of Data Base Sublanguages* (1972). As IBM researcher Donald Chamberlin recalled later: *[Codd] gave a seminar and a lot of us went to listen to him. This was as I say a revelation for me because Codd had a bunch of queries that were fairly complicated queries and since I'd been studying CODASYL, I could imagine how those queries would have been represented in CODASYL by programs that were five pages long that would navigate through this labyrinth of pointers and stuff. Codd would sort of write them down as one-liners. These would be queries like, "Find the employees who earn more than their managers." He just whacked them out and you could sort of read them, and they weren't complicated at all, and I said, "Wow." This was kind of a conversion experience for me, that I understood what the relational thing was about after that.*

Donald Chamberlin and fellow IBM researcher Raymond Boyce went on to implement the first "data base sublanguage" based on Codd's proposals and described it in a short paper titled *SEQUEL: A Structured English Query Language* (1974). The acronym SEQUEL was later shortened to SQL because SEQUEL was a trademarked name; this means that the correct pronunciation of SQL is sequel not es-cue-el. Codd's paper and Chamberlin's paper can be downloaded from the Internet; they were written using manual typewriters, and Codd's paper even contains handwritten corrections.

The SQL of today is much more powerful than the SQL of my kindergarten days. The latest ANSI SQL standard comprises thousands of pages and the one-liners of Codd's day have given way to hundred-liners that solve extremely complex problems. In the example that follows, we demonstrate several powerful features of SQL, including common table expressions, scalar subqueries, pivoting, recursive common table expressions, outer joins, and analytic functions. Our assignment is to create a database load profile in time series format. A typical STATSPACK report only lists a point-in-time snapshot of such a database load profile, and it would therefore be useful to review the history of each component, such as logical reads or physical reads.

Load Profile	Per Second	Per Transaction
Redo size	901,736.28	20,043.04
Logical reads	247,983.21	5,511.96
Block changes	5,178.77	115.11
Physical reads	2,282.71	50.74
Physical writes	732.79	16.29
User calls	2,491.71	55.38
Parses	2,255.89	50.14
Hard parses	26.79	0.60
Sorts	1,759.71	39.11
Logons	3.32	0.07
Executes	12,469.57	277.16
Transactions	4.99	

The data for the problem is available in the stats$snapshot table (snap_id, snap_time, startup_time) and the stats$sysstat table (snap_id, name, value), both of which are part of the STATSPACK schema. The data values in stats$sysstat increase monotonically for the life of the database and start again from zero every time the database is restarted. Here is a sample of some raw data.

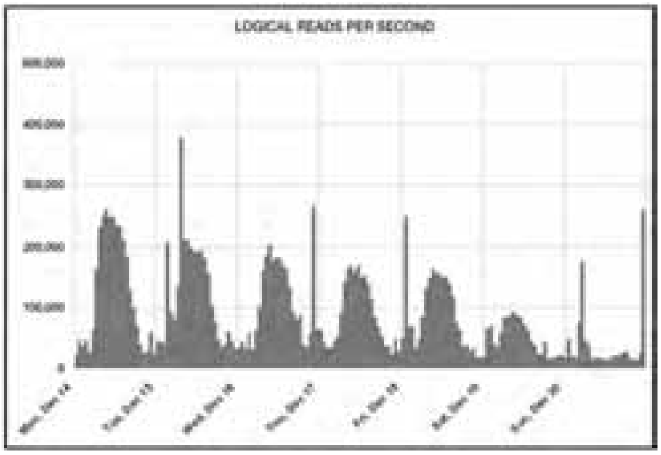
Snap Id	Snap Time	Startup Time
41566	12/14/2009 08:53	11/3/2009 22:48
41576	12/14/2009 09:23	11/3/2009 22:48
41586	12/14/2009 09:53	11/3/2009 22:48

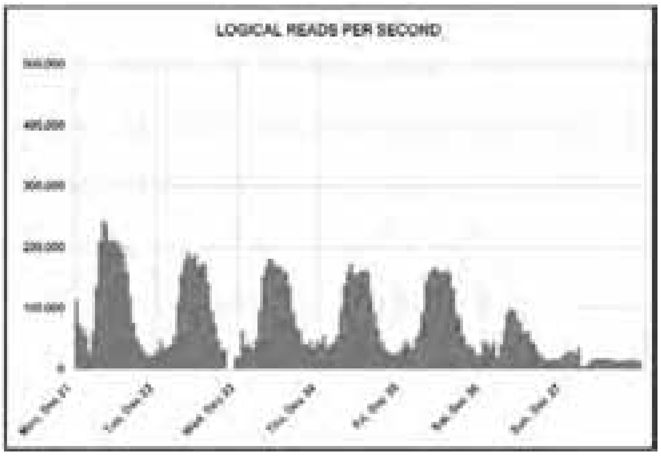
Snap Id	Name	Value
41566	session logical reads	4,272,750,011
41576	session logical reads	4,711,935,207
41586	session logical reads	5,158,057,003
41566	physical reads	57,151,567
41576	physical reads	61,038,074
41586	physical reads	65,144,663
41566	user commits	1,006,064
41576	user commits	1,082,359
41586	user commits	1,159,213
41566	user rollbacks	54,483
41576	user rollbacks	58,524
41586	user rollbacks	62,607

What we now need is the *time series representation* of the above data. Here is an example showing just a few of the columns. We used interpolation techniques to produce exactly one data point per hour. Also, we properly handled situations in which the database was restarted and the values in the database load profile were reset to zero.

Timestamp	Logical Reads Per Second	Physical Reads Per Second	Transactions Per Second
12/14/2009 09:00	258,450.28	2,047.59	46.68
12/14/2009 10:00	247,301.23	2,254.32	45.07
12/14/2009 11:00	246,408.77	2,293.54	45.35

From the time series data above, we can then generate nice graphs using our charting tool of choice such as Excel or RRDtool. Here are a couple of examples:





The first nifty SQL feature displayed in our solution is the "common table expression" (CTE). A CTE is similar to an "inline view" but offers several advantages. First, it divides the code into manageable pieces; long SQL statements that don't use common table expressions are very difficult to debug and maintain. There is also a performance consideration; the data of the CTE can be used multiple times within a SQL statement, but Oracle will not need to repeatedly recompute the CTE. Here is a list of the common table expressions in our solution; their names indicate their purpose and help the reader understand how the query progresses toward the final result.

- constants
- snaphots
- sysstat
- pivoted_sysstat
- numbers
- interpolation_formulas
- interpolated_sysstat

The other nifty feature seen below is "scalar subqueries"; a SQL query returning a single value can be placed wherever a "scalar"—a single value—is required.

```
WITH
constants AS
(
 SELECT
   (
     SELECT min(snap_id)
     FROM stats$snapshot
     WHERE snap_time >= trunc(to_date('&&begin_date')) - 1
   )
     AS begin_snap_id,

   (
     SELECT max(snap_id)
     FROM stats$snapshot
     WHERE snap_time <= trunc(to_date('&&begin_date')) + 8
   )
     AS end_snap_id
 FROM dual
),
```

Here are the results of the above code segment:

Begin Snap Id	End Snap Id
41323	44337

We next identify a subset of data from the stats$snapshot table. Scalar subqueries are once again on display. Notice how this CTE refers to the previous one.

```
snapshots AS

(

  SELECT
    snap_id,
    snap_time,
    startup_time
  FROM stats$snapshot

  WHERE dbid = &&dbid
  AND instance_number = &&instance_number
  AND snap_id >= (SELECT begin_snap_id FROM constants)
  AND snap_id <= (SELECT end_snap_id FROM constants)

),
```

Here are the results—with some rows omitted—of the above code segment:

Snap ID	Snap Time	Startup Time
41566	12/14/2009 08:53	11/3/2009 22:48
41576	12/14/2009 09:23	11/3/2009 22:48
41586	12/14/2009 09:53	11/3/2009 22:48

We next identify a subset of data from the stats$sysstat table. Scalar subqueries are once again on display.

```
sysstat AS

(

  SELECT
    snap_id,
    name,
    value AS value
  FROM stats$sysstat

  WHERE dbid = &&dbid
  AND instance_number = &&instance_number
  AND snap_id >= (SELECT begin_snap_id FROM constants)
  AND snap_id <= (SELECT end_snap_id FROM constants)
  AND name IN
  (
    'session logical reads',
    'physical reads',
    'user rollbacks',
    'user commits'
  )

),
```

Here are the results—with some rows omitted—of the above code segment:

Snap Id	Name	Value
41566	physical reads	57,151,567
41566	session logical reads	4,272,750,011
41566	user commits	1,006,064
41566	user rollbacks	54,483
41576	physical reads	61,038,074
41576	session logical reads	4,711,935,207
41576	user commits	1,082,359
41576	user rollbacks	58,524
41586	physical reads	65,144,663
41586	session logical reads	5,158,057,003
41586	user commits	1,159,213
41586	user rollbacks	62,607

The next SQL feature on display is the PIVOT operator, which was introduced in Oracle Database 11*g*R1. Pivoting is well known to Excel power users; it converts rows of data into a two-dimensional matrix. More information on the PIVOT operator and its sister operator, UNPIVOT, can be found in Arup Nanda's article *Oracle Database 11g: The Top New Features for DBAs and Developers*.

```
pivoted_sysstat AS

(

  SELECT
    snap_id,
    logical_reads AS logical_reads,
    physical_reads AS physical_reads,
    user_rollbacks + user_commits AS transactions
  FROM
    (
      SELECT *
      FROM sysstat
    )
    PIVOT
    (
      SUM(value)
      FOR NAME IN
      (
        'session logical reads' AS logical_reads,
        'physical reads' AS physical_reads,
        'user rollbacks' AS user_rollbacks,
        'user commits' AS user_commits
      )
    )

),
```

Here are the results—with some rows omitted—of the above code segment:

Snap ID	Logical Reads	Physical Reads	Transactions
41566	4,272,750,011	57,151,567	1,060,547
41576	4,711,935,207	61,038,074	1,140,883
41586	5,158,057,003	65,144,663	1,221,820

Here is the Oracle Database 10*g* version of the above code. It requires a very unintuitive use of the DECODE function.

```
pivoted_sysstat AS

(

  SELECT

    snap_id,

    sum(decode(name,'session logical reads',value,0))
      AS logical_reads,

    sum(decode(name,'physical reads',value,0))
      AS physical_reads,

    sum(decode(name,'user rollbacks',value,'user commits',value,0))
    AS transactions

  FROM sysstat
  GROUP BY snap_id

),
```

Next we have a simple example of a "recursive common table expression." Oracle has provided recursive functionality using CONNECT BY for a long time, but recursive common table expressions are new in Oracle Database 11*g*R2 and can handle problems that CONNECT BY could not. A recursive CTE takes the form of one or more "anchor members" plus a recursive member that invokes the CTE repeatedly until a terminating condition is encountered. A good explanation of recursive common table expressions can be found in an article by Jonathan Gennick titled *Understanding the WITH Clause*.

```
numbers(n) AS

(
  SELECT 1
  FROM dual

  UNION ALL

  SELECT n + 1
  FROM numbers
  WHERE n < 168
),
```

Here are the results—with some rows omitted—of the above code segment:

N
1
2
3
4
5

For contrast, here is the Oracle Database 10*g* version using the CONNECT BY clause.

```
numbers AS

(
  SELECT level AS n
  FROM dual
  CONNECT BY level <= 169
),
```

The next code section is fairly long and illustrates three interesting features. The CASE expression is a great advancement over the DECODE function and allows the evaluation of complex Boolean expressions. "Windowing functions"—a subclass of analytic functions—allow the evaluation of data contained in specified rows other than the current row. Finally, SQL now offers full support for outer joins of all flavors, including LEFT OUTER JOIN, RIGHT OUTER JOIN, and FULL OUTER JOIN; left outer joins are used below.

```
interpolation_formulas AS

(

  SELECT

    temp.timestamp,

    CASE
      WHEN s1.startup_time = s2.startup_time
      THEN s1.startup_time
      ELSE NULL
    END AS startup_time,

    CASE
      WHEN s1.startup_time = s2.startup_time
      THEN s2.snap_id
      ELSE NULL
    END AS next_snap_id,

    CASE
      WHEN s1.startup_time = s2.startup_time
      THEN (timestamp - s1.snap_time) / (s2.snap_time - s1.snap_time)
      ELSE NULL
    END AS fraction

  FROM

    (

      SELECT

        snap_id,
        snap_time AS timestamp,
        LAST_VALUE(snap_id IGNORE NULLS)
          OVER (ORDER BY snap_time ROWS BETWEEN UNBOUNDED
PRECEDING AND 1 PRECEDING)
          AS previous_snap_id,
        FIRST_VALUE(snap_id IGNORE NULLS)
          OVER (ORDER BY snap_time ROWS BETWEEN 1 FOLLOWING
AND UNBOUNDED FOLLOWING)
          AS next_snap_id

      FROM

        (
          SELECT snap_id, snap_time FROM snapshots
          UNION ALL
          SELECT NULL, trunc(to_date('&&begin_date')) + (n - 2) * 1/24 FROM
numbers
        )

    ) temp

    LEFT OUTER JOIN snapshots s1
    ON (temp.previous_snap_id = s1.snap_id)

    LEFT OUTER JOIN snapshots s2
    ON (temp.next_snap_id = S2.snap_id)

  WHERE temp.snap_id IS NULL

),
```

Here are the results—with some rows omitted—of the above code segment:

Timestamp	Startup Time	Previous Snap ID	Next Snap ID	Fraction
12/14/2009 09:00	11/3/2009 22:48	41566	41576	0.230940456
12/14/2009 10:00	11/3/2009 22:48	41586	41596	0.232648529
12/14/2009 11:00	11/3/2009 22:48	41606	41616	0.232686981

We're now ready to perform some interpolation magic. LEFT OUTER JOIN is once again on display in the following code section.

```
interpolated_sysstat AS

(

  SELECT

    timestamp,

    startup_time,

      ps1.logical_reads
    + if.fraction * (ps2.logical_reads - ps1.logical_reads)
      AS logical_reads,

      ps1.physical_reads
    + if.fraction * (ps2.physical_reads - ps1.physical_reads)
      AS physical_reads,

      ps1.transactions
    + if.fraction * (ps2.transactions - ps1.transactions)
      AS transactions

  FROM

    interpolation_formulas if

    LEFT OUTER JOIN pivoted_sysstat ps1
    ON (if.previous_snap_id = ps1.snap_id)

    LEFT OUTER JOIN pivoted_sysstat ps2
    ON (if.next_snap_id = ps2.snap_id)

),
```

Here are the results—with some rows omitted—of the above code segment:

Timestamp	Startup Time	Logical Reads	Physical Reads	Transactions
12/14/2009 09:00	11/3/2009 22:48	4,374,175,640.57	58,049,118.70	1,079,099.83
12/14/2009 10:00	11/3/2009 22:48	5,264,460,051.52	66,164,677.67	1,241,347.82
12/14/2009 11:00	11/3/2009 22:48	6,151,531,630.47	74,421,410.14	1,404,615.90

The next section uses the LAG analytic function to compute the difference between data values in adjacent rows.

```
delta_sysstat AS

(

  SELECT

    timestamp,

    logical_reads - lag(logical_reads)
      OVER (PARTITION BY startup_time ORDER BY timestamp)
      AS logical_reads,

    physical_reads - lag(physical_reads)
      OVER (PARTITION BY startup_time ORDER BY timestamp)
      AS physical_reads,

    transactions - lag(transactions)
      OVER (PARTITION BY startup_time ORDER BY timestamp)
      AS transactions

  FROM interpolated_sysstat

)
```

Here are the results—with some rows omitted—of the above code segment:

Timestamp	Logical Reads	Physical Reads	Transactions
12/14/2009 09:00	930,421,023.36	7,371,308.57	168,060.61
12/14/2009 10:00	890,284,410.95	8,115,558.97	162,247.99
12/14/2009 11:00	887,071,578.94	8,256,732.47	163,268.08

We've written a lot of code so far but it was always in unmanageable chunks. We're finally ready to display the results of the query; the final section is equally short and sweet.

```
SELECT

  timestamp,

  logical_reads / 3600
    AS logical_reads_per_second,

  physical_reads / 3600
    AS physical_reads_per_second,

  transactions / 3600
    AS transactions_per_second,

  logical_reads / transactions
    AS logical_reads_per_transaction,

  physical_reads / transactions
    AS physical_reads_per_transaction

FROM delta_sysstat
WHERE timestamp >= trunc(to_date('&&begin_date'))
ORDER BY timestamp;
```

Here are the final results with some rows and columns omitted:

Timestamp	Logical Reads Per Second	Physical Reads Per Second	Transactions Per Second
12/14/2009 09:00	258,450.28	2,047.59	46.68
12/14/2009 10:00	247,301.23	2,254.32	45.07
12/14/2009 11:00	246,408.77	2,293.54	45.35

I hope you enjoyed this little tour-by-example of the newer features of SQL. I didn't have enough space or time to do them any real justice, but you can easily find more information about them on the Internet. You can download all the code in this article from my blog. ▲

Iggy Fernandez is an Oracle DBA with Database Specialists and has more than ten years of experience in Oracle database administration. He is the editor of the quarterly journal of the Northern California Oracle Users Group (NoCOUG) and the author of Beginning Oracle Database 11g Administration (Apress, 2009). He blogs at **iggyfernandez.wordpress.com**.