TECHNICAL NOTE

FACTORS AFFECTING CONCURRENT TRUNCATE DURING BATCH PROCESSES

Prepared By David Kurtz, Go-Faster Consultancy Ltd.

Technical Note

Version 1.00

Thursday 2 April 2009

(E-mail: david.kurtz@go-faster.co.uk, telephone +44-7771-760660)

File: local_write_wait.doc, 2 April 2009

Contents

Introduction

Over the past year, I have seen problems with Local Write Wait[1] in the Oracle database on two different Oracle systems. One occasion was in a PeopleSoft Time and Labour batch process, the other was in a custom PL/SQL process in non-PeopleSoft system.

In both cases, normal tables in the databases were being used for temporary working storage before that data was then written to another table. The content of the working storage tables was then cleared out by periodically truncating them. In order to increase overall batch throughput, several instances of the program were run in parallel. The resulting concurrent truncate operations contended with each other, and the processes did not scale well.

I have written about this subject previously in my blog[2]. These problems have prompted me to do some research and testing. I am now able to make definite recommendations.

Oracle Note 334822.1 (which I have also quoted before) provides a good technical description of the database's internal behaviour. It warns that *'processes that involve temporary tables being truncated and repopulated in multiple, concurrent batch streams may present this situation. The underlying problem is [that the Oracle database has] to write the object's dirty buffers to disk prior to actually truncating or dropping the object. This ensures instance recoverability and avoids a stuck recovery. It seems at first glance perfectly reasonable to simply truncate a temporary table, then repopulate for another usage. And then to do the temporary populate/truncate operations in concurrent batches to increase throughput.*

'However, in reality the concurrent truncates get bogged down as the database write process gets busy flushing those dirty block buffers from the buffer cache . You will see huge CI enqueue waits.

'The foreground process first acquires the RO enqueue in exclusive mode so that an object can be flushed out of buffer cache. Then the CI enqueue is held so that cross instance calls (CIC) can be issued to background processes. The CKPT process executes the CIC by scanning the whole buffer cache for the candidate blocks and moves the dirty blocks to a special list so that the DBWR [database writer] processes can write them out. This CIC completes after all the blocks have been either written out or invalidated. The RO enqueue is then released by the foreground so that another session can proceed with its drop or truncate operation'.

Put simply; truncate (and drop) operations serialise. Only one process can truncate at any one time. If you have multiple concurrent processes all trying to truncate their own working storage tables, you could experience performance problems. Such processes not scale well as the number of concurrent processes increases.

[1] My thanks to Jonathan Lewis (www.jlcomp.demon.co.uk, http://jonathanlewis.wordpress.com) for his assistance in understanding this issue. Any mistakes remain my own.

[2] http://blog.psftdba.com/2008/01/global-temporary-tables-and-peoplesoft.html

Real Problems

In the case of the non-PeopleSoft PL/SQL process, I was able to recreate the working storage tables as Global Temporary Tables (GTTs) that deleted the rows on commit because the process committed only when each transaction was complete. Local write wait totally disappeared in this case. Temporary objects do not need to be recovered, so this mechanism does not apply to them.

The PeopleSoft scenario involved one of the 'Time & Labor' batch processes, TL_TIMEADMIN. However, GTTs cannot easily be introduced into the T&L batches because there are 'restartable'. Therefore, the contents of temporary working storage tables need to be preserved after the process and its session terminates. This precludes the use of GTTs.

Below is an event profile produced from an Oracle 10g SQL Trace of an instance of TL_TIMEADMIN. Several other instances of the same program were running concurrently.

Event Name	**% Time**	**Seconds**	**Calls**	**- Time per Call -**		
				Avg	**Min**	**Max**
local write wait	19.9%	25.7509s	897	0.0287s	0.0000s	0.9845s
unaccounted-for time	19.0%	24.6180s				
EXEC calls [CPU]	17.9%	23.2000s	41,398	0.0005s	0.0000s	1.9700s
db file sequential read	13.1%	16.9244s	7,745	0.0021s	0.0001s	0.0970s
enq: RO - fast object reuse	11.2%	14.5646s	272	0.0535s	0.0000s	1.8118s
PARSE calls [CPU]	5.6%	7.2300s	15,241	0.0004s	0.0000s	0.8400s

You can see that the combination of 'Local Write Wait' and 'enq: RO - fast object reuse' account for 31% of the total response time. This is a significant proportion of the total response time.

Individual truncate commands are taking anything up to 1.8 seconds. This example from the same process is typical.

Call	Cache Misses	Count	- Seconds -		Physical Reads	- Logical Reads -		Rows
			CPU	Elapsed		Consistent	Current	
Exec	0	1	0.0300s	1.6489s	17	197	84	0

Event Name	% Time	Seconds	Calls	- Time per Call -		
				Avg	Min	Max
local write wait	53.1%	0.8684s	7	0.1241s	0.0313s	0.5529s
enq: RO - fast object reuse	42.4%	0.6938s	2	0.3469s	0.1896s	0.5041s
db file sequential read	2.6%	0.0421s	17	0.0024s	0.0002s	0.0184s
Total	**100.0%**	**1.6362s**				

- 'local write wait' occurs (as the name suggests) when the session is waiting for its own write operations. The RO enqueue is used to protect the buffer cache chain while it is scanned for dirty blocks in an object for the database writer to then write to the data files.

- 'enq: RO - fast object reuse' occurs when a process waits to acquire the RO enqueue, in other words, while somebody else is truncating or dropping an object.

Two factors affect the time for which the RO enqueue is held:

i. The time taken to write the blocks to disk. Processes that are frequently truncating temporary working storage are also doing a lot of DML operations to populate the working storage and other tables. The disks under the data files are going to be busy. If the disk becomes a bottleneck, the duration of the local write wait will certainly increase.

ii. The time taken to scan the buffer cache for dirty blocks to be written to disk and flushed from cache. The larger the buffer cache, the longer it will take to find these blocks.

The Metalink note also suggests using a different block size, saying that "a separate buffer pool for temporary tables will also reduce RO enqueue". It is not clear whether it is more important to have a different block size or a separate buffer pool. I wanted to find out which factor was more important.

Test

I created a simple test to model the behaviour of T&L. I created pairs of simple tables, populated one of each pair, and then repeatedly copied the data back and forth between them, truncating the source after the copy.

```
INSERT INTO lwr12 SELECT * FROM lwr11;
TRUNCATE TABLE lwr11;
INSERT INTO lwr11 SELECT * FROM lwr12;
TRUNCATE TABLE lwr12;
```

I wanted to test the effect of various options:

- Block Size: I created tables in tablespaces with 8Kb, 16kb, 32KB blocks. I also repeated the tests in databases with a default block size of 8kb and 16kb, but on the same physical server.
- Buffer Pool: I also created a RECYCLE pool for the default block size.
- Extent Size: I created tables with an automatically allocated extents size and with a larger uniform extent size. I only tested a 1Mb uniform extent size
- REUSE STORAGE: I tested the effect of truncating the tables with this option.
- Delete: I wanted to compare the performance of delete (which has to write undo information to the redo logs) with truncate (that will serialise on the RO enqueue and wait for local writes).
- Global Temporary Tables: I want to test the behaviour of both truncate and delete on GTTs.
- Concurrency: I ran 5 and 10 test scripts concurrently to see how truncate scales.
- Indexes: I repeated tests with and without indexes on the temporary tables.

The test script has evolved into a PL/SQL package procedure, mainly so that the tests could be submitted to and run concurrently by the Oracle job scheduler. There are also procedures to create, populate, and drop the pairs of working storage tables. It is available on my website[3].

I have run the tests on Oracle 10.2.0.3 on various platforms with similar results.

[3] The test script and package procedure can be downloaded from http://www.go-faster.co.uk/scripts.htm#lwr

Test 1

Even on a large multi-CPU server I found 5 concurrent processes was more than sufficient to the cause local write wait.

- 8Kb default block size
- 5 concurrent processes
- 100 truncates per process

	Block Size	Buffer Pool	Tablespace Type	Elapsed Duration	Total	CPU	enq: RO	Local Write Wait	Row Cache Lock	Latch Free	Others
Default	8K	DEFAULT	Autoallocate	**171**[4]	***93.37***	8.68	51.15	25.20	6.69		1.65
	8K	DEFAULT	Uniform 1M	143	98.33	4.44	61.72	29.62	0.13		2.42
	8K	RECYCLE	Autoallocate	201	105.43	9.48	59.96	27.68	6.22		2.09
	8K	RECYCLE	Uniform 1M	162	97.58	4.65	45.53	25.72	0.04	18.16[5]	3.48
	16K		Autoallocate	137	93.71	8.18	54.61	21.08	8.28		1.56
	16K		Uniform 1M	113	82.57	4.10	57.34	19.02	0.06		2.05
	32K		Autoallocate	133	74.97	7.80	38.03	21.30	6.07		1.77
	32K		Uniform 1M	**99**[6]	***61.99***	4.13	32.26	22.37	0.00		3.23
Reuse Storage	8K	DEFAULT	Autoallocate	190	98.82	3.54	59.41	32.16			3.71
	8K	DEFAULT	Uniform 1M	151	93.67	3.22	57.98	30.89			1.58
	8K	RECYCLE	Autoallocate	180	103.06	3.39	66.10	32.00			1.57
	8K	RECYCLE	Uniform 1M	143	82.76	2.96	52.28	26.25			1.27
	16K		Autoallocate	131	80.69	3.07	52.18	24.36			1.08
	16K		Uniform 1M	123	79.22	2.91	53.74	20.46			2.11
	32K		Autoallocate	128	66.77	3.14	41.67	20.58			1.38
	32K		Uniform 1M	110	66.85	3.03	39.17	23.05			1.60

4 This is the default scenario for most systems. Default buffer pool and block size. The whole test took 171 seconds, of which the truncate accounted for 93 seconds.

5 The occurrence of wait ‘latch free’ in this event is anomalous. This test is the only occurs of such a wait from several repetitions of the test.

6 This was best result. 32Kb block size, with a uniform extent size of 1M.

- The figures in bold are the effective starting point for most systems, and the pest possible outcome by using a 32Kb tablespace and a large uniform extent size – I used 1M.

- For comparison, these are the results when the rows are DELETEd instead.

	Block Size	Buffer Pool	Tablespace Type	Elapsed Duration	Total	CPU	Log Buffer Space	Log File Switch Completion	Log File Switch (Incomplete)	Free Buffer Waits	Others
Delete	8K	DEFAULT	Autoallocate	309	191.17	77.98	1.29	49.93	63.31	0.00	-1.34
	8K	DEFAULT	Uniform 1M	295	183.55	76.24	1.83	50.42	53.95	0.00	1.11
	8K	RECYCLE	Autoallocate	281	159.58	75.56	0.43	36.53	46.19	0.00	0.87
	8K	RECYCLE	Uniform 1M	271	166.21	75.72	0.98	35.13	54.65	0.00	-0.27
	16K		Autoallocate	259	186.70	71.37	5.33	45.60	47.16	4.61	12.63
	16K		Uniform 1M	258	199.20	71.90	4.06	33.54	39.27	25.93	24.50
	32K		Autoallocate	273	165.38	72.69	3.46	45.79	37.44	0.00	6.00
	32K		Uniform 1M	261	164.95	71.41	4.66	40.42	45.48	3.97	-0.99

- The Elapsed Duration is for the entire test, including the time taken to insert rows into the temporary tables.

- The negative numbers are caused by rounding errors in TKPROF

Test 2

- 8Kb default block size
- 10 concurrent processes
- 100 truncates per process

	Block Size	Buffer Pool	Tablespace Type	Elapsed Duration	Total	CPU	enq: RO	Local Write Wait	Row Cache Lock	Latch Free	Others
Default	8K	DEFAULT	Autoallocate	232	***139.86***	8.91	63.27	50.47	10.36		6.85
	8K	DEFAULT	Uniform 1M	225	178.51	4.63	92.65	75.61	0.10		5.52
	8K	RECYCLE	Autoallocate	243	174.25	8.96	101.58	48.14	12.55		3.02
	8K	RECYCLE	Uniform 1M	224	175.85	4.49	93.41	69.10	0.07		8.78
	16K		Autoallocate	198	143.81	8.27	72.62	50.58	8.10		4.24
	16K		Uniform 1M	209	165.62	4.40	95.59	59.75	0.14		5.74
	32K		Autoallocate	166	126.21	7.83	57.82	48.86	4.02		7.68
	32K		Uniform 1M	177	***137.33***	4.64	73.95	52.33	0.11		6.30
Reuse Storage	8K	DEFAULT	Autoallocate	289	175.79	4.58	99.08	68.18			3.95
	8K	DEFAULT	Uniform 1M	265	178.26	4.13	106.38	63.76			3.99
	8K	RECYCLE	Autoallocate	260	172.00	4.52	96.48	67.48			3.52
	8K	RECYCLE	Uniform 1M	306	186.69	4.39	111.04	67.25			4.01
	16K		Autoallocate	207	167.78	3.79	61.27	97.77			4.95
	16K		Uniform 1M	227	176.91	3.64	85.50	80.63			7.14
	32K		Autoallocate	171	141.89	3.40	53.42	79.59			5.48
	32K		Uniform 1M	236	152.66	3.62	76.01	67.34			5.69

	Block Size	Buffer Pool	Tablespace Type	Elapsed Duration	Total	CPU	Log Buffer Space	Log File Switch Completion	Log File Switch (Incomplete)	Free Buffer Waits	Others
Delete	8K	DEFAULT	Autoallocate	768	498.90	90.67	53.70	94.91	255.65	0.00	3.97
	8K	DEFAULT	Uniform 1M	645	447.01	89.31	54.16	111.60	193.29	0.00	-1.35
	8K	RECYCLE	Autoallocate	666	413.21	86.87	44.96	78.49	172.61	15.34	14.94
	8K	RECYCLE	Uniform 1M	675	417.96	88.44	47.75	95.84	178.48	7.86	-0.41
	16K		Autoallocate	590	436.36	82.66	52.18	97.20	133.56	70.55	0.21
	16K		Uniform 1M	657	452.25	81.50	20.38	54.89	59.97	210.24	25.27
	32K		Autoallocate	579	390.28	80.38	58.17	88.42	106.72	56.42	0.17
	32K		Uniform 1M	596	383.50	80.44	57.27	83.77	113.53	49.12	-0.63

Scalability

The following table compares the timing for 10 concurrent tests to the corresponding timings for 5. Scalability is calculated as

$$s = \frac{t_{10}}{2*t_5} - 1$$

Where:

- t_n = timing for *n* concurrent processes
- s = scalability

So, if the time for 10 concurrent processes was:

- the same as that for 5 processes, that would be 100% scalability. The number of processes has no bearing on performance.
- exactly twice that for 5 processes, that would be 0% scalability. The overall throughput does not improve as the number of processes increases
- more than twice that for 5 processes, that would be negative scalability. The overall throughput of the processes goes down as the number of processes increases

	Block Size	Buffer Pool	Tablespace Type	Elapsed Duration	Total	CPU	enq: RO	Local Write Wait	Row Cache Lock
Default	8K	DEFAULT	Autoallocate	47%	34%	95%	62%	0%	29%
	8K	DEFAULT	Uniform 1M	27%	10%	92%	33%	-22%	160%
	8K	RECYCLE	Autoallocate	65%	21%	112%	18%	15%	-1%
	8K	RECYCLE	Uniform 1M	45%	11%	107%	-3%	-26%	14%
	16K		Uniform 1M	8%	0%	86%	20%	-36%	-14%
	32K		Autoallocate	60%	19%	99%	32%	-13%	202%
	32K		Uniform 1M	12%	-10%	78%	-13%	-15%	
Reuse Storage	8K	DEFAULT	Autoallocate	31%	12%	55%	20%	-6%	
	8K	DEFAULT	Uniform 1M	14%	5%	56%	9%	-3%	
	8K	RECYCLE	Autoallocate	38%	20%	50%	37%	-5%	
	8K	RECYCLE	Uniform 1M	-7%	-11%	35%	-6%	-22%	
	32K		Autoallocate	50%	-6%	85%	56%	-48%	
	32K		Uniform 1M	-7%	-12%	67%	3%	-32%	

	Block Size	Buffer Pool	Tablespace Type	Elapsed Duration	Total
Delete	8K	DEFAULT	Autoallocate	-20%	-23%
	8K	DEFAULT	Uniform 1M	-9%	-18%
	8K	RECYCLE	Autoallocate	-16%	-23%
	8K	RECYCLE	Uniform 1M	-20%	-20%
	32K		Autoallocate	-6%	-15%
	32K		Uniform 1M	-12%	-14%

Global Temporary Tables

For the sake of completeness I have also tested the behaviour of Global Temporary Tables on a database with an 8Kb block size. The mechanism described above does not apply to temporary objects, so there is no wait at all on either *local write wait*, *RO enqueue*, or *row cache lock*. Also, DML operations on GTTs do not generate redo. Hence, with one exception, the test performs better than with permanent objects.

Elapsed Duration		Number of Concurrent Processes	
Operation	GTT Type	5	10
Truncate	PRESERVE	19	30
	DELETE	7	7
Delete	PRESERVE	154	290
	DELETE	6	4

The exception is the delete operation on tables that preserve rows on commit. In this case the delete operation generates undo and writes to the redo log files can still become a bottleneck.

General Recommendations

If you have to store temporarily working data in a database table, it is much better to use a Global Temporary Table, although the design of the application may preclude this. It is not possible to do this with data used by restartable Application Engine processes, because the contents of the GTT would be lost when the process terminates.

The Metalink note references unpublished bug 414780 in which a PeopleSoft customer reported this problem, but "they seemed to fix it by changing some PeopleSoft code to implement delete[7] rather than truncate on small temporary tables". However, my tests show that this probably degraded performance further. The individual delete statements take longer than the truncate operations, and the overall test times increased. Although the truncate operations serialise on the RO enqueue and wait for local writes, this is still better than deleting the data and waiting for the undo information to be written to the redo log. Furthermore, although the truncate operations did not scale well, the delete operations exhibited negative scalability for the same volumes and concurrency. They became bottlenecked on redo log.

Using a recycle pool of the same block size as the rest of the database was not effective; possibly because these pools use the same LRU latches.

Using a larger non-default block size improved performance of truncate, and of the overall test. The performance with 32Kb blocks was better than with 16Kb.

Using a larger uniform extent size produced the best the performance for truncate and the test as a whole. Fewer, larger extents were involved, hence less time was spent on CPU and *row cache lock*. The overall thoughput truncate operations degraded as the number of processes increased, although, the throughput of the test as whole did scale.

[7] You could do this by reducing the number of temporary table instances available on all Application Engine programs that reference the table to 0, forcing use the shared instance, and automatically changing the behaviour of the *%TruncateTable* macro to delete.

The presence or absence of indexes did not have a significant effect on the relative test timings, and does not alter my advice.

The effect of truncating with the REUSE STORAGE option is less clear cut. There are no waits on *row cache lock* because the blocks do not have to be cleared out of the buffer cache, but on the other hand more time is spent on *local write wait* because all the dirty blocks have to be written to disk, hence the RO enqueue is held for longer and more time is spent on *enq: RO - fast object reuse*. If you are using an AUTOALLOCATE tablespace then you would be better to use REUSE STORAGE option, but generally you would be slightly better to use a larger uniform extent size and not to use the REUSE STORAGE option.

PeopleSoft Recommendations

Over time, PeopleSoft batch processing has moved slightly away from SQR and COBOL. These types of process cannot be restarted, and so tables used for temporary working storage within the process can usually be recreated as Global Temporary Tables. This will produce better performance and scalability that any option that involves retaining the permanent table.

However, we are seeing more processing in PeopleSoft applications done with Application Engine. If restart has been disabled for an Application Engine program, then temporary records can also be rebuilt as Global Temporary Tables because their contents does not need to be preserved for another session to pick up.

Otherwise, move the temporary records and their indexes to tablespace with a 32Kb block size. The change of assigned tablespace can be managed within Application Designer, and released like any other patch or customisation. A 32Kb buffer cache must be created in the database instance. Sizing this is going to be a trade-off between how much memory can be taken from other activities to cache just working storage tables, and how much physical I/O you are going to have to wait for. Oracle's Automatic Shared Memory Management is of no assistance here, the KEEP, RECYCLE, and other block size buffer caches must be sized manually (see Oracle Reference Manual for SGA_TARGET).

No change to the application code is required. There is no performance improvement to be obtained by customising the application code, either to add the REUSE STORAGE option to the TRUNCATE TABLE commands, nor to use DELETE commands instead.

Oracle Bug 4224840/4260477

Unfortunately, nothing is quite as simple as it seems. If you have a transaction that locks more than 4095 rows in a 32Kb block you can encounter block corruption (this is bug 4224840). The fix/workaround in Oracle 10g (bug 4260477) is that a transaction will fail with this message before the corruption occurs[8].

```
ORA-08007: Further changes to this block by this transaction not
allowed
```

This will not be resolved until Oracle 11g, however, it does not occur with smaller block sizes.

The workaround is either to commit more frequently, or to move the table concerned back to a tablespace with a smaller block size. I have run into this with Time & Labor in a particular scenario.

[8] See http://hemantoracledba.blogspot.com/2008/08/testing-bug-4260477-fix-for-bug-4224840.html for an excellent explanation of this problem, and a test script to reproduce it.