

# GC Buffer Busy waits

## Introduction

If you have the opportunity to work in a RAC environment, you probably encountered this wait event: 'GC Buffer busy'. We will explore issues leading to excessive waits for this wait events and how to resolve the issue effectively.

## What is a GC Buffer Busy wait?

In a simple sense, GC buffer busy means that the buffer in the buffer cache, that the session is trying to access is already involved in another ongoing global cache operation. Until that global cache operation completes, session must wait. I will explain this with an example: Let's say that session #1 is trying to access a block of file #7 block ID 420. That block is in the remote cache and so, session #1 opened a BL lock on that block, requested the remote LMS process to send the block, and waiting for the block shipping to complete. Session #2 comes along shortly thereafter and tries to access the same buffer. But, the block is already involved in a global cache operation and so, session #2 must wait for the session #1 to complete GC (Global Cache) activity before proceeding. In this case, Session #2 will wait for 'gc buffer busy' wait event with a time-out and repeatedly tries to access that buffer in a loop.

Consider the scenario if the block is a hot block such as segment header, index branch block or transaction table header block etc. In this case, you can see that many such sessions waiting for the 'Gc buffer busy' wait event. This can lead to complex wait scenario quickly as few background processes also can wait for 'gc buffer busy' event leading to an eventual database hang situation. If you kill the processes, then pmon might need to access that block to do a rollback, which means that pmon can get stuck waiting for 'gc buffer busy' waits too.

## Few Scenarios

This wait event can occur for many different reasons, including bugs. For example, I encountered a bug in which the index branch block split can cause excessive 'gc buffer busy' waits. So, It is not possible to document all scenarios that can lead to gc buffer busy waits. But, it is worth exploring few most common scenarios, and then discuss a mitigation plans for those scenarios. The methods discussed here will be helpful to understand which types of blocks are involved in this issue too.

## Right Hand Growth indexes

Typically, applications generate surrogate keys using a sequence based key generation, an example would be employee\_id column in the employee table. These types of unique or primary key columns are usually populated using a sequence generated value with a unique constraint on the column. A unique index may be created to support the unique constraint. Although, it is possible to create a non-unique index to support the unique constraint, and that non-unique index also will suffer from the issues similar to its unique counterparts. This problem is related to more about uniqueness of data and locality of new rows, rather than the type of index.

Indexes store sorted (key[s], ROWID) pair, meaning values in the index are arranged in an ascending or descending key column order. ROWIDs in the (key[s], ROWID) pair points to a specific row in

the table segment with that index column value. Also, Indexes are implemented as B-Tree indexes. In the case of unique indexes, on columns populated by sequence based key values, recent entries will be in the right most leaf block of the B-Tree. All new rows will be stored in the right most leaf block of the index. As more and more sessions insert rows in to the table, that right most leaf block will be full. Oracle will split that right most leaf block in to two leaf blocks: One block with all rows except one row and a new block with just one row. (This split, aka Index 90-10 split, needs to modify branch block also ). Now that new leaf block becomes the right most leaf block and the concurrency moves to the new leaf block. Simply put, you can see concurrency issues moving from one block to another block in an orderly fashion.

As you can imagine, all sessions inserting in to the table will insert rows in to the current right most leaf block of the index. This type of index growth termed as “Right Hand Growth” Indexes. As sessions inserts in to the right most leaf block of the index, that index becomes hot block, and concurrency on that leaf block leads to performance issues.

In RAC, this problem is magnified. Sequence cache is instance specific and if the cache is small (defaults to 20), then the right most leaf block becomes hot block, not just in one instance, but across all instances. That hot – right most – leaf block will be transferred back and forth between the instances. If the block is considered busy, then LMS process might induce more delays in transferring the blocks between the instances. While the block is in transit, then the sessions accessing that block must wait on ‘GC buffer busy’ waits and this quickly leads to excessive GC buffer busy waits. Also, immediate branch block of those right most leaf block will play a role in the waits during leaf block splits.

So, how bad can it get? It can be very bad. A complete database hang situation is a possibility. Notice below that over 1000 sessions were waiting for ‘gc buffer busy’ events across the cluster. Application is completely down.

INST_ID	SQL_ID	EVENT	STATE	COUNT(*)
4	4jtbgawt37mcd	gc cr request	WAITING	9
3	4jtbgawt37mcd	gc cr request	WAITING	9
3	a1bp5ytpvfj48	gc buffer busy	WAITING	11
4	a1bp5ytpvfj48	gc buffer busy	WAITING	17
4	14t0wadn1t0us	gc buffer busy	WAITING	33
4	gt1rdqk2ub851	gc buffer busy	WAITING	34
4	a1bp5ytpvfj48	buffer busy waits	WAITING	35
2	a1bp5ytpvfj48	gc buffer busy	WAITING	65
1	a1bp5ytpvfj48	gc buffer busy	WAITING	102
2	7xzqcrdrnyw1j	gc buffer busy	WAITING	106
2	7xzqcrdrnyw1j	enq:TX - index c	WAITING	173
1	7xzqcrdrnyw1j	gc buffer busy	WAITING	198
3	7xzqcrdrnyw1j	gc buffer busy	WAITING	247
4	7xzqcrdrnyw1j	gc buffer busy	WAITING	247

### How do you analyze the problem with right hand indexes?

First, we need to verify that problem is due to right hand indexes. If you have access to ASH data, it is easy. For all sessions waiting for ‘gc buffer busy’ event query the current\_obj#. Following query on ASH can provide you with the object\_id involved in these waits. Also, make sure the statement is UPDATE or INSERT statements, not SELECT statement[ SELECT statements are discussed below].

```
select sample_time, sql_id, event, current_obj#,sum (cnt) from
gv$active_session_history
  where sample_time between to_date ('24-SEP-2010 14:28:00','DD-MON-
YYYY HH24:MI:SS') and
```

```

to_date ('24-SEP-2010 14:29:59','DD-MON-YYYY HH24:MI:SS')
group by sample_time, sql_id, event, current_obj#
order by sample_time
/

```

```

SAMPLE_TIME          |SQL_ID          |EVENT
|CURRENT_OBJ#| COUNT(*)
-----|-----|-----
...
26-AUG-10 02.28.18.052 PM          |14t0wadn1t0us          |gc buffer busy
|      8366|      33
..

```

```

select owner, object_name, object_type from dba_objects where
object_id=8366 or data_object_id=8366;

```

In this example, current\_obj# is 8366, which we can query the dba\_objects to find the correct object\_id. If this object is an unique index or almost unique index, then you might be running in to a right hand growth indexes.

If you don't have access to ASH then, you need to sample gv\$session\_wait (or gv\$session from 10g), group by p1, p2 to identify the blocks inducing 'gc buffer busy' waits. Then, map those blocks to objects suffering from the issue.

```

select event,    p1, p2, count(*) from gv$session s
where
event not like '%pipe%'
and event not like 'SQL*%'
and event not like 'Streams AQ:%'
and event ='gc buffer busy' and state='WAITING'
group by event, p1, p2
order by 4
/

```

You can also use my script [segment\\_stats\\_delta.sql](#) to see the objects suffering from 'gc buffer busy' waits. See below for an example use:

```

@segment_stats_delta.sql
segment_stats_delta.sql v1.01 by Riyaj Shamsudeen @orainternals.com

...Prints Change in segment statistics in the past N seconds.
...Default collection period is 60 seconds.... Please wait for at least
60 seconds...

```

```

!!! NOTICE !!! This scripts drops and recreates two types:
segment_stats_tbl_type and segment_stats_type.

```

Following are the available statistics:  
Pay close attention to sampled column below. Statistics which are sampled are not exact and so, it might not reflect immediately.

NAME	SAM
logical reads	YES
buffer busy waits	NO
gc buffer busy	NO
db block changes	YES
physical reads	NO
physical writes	NO

```

physical reads direct                                NO
physical writes direct                              NO
gc cr blocks received                               NO
...
Enter value for statistic_name: gc buffer busy
Enter value for sleep_duration: 60
WSH                                                  | WSH_DELIVERY_DETAILS_U1      | 34
APPLSYS                                             | WF_ITEM_ATTRIBUTE_VALUES_PK  | 2
WSH                                                  | WSH_DELIVERY_DETAILS_U1      | 2
INV                                                  | MTL_MATERIAL_TRANSACTIONS_U1 | 1
ONT                                                  | OE_ORDER_LINES_ALL           | 1

```

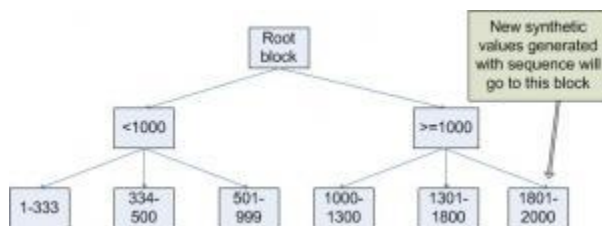
PL/SQL procedure successfully completed.

## How do you resolve Gc buffer busy waits due to right hand indexes?

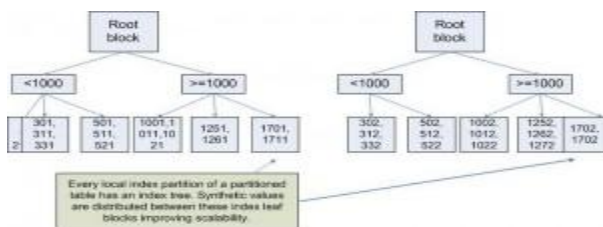
In a simplistic sense, You need to reduce the concurrency on the right most leaf block. There are few options to reduce the concurrency, and hash partitioning that unique index (or almost unique index) is a better solution of all. For example, if we convert the unique index as hash partitioned index with 32 partitions, then you are reducing the concurrency on that right most leaf block by 32 fold. Why? In hash partitioning scheme with 32 partitions, there are 32 index trees and inserts will be spread across 32 right most leaf blocks. In contrast, there is just 1 index tree in the case of non-partitioned index. Essentially, Each partition gets its own index tree. Given an ID column value, that row is always inserted in to a specific partition and that partition is identified by applying hash function over the partitioning column.

In the case of non-partitioned index, for example, values from 1000 to 1010 will be stored in the right most leaf block of the index. In the case of partitioned index with 32 partitions, value 1000 will be stored in partition 24, value 1001 will be stored in partition 19, meaning, values are hashed and spread around 32 partitions leading to improved concurrency. This will completely eliminate Right hand Growth index concurrency issue.

### Non-partitioned index



### Hash partitioned index with 2 partitions



## What is the drawback of hash partitioning an index?

If your query needs to do range scan with a predicate similar to 'id between 1000 and 1005', then the index range scan will need to scan 32 index trees, instead of one index tree as in the case of non-partitioned tables. Unless, your application executes these sort of queries millions of times, you probably wouldn't notice the performance difference. For equality predicate, such as 'id=:B1', this is not an issue as the database will need to scan just one index tree.

Let's discuss about reverse key indexes too. As Michael Hallas and Greg Rahn of real-world performance group said (and I happily concur), reverse key indexes are evil. If you are in Oracle Database version 10g, you can create a partitioned index on a *non-partitioned* table. So, If your application is suffering from a right hand growth index contention issue, you can convert the non-partitioned index to a hash-partitioned index with minimal risk. So, there aren't many reasons to use reverse key indexes in 10g [ Remember that range scan is not allowed in the reverse key indexes either]. But, if you are unfortunate enough to support Oracle 9i database, you can NOT create a partitioned index on a non-partitioned table. If your application suffers from right hand growth index concurrency issues in 9i, then your options may be limited to reverse key indexes (or playing with sequences and code change or better yet – upgrade to 10g).

## What if the statement is a SELECT statement?

It is possible for the SELECT statement to suffer from gc buffer busy waits too. If you encounter a scenario in which the object is an index and the statement is a SELECT statement, then this paragraph applies to you. This issue typically happens if there is higher concurrency on few blocks. For example, excessive index full scan on an index concurrently from many instances can cause 'gc buffer busy' waits. In this case, right approach would be tune the SQL statement to avoid excessive access to those index blocks.

In my experience, gc buffer busy waits on SELECT statement generally happens if you have problems with statistics and execution plans. So, verify that execution plan or concurrency didn't change recently.

## Freelists, Free list groups

What if the object ID we queried in Active Session History belongs to a table block and the statement is an INSERT statement? We need to check to see if that block is a segment header block. If there are many concurrent inserts, and if you don't use ASSM tablespace, then the inserts need to find free blocks. Segment header of an object stores the free list [ if you don't use freelist groups]. So,

concurrent inserts in to a table will induce excessive activity on the Segment header block of that table leading to 'gc buffer busy' waits.

Concurrent inserts in to a table with 1 freelist/1 freelist groups will also have contention in a non-segment header block too. When the session(s) searches for a free block in a freelist, all those sessions can get one or two free blocks to insert. This can lead to contention on that block.

Right approach in this case is to increase freelists, free list groups and initrans on those objects (and might need reorg for these parameters to take effect). Better yet, use ASSM tablespaces to avoid these issues.

### **Other Scenarios**

We discussed just few common issues. But, 'gc buffer busy' waits can happen for many reasons. Few of them are: CPU starvation issues, Swapping issues, interconnect issues etc. For example, if the process that opened the request for a block did not get enough CPU, then it might not drain the network buffers to copy the buffer to buffer cache. Other sessions accessing that buffer will wait on 'gc buffer busy' waits. Or If there is a network issue and the Global cache messages are slower, then it might induce higher gc buffer busy waits too. Statistics 'gc lost packets' is a good indicator for network issues, but not necessarily a complete indicator.

As a special case, if the sequences have been kept with lower cache value, then blocks belonging to seq\$ table can be an issue too. Increasing cache for highly used sequence should be considered as a mandatory task in RAC.

### **Summary**

In summary, next time you encounter this issue, drill down to see which object types and statements are involved. Debug to understand the root cause as 'gc buffer busy' waits are usually symptoms.