Testing, testing, testing ...

Jonathan Lewis
www.jlcomp.demon.co.uk
jonathanlewis.wordpress.com

Who am I ?

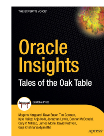
Independent Consultant.

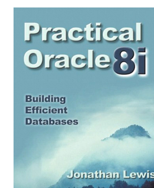
23+ years in IT
20+ using Oracle

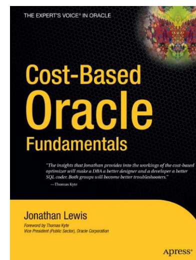
Strategy, Design, Review
Briefings, Seminars
Trouble-shooting

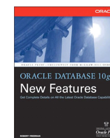
www.jlcomp.demon.co.uk
jonathanlewis.wordpress.com

One of the directors of the UKOUG
Member of the Oak Table Network.
Oracle Author of the year 2006
“Select” Editor’s choice 2007









Highlights

Why do we test

Three areas for testing

Principles of testing

Traps

Why do we test

We invest to avoid waste

- We aim to eliminate threats as early as possible in the development
- When things do go wrong, we reduce the time it takes to fix them.

Common reasons for testing

Diagnosing possible bugs

Confirming new code ***will*** work

Predicting performance threats

Common features of testing

Build the simplest viable model

- Incremental complexity minimises effort

Imagine the boundaries

- But this means you have to know the technology

Try to break it

- Proving it won't work is a lot easer than proving it will

Case Study (Debug – 1)

A batch job updates 830,000 rows in a two-column table, copying the data from column 1 to column 2.

Pre-production (9.2.0.8, 16KB blocksize, 64-bit RHEL 4) takes 30 ***minutes***.

Development (9.2.0.8, 4KB blocksize, 32-bit RHEL 4) takes 10 *seconds*.

The investigation included copying the pre-production data into new tablespaces. Tests using the 16KB block size were always slow, tests using the 4KB block size were always much faster.

`http://www.oraclealchemist.com/oracle/hey-guys-does-size-matter/`

Case Study (Debug – 2)

A 10046 trace of the final test produced the following stats:

```
EXEC #1: (16K block size)
c=1822034009,e=1779788042,p=768,cr=1541885,cu=446195350,mis=0,r=829484

EXEC #1:(4K block size)
c=   8924643,e=  10332483,p=  0,cr=  12681,cu=  2219343,mis=0,r=829484
```

16K block:	current gets = 446,000,000	CPU = 1,822.0 seconds
4K block:	current gets = 2,200,000	CPU = 8.9 seconds

(Note: without indexes, 830,000 updates should give about 830,000 CU gets)

Case Study (Debug – 3)

Let's try to model the problem from the description so far

```
create table t1
-- tablespace test_4k
-- tablespace test_16k
as
select
        trunc(dbms_random.value(10000000,100000000)) n1,
        trunc(dbms_random.value(10000000,100000000)) n2
from
        dual
connect by
        level <= 830000
;

update t1 set n2 = n1;
```

```
No difference in performance,
Small differences in the stats
```


Case Study (Debug – 4)

16KB blocks: Time:- 23 seconds

Name	Value
session logical reads	851,198
CPU used when call started	685
CPU used by this session	685
db block gets	849,964
consistent gets	1,234

4KB blocks: Time:- 23 seconds

Name	Value
session logical reads	856,931
CPU used when call started	747
CPU used by this session	747
db block gets	849,964
consistent gets	6,967

(The "db block gets" results are consistent with the expected ca, 830,000

Case Study (Debug – 5)

What if …

I make the tablespaces ASSM	-	no differences
I index the updated column	-	no differences
I “exercise” the data first	-	no differences

The numbers for current gets were always as expected.

There is ***no reasonable cause for*** the excess 444,000,000 CU gets to appear – so where is the model wrong ?

Case Study (Debug – 6)

The SQL updates 830,000 rows in a two-column table, copying column 1 to column 2.

What if column 2 is initially null ?

```
create table t1
tablespace test_16k_assm
as
select
        trunc(dbms_random.value(10000000,100000000)) n1,
        to_number(null)                               n2
from
        dual
connect by
        level <= 830000
;

update t1 set n2 = n1;
```

Case Study (Debug – 7)

```
16KB blocks - ASSM:    Time:-  5805 seconds
Name                                                  Value
session logical reads                           846,972,182
CPU used when call started                          579,244
CPU used by this session                            579,244
db block gets                                   845,084,110
consistent gets                                   1,888,072

4KB blocks - ASSM:     Time:-  89 seconds
Name                                                  Value
session logical reads                             6,698,517
CPU used when call started                            3,602
CPU used by this session                              3,602
db block gets                                     5,547,182
consistent gets                                   1,151,335
```

(The logical I/O was unreasonably high in tests with ASSM – especially when I used the 16KB block size.

Case Study (Debug – 8)

We seem to have emulated the problem under ASSM.

Moving to a 4K block appears to improve performance.

Is it the correct fix ?

Can we work out why this effect appears ?

Why are the current gets still too high in 4K ?

(5.5M instead of about 830,000)

Case Study (Debug – 9)

What is the real impact of this update ?

```
trunc(dbms_random.value(10000000,100000000))   n1,
to_number(null)                                n2

update t1 set n2 = n1;
```

The row size “doubles”

Case Study (Debug – 10)

Set pctfree to 50

```
16KB Blocks:   Time:-  22 seconds
Name                                          Value
session logical reads                       849,704
CPU used when call started                      819
CPU used by this session                        819
db block gets                               848,345
consistent gets                               1,359

4KB Blocks:    Time:-  22 seconds
Name                                          Value
session logical reads                       856,032
CPU used when call started                      801
CPU used by this session                        801
db block gets                               851,167
consistent gets                               4,865
```

Case Study (Debug – 11)

What ***were*** those block gets ? Mostly "free space search" for row migration

```
16KB Blocks - ASSM:    Time:-  5805 seconds
          Why0     Why1    Why2     Other Wait
   144,587,672        0       0              0 ktspfwh10: ktspscan_bmb
   696,965,277        0       0              0 ktspbwh1: ktspfsrch
       830,778        0       0              0 kduwh01: kdusru

4KB Blocks - ASSM:     Time:-  89 seconds
          Why0     Why1     Why2     Other Wait
     1,321,618        0        0              0 ktspfwh10: ktspscan_bmb
       680,379        0        0              0 ktspfwh12:
       660,257        0        0              0 ktspswh12: ktspffc
       660,257        0        0              0 ktsphwh39: ktspisc
       668,945        0        0              0 ktspbwh1: ktspfsrch
       481,868        0        0              0 ktuwh05: ktugct
       660,257        0        0              0 kdtwh00:
       830,629        0        0              0 kduwh01: kdusru
       660,257        0        0              0 kduwh07: kdumrp
```


Case Study (Debug – 12)

The difference in workload HAD to be a "side-effect".

A simple model showed a dramatic performance difference

Knowledge of the technology pinpoints the bug

Know what you need to measure (x$kcbsw)

Comparison of measurements highlights the error

The more you test, the faster you can design tests.

If you create a test, document it and keep it

New Code – 1

- Range Partitioning is terrific but ...
  - I want to add a partition every hour.
  - I want to keep one year's worth of data
  - I need 4 indexed access paths
- What happens when you -
  - add a partition to 8,760
  - drop 24 partitions out of 8,760
    - which is actually 120 out of 43,800 (table + 4 indexes)

New Code – 2

```
create table pt_big (
   n1   number,
   n2   number,
   n3   number,
   n4   number,
   n5   number,
   v1   varchar2(10)
)
partition by range(n1)     (
   partition p0 values less than (0)
);


create index pb_2 on pt_big(n2) local;
create index pb_3 on pt_big(n3) local;
create index pb_4 on pt_big(n4) local;
create index pb_5 on pt_big(n5) local;
```

New Code – 3

```
declare
  m_ts       timestamp := systimestamp;
begin
  for i in 1..8760 loop
      execute immediate
             'alter table pt_big add partition p' ||
             to_number(i,'FM9999') ||
             ' values less than(' ||
             to_number(i,'FM9999') ||
             ')'
      ;

      dbms_output.put_line(systimestamp - m_ts);
      m_ts := systimestamp;

  end loop;
end;
/
```


New Code – 4

```
select
      obj#, part#
from
      tabpart$
where
      bo# = {table object id};
```

Results 9.2.0.8

OBJ#	PART#
48008	1
48011	2
48013	3
48015	4
...	

Results 10.2.0.3

OBJ#	PART#
54815	10
54818	20
54820	30
54822	40
...	

New Code – 5

dba_tab_partitions references:

```
        9i    tables:       tabpart$,  tabcompart$
       10g    views:        tabpartv$, tabcompartv$
```

View tabpartv$

```
select ...
       row_number() over (partition by bo# order by part#),
       ...
from   tabpart$
```

View tabcompartv$

```
select ...
       row_number() over (partition by bo# order by part#),
       ...
from   tabcompart$
```


New Code – 6

Ideas to investigate – dealing with dropping partitions.

How about a ‘recent’ and ‘history’ table.

Exchange partition out of ‘recent’, then add, then exchange partition into ‘history’ - maybe once per day

Create a UNION ALL view of the two tables ?

Maybe build history as daily partitions by insert/append

Add date predicates to view to hide ‘overlap’ data.

Could query rewrite do anything clever ? *(probably not)*

Performance – 1

`From a recent email`

I want to test the effect of ***disk_async_io*** and ***filesystemio_options***

I did some testing in which I update 4 separate tables in 4 sessions and 1 million updates per session . But I can see no significant difference in the elapsed time for the combinations (***true*** and ***none***, ***true*** and ***setall*** etc. all possible combinations)

A bulk update on the other hand shows significant differences in elapsed times of the combinations.

Performance – 2

Disk I/O “tricks” are likely to be of some benefit during peak loading – async i/o, for example, typically flattens out peaks. This is why there was some effect during the bulk loading.

How do you emulate a heavily loaded OLTP system though ?

Maximise random I/O – reads and writes

Find a way to scale up concurrency

Performance – 3

I would create a very large table (say 25M rows) with at least four indexes on it. The indexes could be numeric columns with randomly integer values at about 10 rows per value – and one primary key.

Update the table randomly, frequently, and concurrently.

Run at least 20 concurrent processes which do something like:

- Pick a row at random by key
- Update all four indexed columns
- commit
- sleep for 2/100 second
- repeat 100,000 times (ca. 2,000 seconds)

Performance – 4

```
execute dbms_random.seed(0)
create table t1 nologging pctfree 90 pctused 10
as
with generator as (
        select  --+ materialize
                rownum  id
        from    all_objects
        where   rownum <= 5000         -- 5K * 5K = 25M
)
select
        rownum                              id,
        trunc(dbms_random.value(1,2500000)) n1,     -- 2.5M
        ...
        trunc(dbms_random.value(1,2500000)) n4,
        lpad(rownum,50,'0')                 vc1
from    generator      v1,     generator    v2
where   rownum <= 25000000                          -- 25M
;
```

Performance – 5

```
alter table t1 add constraint t1_pk primary key (id);

create index t1_n1 on t1(n1);
create index t1_n2 on t1(n2);
create index t1_n3 on t1(n3);
create index t1_n4 on t1(n4);

begin
   dbms_stats.gather_table_stats(
        ownname               => user,
        tabname               =>'T1',
        estimate_percent      => 1,
        block_sample          => true,
        method_opt            => 'for all columns size 1',
        cascade               => true
   );
end;
/
```


Performance – 6

Easy Synchronisation -- ***controller***

```
variable       g_lock_handle  varchar2(32)

execute dbms_lock.allocate_unique(                            -
               lockname              => 'External id1',      -
               lockhandle            => :g_lock_handle,      -
               expiration_secs       => 120                   -
)

execute dbms_output.put_line(                                 -
           dbms_lock.request(                                 -
               lockhandle            => :g_lock_handle,      -
               lockmode              => dbms_lock.X_mode,    -
               timeout               => 60,                   -
               release_on_commit     => TRUE                  -
           )                                                  -
)
```

Performance – 7

Easy Synchronisation -- ***worker***

```
variable        g_lock_handle  varchar2(32)

execute dbms_lock.allocate_unique(                          -
               lockname              => 'External id1',     -
               lockhandle            => :g_lock_handle,     -
               expiration_secs       => 120                 -
)

execute dbms_output.put_line(                               -
            dbms_lock.request(                              -
               lockhandle            => :g_lock_handle,     -
               lockmode              => dbms_lock.S_mode,   -
               timeout               => 60,                 -
               release_on_commit     => TRUE                -
            )                                               -
)
```


Performance – 8

Worker code:

```
execute dbms_random.seed( &1 )
begin
   for i in 1..100000 loop
        update t1
        set
                n1 = trunc(dbms_random.value(1,2500000)),
                n2 = trunc(dbms_random.value(1,2500000)),
                n3 = trunc(dbms_random.value(1,2500000)),
                n4 = trunc(dbms_random.value(1,2500000))
        where
                id = trunc(dbms_random.value(1, 25000000))
        ;
        commit write immediate wait;
        dbms_lock.sleep(0.02);
   end loop;
end;
```

Performance – 9

Running:

Session 0	start controller_lock_code

Acquires exclusive lock on user-defined lock

Session 1	start worker_code 1
Session 2	start worker_code 2
...	
Session N	start worker_code N

Sessions 1..N are waiting on session 0

Session 0	commit;

Releases exclusive lock
Session 1..N acquire shared lock and start running simultaneously

Performance – 10

What do you want to check ?

Session workload	v$sessstat
Session time lost	v$session_event
File I/O	v$filestat / v$tempstat
Latch contention	v$latch
Data contention	v$segstat
Buffer contention	v$buffer_pool_statistics
ASM issues	v$asm_disk_stat
O/S issues	See relevant o/s tools

Approaches (a)

"Empirical"

Create a sufficiently realistic model and exercise it to see if it breaks – you may get lucky, you may get unlucky

"Analytical"

Examine the actions of a simple model and predict the breaking point. Then build a complex model to test the prediction (*if necessary*).

Approaches (b)

Proof of Concept:

I want one table (which has to be an IOT)

8 Concurrent loading processes

One partition per process avoids contention

A process splits "its" partition every 15 minutes

It's a novel strategy - will it work ?

Approaches (c)

"Empirical"

Create the table

Snapshot wait events and workload statistics

Run eight copies of pl/sql to do:

```
insert 10 rows into my partition;
commit;
split my partition;
```

See what happens

© 2008

Approaches (d)

"Analytical"

Enable SQL trace to check dictionary activity

Use DDL triggers to check library cache effects

Run once through the cycle

```
insert 10 rows into one partition;
commit;
split the partition;
```

Look at the results and predict the problems.

© 2008

Approaches (e)

Early in 8i concurrent splits of partitioned IOTs could deadlock (ORA-04020) due to a defect in Oracle's internal code.

"Empirical" testing

The stress test ***might*** hit the critical concurrency condition.

"Analytical" testing

The threat ***is*** visible in the library cache locking sequence

Summary

Why are you testing

What are you going to model ?

What is a positive result, what is negative ?

Degree of realism (sanity check)