# How Oracle secretly changed multiblock reads

Frits Hoogland, Enkitec

## Abstract

The Oracle database uses multiblock reads as an optimized method for reading an entire segment. This method has been with the Oracle database since at least Oracle 7. Whilst this may look like a simple and easy to understand topic (the system fetches multiple blocks conforming db_file_multiblock_read_count blocks instead of one, right?), in reality it isn't. The description of the former line is mostly true for version 10 non-parallel query (PQ) multiblock reads, but with Oracle version 11 this has changed.

Oracle silently introduced true asynchronous reads with version 11, called 'adaptive direct path reads', which happen under specific circumstances. This paper outlines these circumstances. One of the most eye-catching features is reading blocks to the PGA, which makes the reads non-shared, which is different from the traditional reading into buffer cache/SGA.

## Target Audience

The contents of this paper are targeted for performance analysts and people who specialize in database internals. This paper is setup to start from the database SQL layer/SQL trace and gradually dive deeper into the inner working of the Oracle database on Linux.

## Executive Summary

The learner will be able to:
Understand how Oracle executes full scans between database version 10.2.0.1 and 11.2.0.3.
Have the ability to identify buffered full segment scans and unbuffered full segments scans.
Understand the criteria that influence an Oracle 11 database to choose between buffered and unbuffered full segment scans.
Understand how the segment extent size and administration bitmap blocks influence the number of blocks read in one go.
Understand what actually is happening when the wait events ‘db file scattered read’ and ‘direct path read’ are encountered, and how the two events differ in meaning.
Understand that Oracle keeps track of asynchronous IOs using a mechanism that is called ‘slots’.
How the number of concurrent IOs can increase.

## Background

Whenever a rowsource in an explain plan lists 'TABLE ACCESS FULL', 'FAST FULL INDEX SCAN' or 'BITMAP FULL SCAN' (among others, these are the the rowsources which seem to be the most encountered in my situation), Oracle needs to read the entire segment up to the high water mark. In order to make use of operating system and SAN/NAS readahead optimizations (among other layers which might do readahead optimization, like the Logical Volume Manager (LVM)), Oracle tries to read as much adjacent blocks as possible, restricted by the DB_FILE_MULTIBLOCK_READ_COUNT parameter. This optimization also prevents the database from doing multiple repetitive IOs (where each IO introduces a latency penalty). Most operating systems can read up to 1 megabyte in a single IO.

When a segment is scanned using a full segment scan, this is how most people assume this is processed:
(This example is done using an Oracle 10.2.0.1 database on X86_64 Linux)

```
TS@v10201 > select count(*) from t2;

  COUNT(*)
----------
   1000000

Execution Plan
----------------------------------------------------------
Plan hash value: 3724264953

-------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |  3674   (1)| 00:00:45 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T2   |  1007K|  3674   (1)| 00:00:45 |
-------------------------------------------------------------------------
Statistics
----------------------------------------------------------
        212  recursive calls
          0  db block gets
      20976  consistent gets
      20942  physical reads
          0  redo size
        515  bytes sent via SQL*Net to client
        469  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          4  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

In order to find out how much of the blocks are in the cache, the X$ view X$KCBOQH (Kernel Cache Buffers Object Queue Headers seems to be a reasonable guess) can be used.

This requires the object id or “obj#”, which can be found in the following way:

```
SYS@v10201 AS SYSDBA> select object_id, object_name, owner from dba_objects where
object_name = 'T2';

OBJECT_ID OBJECT_NAME                                        OWNER
---------- ------------------------------------------------ ------------------
     10237 T2                                               TS

Now we query X$KCBOQH using the object id:

SYS@v10201 AS SYSDBA> select * from x$kcboqh where obj# = 10237;

ADDR                 INDX    INST_ID     TS#    OBJ#   NUM_BUF HEADER
---------------- -------- --------- ------- ------- --------- ----------------
FFFFFD7FFD5C6FA8      335          1       5   10237     20942 000000038FBCF840
```

If we look at the number of blocks read via a physical path in the statistics (20942), we see the number of blocks in the cache (NUM_BUF) to be the same. This tells us all the blocks are in the cache. If the same SQL is executed again, we take advantage of the blocks being cached in the database buffer cache:

```
TS@v10201 > select count(*) from t2;

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
      20953  consistent gets
          0  physical reads
          0  redo size
        515  bytes sent via SQL*Net to client
        469  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed

The number of physical reads is ‘0’. This is how I think most people are taught how
the database works.

Now let’s move on to an Oracle 11.2.0.3 database on the same platform (X86_64
Linux), and do exactly the same on the same objects:

TS@v11203 > select count(*) from t2;

  COUNT(*)
----------
   1000000


Execution Plan
```

```
-------------------------------------------------------
Plan hash value: 3724264953

-------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |  3672   (1)| 00:00:45 |
|   1 |  SORT AGGREGATE    |      |     1 |            |          |
|   2 |   TABLE ACCESS FULL| T2   |  1000K|  3672   (1)| 00:00:45 |
-------------------------------------------------------------------------

Statistics
-------------------------------------------------------
        217  recursive calls
          0  db block gets
      20970  consistent gets
      20942  physical reads
          0  redo size
        526  bytes sent via SQL*Net to client
        523  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          4  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

This output on Oracle version 11.2.0.3 looks the same as the output with version 10.2.0.1. In fact, these statistics cannot tell us something different has happened. However, if we take a look at X$KCBOQH again, we do see something different:

```
SYS@v11203 AS SYSDBA> select * from x$kcboqh where obj# = 66614;

ADDR                INDX  INST_ID    TS#       OBJ#    NUM_BUF HEADER
---------------- ------- -------- ------ ---------- ---------- ----------------
FFFFFD7FFC541B18      43        1      5      66614          1 000000039043E470
```

The field NUM_BUF now only lists 1. If we execute the SQL again, all but one block is read via physical path again:

```
TS@v11203 > select count(*) from t2;

Statistics
-------------------------------------------------------
          0  recursive calls
          0  db block gets
      20945  consistent gets
      20941  physical reads
          0  redo size
        526  bytes sent via SQL*Net to client
        523  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
```

```
1  rows processed
```

Clearly something different is happening in both cases. In order to gain more understanding, let's trace (sql_trace level 8) both sessions and see if something is different:

Oracle 10.2.0.1 (partial output):

```
WAIT #1: nam='db file sequential read' ela= 32941 file#=5 block#=19 blocks=1
WAIT #1: nam='db file scattered read' ela= 4003 file#=5 block#=20 blocks=5
WAIT #1: nam='db file scattered read' ela= 6048 file#=5 block#=25 blocks=8
WAIT #1: nam='db file scattered read' ela= 1155 file#=5 block#=34 blocks=7
WAIT #1: nam='db file scattered read' ela= 860 file#=5 block#=41 blocks=8
WAIT #1: nam='db file scattered read' ela= 837 file#=5 block#=50 blocks=7
```

Oracle 11.2.0.3 (partial output):

```
nam='db file sequential read' ela= 12607 file#=5 block#=43394 blocks=1
nam='direct path read' ela= 50599 file number=5 first dba=43395 block cnt=13
nam='direct path read' ela= 21483 file number=5 first dba=43425 block cnt=15
nam='direct path read' ela= 10766 file number=5 first dba=43441 block cnt=15
nam='direct path read' ela= 12915 file number=5 first dba=43457 block cnt=15
nam='direct path read' ela= 12583 file number=5 first dba=43473 block cnt=15
```

A multiblock read which is read into the buffer cache uses the 'db file scattered read' event, whilst a multiblock read which is read into the process' PGA is using the 'direct path read' event.

When My Oracle Support is searched for information, note 787373.1 'How does Oracle load data into the buffer cache for table scans.' is meant to explain this behavior[1]. However, this is not accurate. The My Oracle Support note mentions that when a segment is bigger than _SMALL_TABLE_THRESHOLD (which defaults to 2% of the buffer cache), it is read via direct path into the process' PGA, instead of the buffer cache. Several sources on the internet report that a segment needs to be bigger than 5 times _SMALL_TABLE_THRESHOLD for Oracle to switch to direct path reads. But that is not the only criterion for switching to direct path reads, others are: less than 50% of the blocks of the segment is allowed in the buffercache (the reason is it probably makes sense to read the rest in the buffercache, instead of reading all blocks from disk), and less than 25% of the segment is allowed to be dirty (not yet written to the datafile). Please mind these percentages seem to be the main criteria, there's reason to believe the Oracle engine uses additional heuristics to determine if it should do a direct path read or a buffered (regular) read for a segment.

Please mind 'runtime decision' in the above paragraph. The choice to do reads into either the buffer cache or the process' PGA is not an optimizer decision, it's decided in the code path which executes the full segment scan.

[1] After the author commented on this note in MOS, the note has disappeared from (public) access.

It is a fair guess that the ability of the database to do direct path reads for a non-parallel query database foreground process is governed by the parameter _SERIAL_DIRECT_READ, which is “FALSE” with version 10.2.0.1 and “TRUE” for Oracle 11.2.0.3.

If we take a look at how an Oracle ASSM (automatic segment space management) tablespace dictates how the extents are allocated, it looks like this:

```
SYS@v10201 AS SYSDBA> select segment_name, extent_id, block_id, blocks, bytes
from dba_extents where segment_name = 'T2' and owner = 'TS' order by extent_id;

SEGMENT_NAME                       EXTENT_ID  BLOCKS     BYTES
---------------------------------- ---------- --------- ----------
T2                                          0         8      65536
...
T2                                         15         8      65536
T2                                         16       128    1048576
...
T2                                         78       128    1048576
T2                                         79      1024    8388608
...
T2                                         91      1024    8388608
```

The first 16 extents consist of 8 blocks, the next 62 extents consist of 128 blocks, etc. If we look at how Oracle 10.2.0.1 reads these blocks (edited):

```
WAIT #2: nam='db file sequential read' ela= 12292 file#=5 block#=19 blocks=1
WAIT #2: nam='db file scattered read' ela= 179162 file#=5 block#=20 blocks=5
WAIT #2: nam='db file scattered read' ela= 47597 file#=5 block#=25 blocks=8
WAIT #2: nam='db file scattered read' ela= 5206 file#=5 block#=34 blocks=7
WAIT #2: nam='db file scattered read' ela= 94101 file#=5 block#=41 blocks=8
WAIT #2: nam='db file scattered read' ela= 512 file#=5 block#=50 blocks=7
```

First the segment header is read, which is a single block, which explains the ‘db file sequential read’ event. The segment header is the third block of the extent (the first two blocks contain the object header). So the first multiblock IO is done for only 5 blocks, which is the remainder of the first extent. The next extent is read fully (blocks=8), but the consecutive extent is read from the second block up to the extent border (blocks=7). The reason is, there is a L1 bitmap space administration block in between, which does not contain any actual data, so it is skipped during a full segment scan. When the number of blocks increases to 128, the first two blocks of every extent are L1 bitmap space management blocks, and when the extent size increases to 1024, the first four blocks of every extent are L1 bitmap space management blocks.

Now look at the waits with Oracle 11.2.0.3 (edited):

```
nam='db file sequential read' ela= 12607 file#=5 block#=43394 blocks=1
nam='direct path read' ela= 50599 file number=5 first dba=43395 block cnt=13
nam='direct path read' ela= 21483 file number=5 first dba=43425 block cnt=15
nam='direct path read' ela= 10766 file number=5 first dba=43441 block cnt=15
nam='direct path read' ela= 12915 file number=5 first dba=43457 block cnt=15
```

```
nam='direct path read' ela= 12583 file number=5 first dba=43473 block cnt=15
```

It starts off exactly the same as the previous (Oracle 10.2.0.1) scan with a single block read of the third block of the first extent. However, the first multiblock read shows a different number of blocks than the previous wait excerpt: here are 13 blocks read when the extent size is still 8 blocks. This means that Oracle can cross the extent border if doing serial direct path reads! Actually, the number of blocks read limited by the L1 bitmap space management blocks when the extent size is 8 blocks.

Another very important clue about serial direct path reads is hidden in this excerpt. I urge you to take a close look again. If you did, you could see an extent is missing from the list (first direct path read starts at dba 43395, and reads 13 blocks. 43395+13=43408, while the second direct path read is at dba 43425; this means the second extent is not listed, however, the second extent does contain data, and is actually read!). If you are very curious about this, go to the part of this whitepaper where the actual physical implementation of serial reads is explained, or even better: read on!

At this point it becomes apparent it's important to understand what actually is happening during 'db file scattered read' and 'direct path read' waits on the operating system layer.

When doing a synchronous multiblock read to the buffer cache, Oracle determines the file, offset in the file and what number of bytes to read from the offset. Oracle always reads a single continuous range of blocks from a datafile. Probably only if the datafile is on ASM, if the continuous range of the datafile is in multiple ASM allocation units, an additional, second pread64() call could be issued. Oracle reads the extents in a strict serial way.

With Oracle 10.2.0.1 doing a multiblock read via asynchronous IO, the extents are strictly read in a serial way. The second extent is only read when the first extent is fully read and processed. This makes the use of asynchronous IO behave in a synchronous way, because the essence is that as soon as one or multiple IOs are issued with the io_submit() call, the database can't do anything else than start waiting for the result. What is notable here, is the small red block following the io_submit() call. This is an io_getevents() call with the parameter timeout set to zero, which makes the call 'non-blocking'. Non-blocking means it's just a quick peek into the result queue in the operating system 'IO context'. In fact, this peek is in the io_submit() codepath. If the non-blocking io_getevents() call can reap all the results (immediately after submitting), the process does not have to go into the part of the code for specifically waiting for IOs to become available. If the non-blocking io_getevents() call cannot reap all submitted IO's, the process issues another io_getevents() call which needs to be blocking (because we need to wait for the IO to complete, there is nothing else we can do). This is done by setting a non-zero timeout value (the timeout is 600 seconds for versions 10.2 – 11.2). The blocking calls are done for a single IO, so if multiple IO's are issued with io_submit(), io_getevents() is called for the number of IOs that are submitted, given the IOs arrive one by one.

I want to stress the serial nature of the reads into the buffercache, which are noticeable by the event ‘db file scattered read’.

With version 11.2.0.3 of the database Oracle decoupled the non-blocking io_getevents() call from the code that submit’s the IO, so it’s actually in the part of the database code that reaps the IO’s. If the non-blocking io_getevents() call cannot reap all the IO’s, Oracle sets up a blocking io_getevents() call for all the IO’s that where active for this extent.

Now let’s move on to how Oracle issues calls when doing non-buffered/PGA full scans. When Oracle chooses serial direct path reads, it’s doing that in a vastly different way, of which one of the most important things is the database will try to keep multiple IOs (of multiple extents) in flight.

Because _SERIAL_DIRECT_READ is set to FALSE by default on Oracle 10.2.0.1, it’s not used in this version. This means it does not make sense to describe this.

When Oracle chooses serial direct path reads, it will start a full scan with two io_submit() calls to bring two IOs into flight. In fact, two IOs is the minimal number of IOs concurrently active for doing serial direct path reads. These are the IO requests for two different extents (!!). Please mind the io_submit() calls are outside of the wait event registration. After the IO requests are submitted, Oracle can issue up to 4 io_getevents() calls for all IO requests which are active for that process at that moment, with timeout set to zero (indicated by the small red blocks), so non-blocking. Please mind there still is no wait event involved. If one of these non-blocking io_getevents() calls are satisfied, the process will stop issuing io_getevents() calls and starts processing the results of the IOs, and there will be no wait line in the tracefile (if trace is enabled) and no wait event time registration in the database. If these non-blocking calls cannot reap all IO requests active for this process, Oracle will register a direct path read wait event, and issue a blocking (timeout set to 600) io_getevents() call for one IO (!). This means that if your IO subsystem is fast enough, you can have no, or very few, IO related wait lines in the tracefile and in the wait event views in the database.

This makes the event ‘direct path read’ differ significantly from most other IO related events. While ‘db file sequential read’ time and ‘db file scattered read’ time (when it measures the time of a single pread64() or the combination io_submit() and io_getevents() for a single IO) means IO latency and IO transfer time, which can be used to measure your IO speed, the ‘direct path read’ event has nothing to do with IO latency. The wait event ‘direct path read’ means the time waiting for IO in general from the perspective of the Oracle database process, and absolutely not IO latency time.

The number of IO requests which the database tries to keep in flight during serial direct path reads are tracked in the database in a structure that is called a ‘slot’. The usage of these slots are visible in v$sysstat and v$sesstat in a statistic called ‘total number of slots’. One of the most eye-catching properties of these statistics is that these are not cumulative, unlike most of

the other statistics in both v$sysstat and v$sesstat. These slots are used with serial direct path reads, and can be used by parallel query slaves as well.

Starting from Oracle database version 11.2, Oracle measures wall-clock time, wait time and throughput, and determines if it should allocate additional slots, which means it will bring more IO requests concurrently in flight. In fact, this mechanism is present with version 11.2.0.1, but only actually works starting from Oracle version 11.2.0.2. Oracle has a debugging event for the adaptive mechanism for scaling up the number of IO's: event 10365. In order to turn on debugging for the adaptive scaling of the number of IO's, simply set the event to level 1:

```
TS@v11203> alter session set events '10365 trace name context forever, level 1';
```

Now the tracefile can be investigated when doing serial direct path reads and the number of slots, alias the number of concurrent IO requests in flight:

```
kcbldrsini: Timestamp 61180 ms
kcbldrsini: Current idx 16
kcbldrsini: Initializing kcbldrps
kcbldrsini: Slave idx 17
kcbldrsini: Number slots 2
kcbldrsini: Number of slots per session 2

*** 2011-11-28 22:58:48.808
kcblsinc:Timing time 1693472, wait time 1291416, ratio 76 st 248752270 cur
250445744
kcblsinc: Timing curidx 17 session idx 17
kcblsinc: Timestamp 64180 ms
kcblsinc: Current idx 17
kcblsinc: Slave idx 17
kcblsinc: Number slots 2
kcblsinc: Number of slots per session 2
kcblsinc: Previous throughput 8378 state 2
kcblsinc: adaptive direct read mode 1, adaptive direct write mode 0
```

kcbldrsini() is the code that initializes the adaptive IO mechanism, kcblsinc() is the code that actually measures  wall-clock time, wait time and throughput. The tracefile output above is the initialization, and the first measurement of the variables.

Below is wat looks like when an additional slot ("slos") is allocated. The mechanism behind how Oracle determines when to add a slot is unknown to me. It also looks like it does scale the number of slots up, but doesn't scale down. I've never seen the number of slots decreasing. The slots mechanism can scale up to 32 slots (IOs in flight) for a single process.

```
kcblsinc:Timing time 2962717, wait time 2923226, ratio 98 st 253662983 cur
256625702
kcblsinc: Timing curidx 19 session idx 19
kcblsinc: Timestamp 70270 ms
kcblsinc: Current idx 19
kcblsinc: Slave idx 19
```

```
kcblsinc: Number slots 2
kcblsinc: Number of slots per session 2
kcblsinc: Previous throughput 11210 state 1
kcblsinc: adaptive direct read mode 1, adaptive direct write mode 0
kcblsinc: Adding extra slos 1
```

References


http://afatkulin.blogspot.com/2009/01/11g-adaptive-direct-path-reads-what-is.html
http://dioncho.wordpress.com/2009/07/21/disabling-direct-path-read-for-the-serial-full-table-scan-11g/
http://www.oracle.com/pls/db112/homepage
http://hoopercharles.wordpress.com/2010/04/10/auto-tuned-db_file_multiblock_read_count-parameter/
http://fritshoogland.wordpress.com/2012/04/26/getting-to-know-oracle-wait-events-in-linux/