

# IBM DB2 11 for z/OS Buffer Pool Monitoring and Tuning

Describe the functions of the DB2  
buffer pools

Check metrics for read and write

Set up and monitor the DB2  
buffer pools



Jeff Berger





International Technical Support Organization

**IBM DB2 11 for z/OS Buffer Pool Monitoring and Tuning**

July 2014

**Note:** Before using this information and the product it supports, read the information in “Notices” on page v.

### **First Edition (July 2014)**

This edition applies to Version 11, Release 1 of DB2 for z/OS (program number 5615-DB2).

This document was created or updated on July 29, 2014.

**© Copyright International Business Machines Corporation 2014. All rights reserved.**

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

# Contents

<b>Notices</b> .....	v
Trademarks .....	vi
<b>Preface</b> .....	vii
Authors .....	viii
Now you can become a published author, too! .....	viii
Comments welcome .....	viii
Stay connected to IBM Redbooks .....	ix
<b>Chapter 1. Introduction</b> .....	1
1.1 Buffer pool attributes .....	2
1.2 Buffer pool tuning .....	6
1.3 Getpage classification and buffer classification .....	8
1.4 Using VPSEQT .....	9
1.5 Utility use of most recently used .....	9
1.6 Page residency time .....	9
1.7 AUTOSIZE(YES) .....	10
1.8 Critical thresholds .....	10
<b>Chapter 2. Prefetch and RID pool</b> .....	13
2.1 Sequential prefetch .....	14
2.2 Sequential detection and dynamic prefetch .....	15
2.3 List prefetch .....	17
2.4 RID pools .....	18
2.5 Sequential detection versus a sorted RID list .....	19
2.6 Indirect references .....	20
2.7 Index insert I/O parallelism and conditional getpages .....	20
2.8 Declared global temporary tables prefetch .....	21
<b>Chapter 3. Buffer pool writes</b> .....	23
3.1 Deferred writes .....	24
3.2 Format writes .....	25
3.3 Write I/O metrics .....	26
3.4 I/O problems and data set level statistics .....	26
<b>Chapter 4. Data sharing</b> .....	29
4.1 GBPCACHE .....	30
4.1.1 AUTOREC .....	30
4.1.2 RATIO .....	31
4.2 GBP cast out .....	31
4.3 GBP write-around .....	31
4.4 Setting up your CFRM policies .....	32
<b>Chapter 5. Workfiles</b> .....	33
5.1 Controlling the growth of workfiles .....	34
5.2 Performance considerations for workfiles .....	35
5.3 Workfile buffer pools .....	36
<b>Chapter 6. Assigning page sets to buffer pools</b> .....	37
6.1 Multiple buffer pools .....	38

6.2 Choosing a page size .....	38
6.2.1 Indexes .....	38
6.2.2 Table spaces .....	39
6.2.3 Large objects .....	40
<b>Chapter 7. Improving CPU performance</b> .....	41
7.1 PGSTEAL(NONE/FIFO) .....	42
7.2 PGFIX(YES) .....	42
7.3 z/OS frame size .....	43
<b>Chapter 8. Index and data organization</b> .....	45
8.1 Index reorganization .....	46
8.2 Data reorganization .....	47
<b>Related publications</b> .....	49
IBM Redbooks .....	49
Other publications .....	49
Online resources .....	49
Help from IBM .....	50

# Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.*

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.


## COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

# Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at <http://www.ibm.com/legal/copytrade.shtml>

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

DB2®	Redpaper™	Tivoli®
FICON®	Redpapers™	UC™
IBM®	Redbooks (logo)  ®	z/OS®
OMEGAMON®	RMF™	zEnterprise®
Redbooks®	System z®	

The following terms are trademarks of other companies:

Other company, product, or service names may be trademarks or service marks of others.



# Preface

IBM® DB2® buffer pools are still a key resource for ensuring good performance. This has become increasingly important as the difference between processor speed and disk response time for a random access I/O widens in each new generation of processor. An IBM System z® processor can be configured with large amounts of storage, which if used wisely, can help compensate by using storage to avoid synchronous I/O. Several changes in buffer pool management have been provided by DB2 10 and DB2 11.

The purpose of this IBM Redpaper™ is to cover the following topics:

- ▶ Describe the functions of the DB2 11 buffer pools
- ▶ Introduce a number of matrixes for read and write performance of a buffer pool
- ▶ Provide information about how to set up and monitor the DB2 buffer pools

The paper is intended to be read by DB2 system administrators, but it might be of interest to any IBM z/OS® performance specialist. It is assumed that the reader is familiar with DB2 and performance tuning.

In this paper, we also assume that you are familiar with DB2 11 for z/OS performance. See DB2 11 for z/OS Technical Overview, SG24-8180; and DB2 11 for z/OS Performance Topics, SG24-8222, for more information about DB2 11 functions and their performance.

This paper is divided into eight chapters:

- ▶ In Chapter 1, “Introduction” on page 1, we introduce the basic concepts of buffer tuning.
- ▶ In Chapter 2, “Prefetch and RID pool” on page 13, we go into depth about DB2 prefetch, RID pool management, and the choices made by the DB2 optimizer that affect prefetching.
- ▶ In Chapter 3, “Buffer pool writes” on page 23, we describe the asynchronous write of DB2, the impact on setting the buffer pool variable thresholds, and look at the most important I/O metrics and how to obtain them.
- ▶ In Chapter 4, “Data sharing” on page 29, we consider data sharing and group buffer pools, and related impact on the Coupling Facility.
- ▶ In Chapter 5, “Workfiles” on page 33, we describe the assignment of workfiles to separate buffer pools for use by sort, star join, trigger, created temporary tables, view materialization, nested table expression, merge join, non-correlated subquery, sparse index, and temporary tables.
- ▶ In Chapter 6, “Assigning page sets to buffer pools” on page 37, we describe how to assign page sets to buffer pools, including the choice of page size, which affects the buffer pool assignments.
- ▶ In Chapter 7, “Improving CPU performance” on page 41, we describe buffer pool parameters such as **PGFIX(YES)** that can affect CPU time, apart from the effect that I/O minimization can reduce CPU time.
- ▶ In Chapter 8, “Index and data organization” on page 45, we stress the positive impact of REORG avoidance brought out by recent hardware and software functions.

## Authors

This paper was produced by a specialist working at the IBM Silicon Valley Lab, San Jose, CA.

**Jeff Berger** is a member of the DB2 for z/OS performance department at IBM Silicon Valley Laboratory. He has been with IBM for 34 years, mostly studying the performance of both the hardware and software of IBM mainframes, splitting his time between IBM storage, DB2, and the synergy among all of them. Jeff has presented at numerous z/OS and DB2 conferences around the world and he has contributed to IBM Redbooks® publications, published several technical IBM Redpapers™ and articles with Computer Measurement Group, IDUG Solutions Journal, IBM Systems Magazine, and IBM Data Magazine. He has a Master's degree in Statistics, Operations Research, and Industrial Engineering from UC Berkeley.

Thanks to the following people for their contributions to this project:

Brian Baggett  
Paolo Bruni  
Adrian Burke  
John Campbell  
Gopal Krishnan  
Terry Purcell  
Akira Shibamiya  
Steve Turnbaugh  
IBM Silicon Valley Lab, San Jose

Thanks to Mike Bracey, author of the previous Redpaper:

*DB2 9 for z/OS Buffer Pool Monitoring and Tuning*, REDP-4604 published in October 2009.

## Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:

[ibm.com/redbooks/residencies.html](http://ibm.com/redbooks/residencies.html)

## Comments welcome

Your comments are important to us!

We want our papers to be as helpful as possible. Send us your comments about this paper or other IBM Redbooks publications in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

[ibm.com/redbooks](http://ibm.com/redbooks)

- ▶ Send your comments in an email to:  
[redbooks@us.ibm.com](mailto:redbooks@us.ibm.com)
- ▶ Mail your comments to:  
IBM Corporation, International Technical Support Organization  
Dept. HYTD Mail Station P099  
2455 South Road  
Poughkeepsie, NY 12601-5400

## Stay connected to IBM Redbooks

- ▶ Find us on Facebook:  
<http://www.facebook.com/IBMRedbooks>
- ▶ Follow us on Twitter:  
<http://twitter.com/ibmredbooks>
- ▶ Look for us on LinkedIn:  
<http://www.linkedin.com/groups?home=&gid=2130806>
- ▶ Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:  
<https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm>
- ▶ Stay current on recent Redbooks publications with RSS Feeds:  
<http://www.redbooks.ibm.com/rss.html>





# Introduction

In this chapter, we introduce the objectives of buffer pool tuning, the major classifications of buffers and getpages, and the major parameters to define and manage the buffer pools.

This chapter contains the following topics:

- ▶ Buffer pool attributes
- ▶ Buffer pool tuning
- ▶ Getpage classification and buffer classification
- ▶ Using VPSEQT
- ▶ Utility use of most recently used
- ▶ Page residency time
- ▶ AUTOSIZE(YES)
- ▶ Critical thresholds

## 1.1 Buffer pool attributes

Buffer pools are used as a cache to help avoid I/O. Buffer pools are in the DB2 DBM1 primary address space above the 2 GB bar. The sum of the storage that is available in all buffer pools cannot exceed 1 TB.

The buffer pools and their sizes, and the default assignments for each page size, are initially defined during the installation or migration of DB2 using panels DSNTIP1 and DSNTIP2. DB2 stores buffer pool attributes in the DB2 bootstrap data set (BSDS). See *DB2 11 for z/OS Installation and Migration Guide*, GC19-4056.

At CREATE time, the BUFFERPOOL bpname identifies the buffer pool to be used for the table space and determines the page size of the table space.

You can change the buffer pool sizes online with the **ALTER BUFFERPOOL** command. The **ALTER BUFFERPOOL** command that is shown in Figure 1-1 lists the attributes that are available for the buffer pool definition. See *DB2 11 for z/OS Command Reference*, SC19-4054.

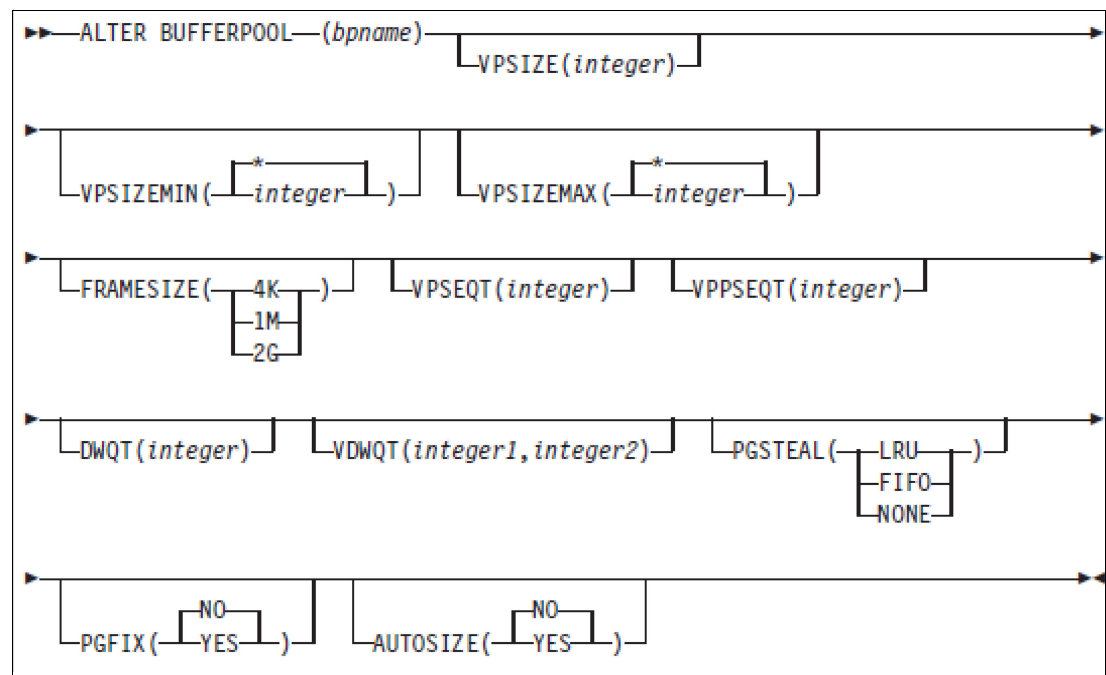


Figure 1-1 ALTER BUFFERPOOL options

Here is a brief description of all options:

**(bpname)**

Specifies the name of the buffer pool to alter.

**VPSIZE (integer)**

Defines the virtual pool size in number of buffers.

**VPSIZEMIN (integer)**

Specifies the minimum number of buffers to allocate to the active virtual pool when AUTOSIZE(YES) is in effect.

**VPSIZEMAX (integer)**

Specifies the maximum number of buffers to allocate to the active virtual pool when AUTOSIZE(YES) is in effect.

**FRAMESIZE(4K/1M/2G)**

Sets the frame size for the buffer pool. Possible values are 4 KB, 1 MB, or 2 GB.

**VPSEQT (integer)**

Sets the sequential steal threshold for the buffer pool.

This threshold is a percentage of the buffer pool that might be occupied by sequentially accessed pages.

**VPSEQT (integer)**

Sets the parallel sequential threshold for the buffer pool.

This threshold is expressed as a percentage of the sequential steal threshold (VPSEQT) and determines how much of the buffer pool is used for parallel processing operations.

**DWQT (integer)**

Sets the buffer pool's deferred write threshold (DWTH).

This threshold is a percentage of the virtual buffer pool that might be occupied by unavailable pages, including updated pages and pages in use. DB2 checks this threshold when an update to a page is complete. If the percentage of unavailable pages in the virtual buffer pool exceeds the threshold, write operations are scheduled for enough data sets (at up to 128 pages per data set) to decrease the number of unavailable buffers to 10% below the threshold.

**VDWQT (integer1, integer2)**

Sets the vertical deferred write threshold.

This threshold is expressed as a percentage of the virtual buffer pool that can be occupied by updated pages from a single data set. It is checked whenever an update to a page is completed. If the percentage of updated pages for the data set exceeds the threshold, writes are scheduled for that data set.

**PGSTEAL**

Specifies the page-stealing algorithm that DB2 uses for the buffer pool.

Least recently used (LRU) keeps pages in the buffer pool that are being used frequently and removes unused pages. It ensures that the most frequently accessed pages are always in the buffer pool.

First-in first-out (FIFO) removes the oldest pages no matter how frequently they are referenced. This approach to page stealing results in a small decrease in the cost of doing a getpage operation, and can reduce internal DB2 latch contention in environments that require very high concurrency. Use it for buffer pools that have table space or index entries that should always remain in memory.

(NONE) keeps all pages for an object resident in the buffer pool.

## PGFIX

Specifies whether the buffer pool should be fixed in real storage when it is used.

To prevent PGFIX(YES) buffer pools from exceeding the real storage capacity, DB2 uses an 80% threshold of real storage on the logical partition (LPAR) when allocating PGFIX(YES) buffer pools. If the threshold is exceeded, DB2 overrides the PGFIX(YES) option with PGFIX(NO).

## AUTOSIZE

Specifies whether the buffer pool size autonomic adjustment is turned on or off.

DB2 performs dynamic buffer pool size adjustments that are based on real-time workload monitoring.

**Note:** When you migrate to a new version of DB2 where the default value of one of the buffer pool parameters was changed, the *default* value has meaning only in the context of a brand new installation, not for the migration. The migration keeps the old definitions. You need to **ALTER** to the new values. Check the values with the **DISPLAY BUFFERPOOL**.

The DB2 command **DISPLAY BUFFERPOOL** displays the status for one or more active or inactive buffer pools. See Figure 1-2.

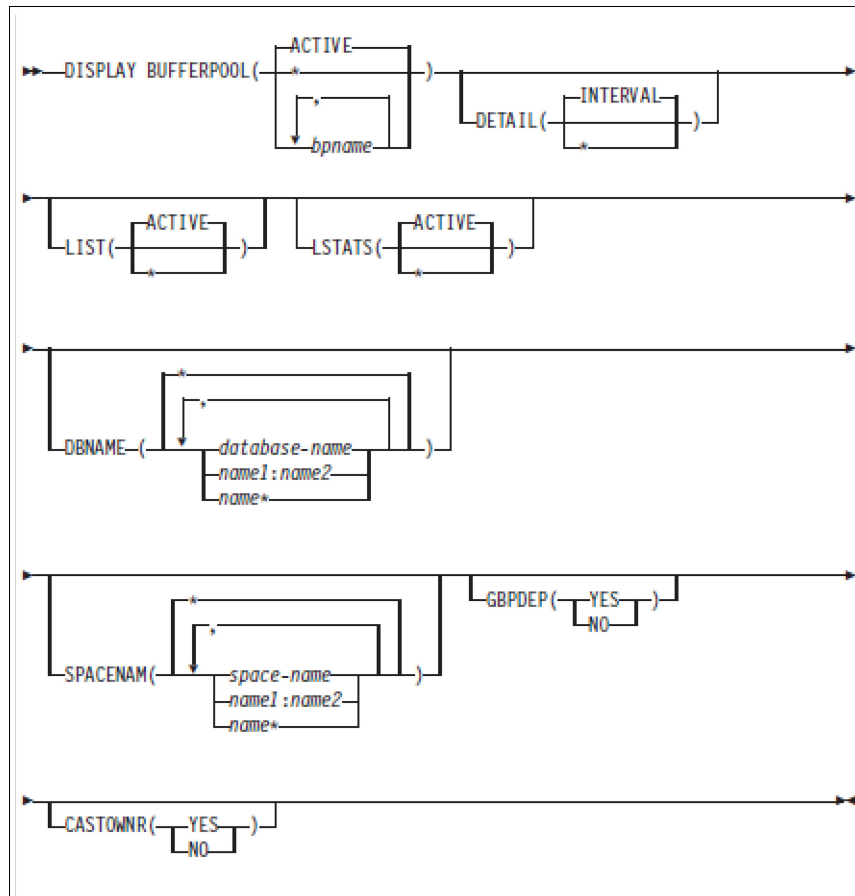


Figure 1-2 **DISPLAY BUFFERPOOL** command



The DSNB401I message, which is shown in Example 1-1, displays the output from the **DISPLAY BUFFERPOOL DETAIL** command with a wealth of information. See *DB2 11 for z/OS Messages*, GC19-4062 for its format and sections.

*Example 1-1 DISPLAY BPOOL(BP0) DETAIL(\*) output*

---

```

-DISPLAY BPOOL(BP0) DETAIL(*)
DSNB401I  -DB1A BUFFERPOOL NAME BP0, BUFFERPOOL ID 0, USE COUNT 251
DSNB402I  -DB1A BUFFER POOL SIZE = 20000 BUFFERS  AUTOSIZE = NO
          VPSIZE MINIMUM = 0 VPSIZE MAXIMUM = 0
          ALLOCATED      =    20000    TO BE DELETED    =          0
          IN-USE/UPDATED =        176
DSNB406I  -DB1A PGFIX ATTRIBUTE -
          CURRENT = NO
          PENDING = NO
          PAGE STEALING METHOD = LRU
DSNB404I  -DB1A THRESHOLDS -
          VP SEQUENTIAL   = 80
          DEFERRED WRITE  = 30    VERTICAL DEFERRED WRT = 5, 0
          PARALLEL SEQUENTIAL =50    ASSISTING PARALLEL SEQT= 0
DSNB546I  -DB1A PREFERRED FRAME SIZE 4K
          20000 BUFFERS USING 4K FRAME SIZE ALLOCATED
DSNB410I  -DB1A CUMULATIVE STATISTICS SINCE 09:43:53 MAR 18, 2014
DSNB411I  -DB1A RANDOM GETPAGE    =1875474
          SYNC READ I/O (R) =1420
          SEQ.  GETPAGE    =734139
          SYNC READ I/O (S) =28
          DMTH HIT          =0
          PAGE-INS REQUIRED =2665
          SEQUENTIAL        =1546
          VPSEQT HIT        =0
          RECLASSIFY        =165
DSNB412I  -DB1A SEQUENTIAL PREFETCH -
          REQUESTS          =5
          PREFETCH I/O      =5
          PAGES READ        =121
DSNB413I  -DB1A LIST PREFETCH -
          REQUESTS          =479
          PREFETCH I/O      =27
          PAGES READ        =164
DSNB414I  -DB1A DYNAMIC PREFETCH -
          REQUESTS          =23229
          PREFETCH I/O      =120
          PAGES READ        =932
DSNB415I  -DB1A PREFETCH DISABLED -
          NO BUFFER         =0
          NO READ ENGINE    =0
DSNB420I  -DB1A SYS PAGE UPDATES  =637197
          SYS PAGES WRITTEN =30954
          ASYNC WRITE I/O   =8596
          SYNC WRITE I/O    =10852
          PAGE-INS REQUIRED  =0
DSNB421I  -DB1A DWT HIT           =0
          VERTICAL DWT HIT  =0
DSNB440I  -DB1A PARALLEL ACTIVITY -
          PARALLEL REQUEST  =0

```

```
DEGRADED PARALLEL=0
1DSNB441I -DB1A LPL ACTIVITY -
          PAGES ADDED      =0
DSN9022I -DB1A DSNB1CMD '-DISPLAY BPOOL' NORMAL COMPLETION
```

---

## 1.2 Buffer pool tuning

We start with a clear understanding of some of the goals of buffer pool tuning:

- ▶ Minimize the frequency of synchronous read I/O suspension time
- ▶ Optimize DB2 prefetch to minimize Other Read I/O suspension time
- ▶ Protect useful random pages in the buffer pool from being flushed out by prefetch
- ▶ Minimize the impact of long running queries, utilities, and sequential inserts on online transactions
- ▶ Minimize Other Write I/O suspension time
- ▶ Minimize the CPU cost

The Class 3 accounting of DB2 distinguishes the suspension time that is associated with page sets. The follow types of suspensions can occur:

- ▶ Synchronous database I/O
- ▶ Other Read I/O
- ▶ Other Write I/O
- ▶ Page latch suspension

Synchronous database I/Os initiated by a given allied thread is reported for both read and write I/Os.

The suspension of a getpage that must wait for a prefetch operation to complete is categorized as Other Read I/O. Other Read I/O suspension can also be caused by a different thread that started a synchronous I/O for the same page.

An I/O that writes a page to disk does not latch the page, but any update request for such a page is suspended if a write I/O is in progress for that page. That suspension time is called *Other Write I/O suspension*. This update request also holds a latch on the page while it is suspended waiting for the write I/O to complete. However, if another update request tries to obtain this page latch, it is suspended and the suspension time is reported as Latch suspension time, not Other Write I/O suspension. Consequently, if you see high page latch suspension, it is symptomatic of concurrent updates. Oftentimes these update requests are trying to update the same space map page.

Data Manager makes getpage requests to read data pages. Index Manager makes getpage requests to read index pages. LOB Manager makes getpage requests to large objects (LOBs). Sometimes these requests are on behalf of SQL operations and sometimes they are on behalf of DB2 utilities.

Buffer pools are used as a cache to help avoid synchronous read I/O. Synchronous I/O typically reads random pages, not sequential pages.

Some getpages merely allocate a buffer to write a new page. These pages are internally called *READ(NO) getpages*, which never do any I/O.

Random buffer hits occur when a page is re-referenced in the buffer pool before the buffer containing the page is stolen. The bigger the buffer pool is, the longer the page residency time is, and the larger the buffer hit ratio tends to be.

In contrast, sequential access patterns depend on various prefetch I/O strategies to avoid synchronous I/Os. Most prefetch uses some kind of read-ahead algorithm. In these cases, the goal is not to avoid I/O, but rather to avoid or minimize Other Read I/O suspension.

All prefetch operations are scheduled to run as a service request block (SRB) in the DBM1 address space (Database Manager), sometimes called *prefetch engines*. DB2 supports 600 prefetch engines. If DB2 runs out of prefetch engines, prefetch becomes disabled. The prefetch SRBs (as well as many other DB2 SRBs) are eligible to run on a System z Integrated Information Processor (zIIP) processor. To minimize the software CPU cost of DB2, configure enough zIIP processors to accommodate DB2 peak prefetch processing in order to minimize scheduling delays.

Prefetch can also be disabled if there are an insufficient number of sequential buffers to perform the prefetch I/O, or because 90% of the buffer pool contains *dirty* pages. The DB2 buffer manager never steals a dirty buffer until that buffer can be written to DASD or the coupling facility.

The disablement of prefetch can be identified in the DB2 statistics. However, more insidious and devastating than prefetch disablement is the following: Prefetched pages being stolen from the buffer pool before the corresponding getpages occur. That can happen when doing lots of parallel prefetching if there is an insufficient number of sequential buffers, or if the queries or utilities have trouble getting dispatched by the system. In general, you need  $2P$  sequential buffers for each prefetch stream, where  $P$  is the prefetch quantity. In the coming chapters, we explain what the prefetch quantity is. To judge whether sequential buffers are getting stolen before the getpages, look in the DB2 statistics for a high count of sequential synchronous I/Os. It may be difficult, however, to correlate these events with a particular transaction because the accounting traces do not distinguish synchronous sequential I/O from synchronous random I/O.

The prefetch engine first searches the buffer pool to see if any of the pages are already there. Since there is overhead to schedule the prefetch engine and search the buffer pool, if you expect the page to be resident in the buffer pool and would like DB2 not to schedule prefetch, you can use PGSTEAL(NONE) or VPSEQT(0).

One exception where the prefetch engine is not always scheduled is sequential prefetch. When using sequential prefetch, DB2 waits until the first getpage miss to schedule the engine.

DB2 provides buffer pool performance information in both the statistics trace records and accounting trace records (IFCID 3) aggregated by buffer pool. The granularity of this information by buffer pool is itself one of the motivations for assigning page sets to different buffer pools. In other words, there is informational value in creating multiple buffer pools and assigning different categories of objects to specific buffer pools.

Both the accounting and statistics traces include the total number of getpages and the total number of prefetch requests by prefetch type, and the total number of pages asynchronously prefetched. However, the granularity of information is much greater in the statistics, which represent aggregate counts for the DB2 subsystem for each statistics interval.

For example, the statistics trace records distinguish random getpages from sequential getpages; the accounting trace records do not. The statistics records also distinguish the number of pages that are prefetched by prefetch type; the accounting trace records do not. Lastly, the statistics trace distinguishes between the number of prefetch requests from the

number of prefetch I/Os; the accounting trace records do not. The accounting trace records contain only the number of prefetch requests.

If the prefetch engine finds all of the pages in the buffer pool, there is no I/O, but the scheduling of the SRB is counted as a request. A large delta between the number of requests and the number of prefetch I/Os is a clue that CPU time is being wasted needlessly scheduling the prefetch engines. From the statistics trace, you can also calculate how many pages were prefetched per I/O by dividing the total number of asynchronous pages prefetched by the number of prefetch I/Os.

## 1.3 Getpage classification and buffer classification

Getpages are always classified as either random or sequential. When considering the buffer hit ratio or page residency time, you should not mix random and sequential getpages because doing so is like mixing apples and oranges.

Instead of looking at the sequential buffer hit ratio, look at the absolute number of sequential synchronous I/Os. If that number is large, you have a problem. It means that either prefetch was disabled, or prefetch occurred and the pages were stolen before the sequential getpages.

Data manager, index manager, and LOB manager determine how the getpages are classified. How the buffers are internally classified does not always match how the getpages are classified. For example, suppose that a page is prefetched into a buffer. Then, the buffer is always classified as sequential, which means the buffer resides on the Sequential Least Recently Used (SLRU) chain. If the page is later referenced by a random getpage, the buffer is reclassified as random, meaning that it is removed from the SLRU chain. The reverse is not true. That is, if a page is first read into the buffer pool by a random getpage and then later referenced by a sequential getpage, the buffer is not added to the SLRU chain.

Buffers on the SLRU chain are called *sequential buffers*. Generally speaking, the length of the SLRU chain cannot exceed VPSEQTxVPSIZE. Thus, VPSEQT limits the number of buffers that can be used by prefetch operations. The length of the SLRU chain can be determined from the SEQUENTIAL statistic that can be obtained by using the **DISPLAY BUFFERPOOL** command with the DETAIL option. The DB2 IFCID 2 statistics record contains both the maximum and minimum value (QBSTSMIN and QBSTSMAX) during the statistics interval.

Usually the getpages that are associated with DB2 prefetch are classified as sequential, but this is not always the case. It is true for disorganized indexes, RID list scans, and prefetch of LOBs. It is true for sequential prefetch and dynamic prefetch as well. However, it is not true for list prefetch when it is used for Incremental Copy, Fast Log Apply, or Runstats table sampling.

DB2 utilities such as LOAD REPLACE, REBUILD INDEX, and REORG use format writes to sequentially write or rewrite to a data set to which it has exclusive access. READ(NO) getpages that are associated with format writes are classified as sequential. However, these buffers are treated as most recently used (MRU). That means that as soon as the pages are written, the buffers are likely to be reused for a different page.

There are some anomalies in these getpage statistics to consider. For example, the prefetch I/Os that are counted as sequential prefetch for Index I/O Parallelism are actually random. Alternatively, the READ(NO) getpages that are associated with sequential inserts are incorrectly classified as random. Therefore, the sequential inserts may devastate the buffer pool hit ratio, but you may observe a false indication that the random buffer pool hit ratio is high. Even if these sequential data pages are written quickly to DASD, they force out other

pages from the buffer pool. Since the pages are not on the SLRU chain, lowering VPSEQT does not help.

In addition to sequential inserts, the DB2 utilities also use READ(NO) getpages to write to page sets sequentially. However, the utilities correctly classify these getpages as sequential to avoid affecting the random buffer hit ratio calculation. Buffer manager also enforces MRU with format writes so that the buffers are likely to be stolen as soon as the pages are written to DASD. Furthermore, DB2 occasionally suspends the utilities while it flushes out all of the dirty pages to DASD. That is why DB2 utilities do not affect the buffer pool as much as sequential inserts do.

## 1.4 Using VPSEQT

You can use VPSEQT to limit the length of the SLRU chain for a buffer pool. The default value for VPSEQT is 80%. For example, suppose that VPSIZE=50000 and VPSEQT=80. The length of the SLRU chain can be at most 40000, in general. The exception is a newly allocated buffer pool. If there were never any random getpages since the buffer pool was allocated, the length of the SLRU chain could reach 50000.

Buffers on the SLRU chain are called *sequential buffers*, but they include buffers allocated by any type of prefetch, even if list prefetch is used to read a random set of pages. To achieve good prefetch performance, the number of sequential buffers needs to be large enough to achieve the level of prefetch parallelism of your workload. You should allow at least 2P buffers for each prefetch stream, where P is the prefetch quantity. For example, if P is 64 pages and you want to allow for 100 concurrent prefetch streams, you need 12800 sequential buffers. If it turns out that there are more than 100 concurrent streams, 12800 sequential buffers are insufficient and you might observe sequential synchronous I/Os despite the prefetch I/O being done because buffers containing prefetched pages could get stolen before the getpages occur. If you detect this, then you could do one of the following actions:

- ▶ Reduce the number of prefetch streams
- ▶ Increase the number of sequential buffers
- ▶ Schedule the work that is using prefetch to run at a different time of day

## 1.5 Utility use of most recently used

As noted previously, the utilities use MRU for format writes. They also use MRU when they use sequential prefetch. However, MRU does not avoid the necessity to provide enough sequential buffers so that the prefetched pages are not stolen before the getpages.

## 1.6 Page residency time

Page residency time is as simple to calculate as the hit ratio, but for some people it is a more intuitive metric. Mike Bracey in *DB2 9 for z/OS: Buffer Pool Monitoring and Tuning*, REDP-4604, described residency time extensively, but here we limit the focus on random page residency time, with the eye towards avoiding random synchronous I/O. The random page residency time is calculated as the maximum of the following two formulas:

$$\text{VPSIZE} / \text{total pages read per second (including pages prefetched)}$$
$$\text{VPSIZE} * (1 - \text{VPSEQT}/100) / \text{random synchronous I/O/sec}$$

Let us consider one example where VPSIZE is 100000, VPSEQT is 80%, there are no prefetch I/Os, and the random synchronous I/O rate is 1000/second. The first formula computes to 100 seconds and the second formula computes to 20 seconds. Hence, the random residency time is 100 seconds. But, in the presence of say 1000 pages being prefetched per second, the first formula reduces to only 50 seconds. In other words, if the rates at which random and sequential pages are entering the buffer pool are the same, the effect is that the random pages have access to only half of the buffer pool, and consequently the residency time is also cut by half.

However, as the formula shows, you can use VPSEQT to limit the impact that prefetch has on the residency time of random pages.

For example, if you set VPSEQT to 1%, the second calculation would change to 99 seconds. Therefore, if the workload reads 1000/second random pages and 1000/second sequential pages, lowering VPSEQT from 80% to 1% increases the random page residency time from 50 to 99 seconds. Of course, doing so would dramatically degrade sequential performance, but random page residency time analysis has nothing to do with the analysis of prefetch performance.

One of the interesting aspects of page residency metric is that, unlike buffer hit ratios, it does not depend on the number of getpages. Consequently, it is insensitive to READ(NO) getpages or buffer hits of any kind. For that reason, it is, in a sense, a more reliable measure of buffer pool performance than the random buffer hit ratio.

## 1.7 AUTOSIZE(YES)

The AUTOSIZE option on ALTER BUFFERPOOL allows Workload Manager (WLM) to determine if buffer pool I/O is the predominant delay for a random getpage. If so, it uses histograms to determine whether increasing the buffer pool size could help achieve the performance goal. Depending on the amount of storage available, WLM may instruct DB2 to increase the buffer pool or first decrease another buffer pool and then increase the buffer pool in question.

You can control the maximum growth or shrinkage in the buffer pool size by using VPSIZEMAX and VPSIZEMIN. Even if you set these parameters, you must also specify AUTOSIZE(YES) to allow WLM to modify the buffer pool size. If you do not specify these parameters, the default fluctuation in buffer pool size is +/-25%. When DB2 reallocates the buffer pool, it remembers the last buffer pool size used and that becomes the new baseline for applying the +/-25% range.

AUTOSIZE(YES) does not require z/OS 2.1, but IBM recommends that you not use it until you migrate to z/OS 2.1. Still, no customers have tested this function yet in production with z/OS 2.1. If you want to use it, you need to test it before considering it for production.

## 1.8 Critical thresholds

There are three so-called fixed critical thresholds that are reported in the DB2 statistics and the **DISPLAY BUFFER POOL** command:

- ▶ Sequential prefetch threshold (SPTH)
- ▶ Data management threshold (DMTH)
- ▶ Immediate write threshold (IWTH)

These thresholds relate to the number of unavailable buffers. A buffer might be unavailable for one of two reasons: either the page in the buffer is dirty or the buffer was acquired by a getpage and it has not yet been released. A *dirty page* is one that has been modified and not yet written to DASD or to the Group Buffer Pool in data sharing.

Starting from the least severe and working the way up to the most severe, the *prefetch threshold* is 90%. When the number of unavailable buffers reaches this threshold, DB2 disables prefetch. In IBM OMEGAMON®, you can look in the statistics report for a message that says PREF.DISABLED-NO BUFFER >0. This message is generally an indicator of a problem, but there are exceptions. For example, when VPSEQT=0 you are asking to disable prefetch.

You could also get this message if your workload is doing lots of parallel prefetch and VPSEQT limited the number of buffers that prefetch can use. Thus, the message does not necessarily mean that you hit the prefetch threshold of 90%.

The next critical threshold is called the *data management threshold*, which is 95%. When the number of unavailable buffers reaches this threshold, sequential scans release the getpages after processing each row. For example, if there are 20 rows per page, you could see 20 getpages for each page. This obviously causes much extra CPU overhead. You might also see elongated lock and latch times for transaction updates. Look for DM THRESHOLD > 0 messages in the DB2 statistics report.

The third critical threshold is called the *immediate write threshold*, which is 97.5%. In this case, if the application updated multiple rows in a page, not only can you observe prefetch disabled and more getpages, you also begin to see one synchronous write I/O for each row that is updated. The fallout from just one buffer pool hitting this threshold is felt at subsystem or group-wide level. The immediate write threshold is not reported directly, but if you observe very elevated synchronous write counts in the statistics, it is likely because you hit the immediate write threshold.







## Prefetch and RID pool

Prefetch is a mechanism for reading ahead a set of pages into the buffer pool with only one asynchronous I/O operation.

Prefetch can allow substantial savings in both CPU and I/O by avoiding costly synchronous read I/O operations. DB2 uses different types of prefetch in different situations, including dynamic prefetch, sequential prefetch, and list prefetch:

- ▶ Sequential prefetch

Sequential prefetch is used for table scans. The maximum number of pages read by a request that is issued from your application program is determined by the size of the buffer pool used.

- ▶ Dynamic prefetch

With dynamic prefetch, DB2 uses a sequential detection algorithm to determine whether data pages are being read sequentially.

- ▶ List prefetch

List prefetch reads a set of data pages that are determined by a list of record identifiers (RIDs) taken from an index. List prefetch is also used for disorganized indexes and large objects (LOBs).

List prefetch access paths are ideally suited for queries where the qualified rows, as determined in index key sequence, are not sequential, are skip-sequential but sparse, or when the value of the `DATA REPEAT FACTOR` statistic is large.

A record identifier (RID) pool of sufficient size can improve the performance of transactions that use the RID processing for:

- ▶ List prefetch, including single index list prefetch access paths
- ▶ Enforcing unique keys for multi-row updates
- ▶ Multiple index access paths
- ▶ Hybrid joins

This chapter contains the following sections:

- ▶ Sequential prefetch
- ▶ Sequential detection and dynamic prefetch

- ▶ List prefetch
- ▶ RID pools
- ▶ Sequential detection versus a sorted RID list
- ▶ Indirect references
- ▶ Declared global temporary tables prefetch

## 2.1 Sequential prefetch

*Sequential prefetch* is used exclusively for table scans and table space scans. What is the difference between a table scan and a table space scan? There is no difference if a table space contains a single table, but there is a difference if a table space consists of multiple tables. A traditional segmented table space can contain multiple tables. As of DB2 11, a universal table space (UTS) is limited to one table.

Sequential prefetch involves trigger pages. Let us say  $P$  is the prefetch I/O quantity. For example, if the page size is 4 KB, the prefetch quantity could be as large as 128 pages. All getpages for a page number that is a multiple of  $P$  is a trigger page. The getpage for page  $N$  initiates prefetch I/O ahead for pages  $N+P$  through  $N+2P-1$ . If this I/O completes before the getpage for page  $N+P$ , the query or utility is not suspended. In general, only one sequential prefetch I/O can occur at a time, because Buffer Manager waits for page  $N$  to be in the buffer pool before that getpage schedules another prefetch I/O. The only exception is at the start of the scan, where two prefetch I/Os are scheduled at the same time.

The sequential prefetch quantity for DB2 in terms of kilobytes is found in Table 2-1 for page sets other than work files.

Table 2-1 Sequential quantity

VPSIZE	SQL	Utilities
4 KB pages: 0 KB to less than 900 KB Other page sizes: 0 KB to less than 384 KB	32 KB	64 KB
4 KB pages: 900 KB to less than 4 MB Other page sizes: 384 KB to less than 3.2 MB	64 KB	128 KB
4 KB pages: At least 4 MB Other page sizes: At least 3.2 MB	128 KB	256 KB

With 4 KB pages if VPSIZE is at least 4 MB (1000 buffers), or with other pages if VPSIZE is at least 3.2 MB, Buffer Manager uses the product of VPSIZE and VPSEQT to decide if it should further increase the prefetch quantity, as shown in Table 2-2.

Table 2-2 Additional sequential quantity

VPSIZE x VPSEQT	SQL	Utilities
At least 160 MB but less than 320 MB	256 KB	256 KB
At least 320 MB	256 KB	512 KB

Table 2-2 shows that in order to maximize the sequential prefetch quantity for DB2 utilities and therefore to maximize throughput, your buffer pool size should be at least 320 MB.

In a multi-table table space, a table might become fragmented, meaning that the segments of the table are not in physical sequence. REORG eliminates the fragmentation. Storing multiple *small* tables in one table space is recommended to minimize the number of data sets that

DB2 must manage. This also improves the efficiency of managing image copies since you need only one image copy per table space.

However, storing multiple *large* tables in one table space can lead to poor prefetch performance. Here is why. Sequential prefetch is managed by Buffer Manager. Buffer Manager does not consider segment boundaries for a table scan. When Buffer Manager triggers a prefetch for pages  $N+P$  through  $N+2P+1$ , what happens if those pages are in a different table? Buffer Manager happily prefetches them. Furthermore, if the next segment of the table is somewhere else, those pages are not prefetched. Consequently, the performance suffers.

Data Manager detects when the next segment is not contiguous to the current segment and it switches from sequential prefetch to dynamic prefetch. However, dynamic prefetch is also very poor at dealing with fragmented tables. Why is DB2 designed this way? Because DB2 does not expect you to mix large tables in one table space. So, do not do it.

Work files use a smaller sequential prefetch quantity than regular table spaces because they are an extreme case of multi-table segmented table spaces and heavy parallelism. For 4 KB page work files, if VPSIZE is less than or equal to 1400, the prefetch quantity is four pages; for larger buffer pools it is eight pages. For 32 KB work files, if VPSIZE is less than or equal to 175, the prefetch quantity is one page; otherwise it is two pages.

## 2.2 Sequential detection and dynamic prefetch

DB2 sometimes uses a technique called *sequential detection*. Data Manager uses this algorithm for data pages that are accessed directly via an index, and Index Manager uses the same algorithm for index pages. Sequential detection and sequential prefetch are mutually exclusive. In other words, when DB2 is using sequential detection, Data Manager and Index Manager instruct Buffer Manager to not use sequential prefetch.

The maximum dynamic prefetch quantity for DB2 in terms of kilobytes is found in Table 2-3.

Table 2-3 Prefetch quantity

VPSIZE	SQL
4 KB pages: 0 KB to less than 900 KB Other page sizes: 0 KB to less than 384 KB	32 KB
4 KB pages: 900 KB to less than 4 MB Other page sizes: 384 KB to less than 3.2 MB	64 KB
4 KB pages: At least 4 MB Other page sizes: At least 3.2 MB	128 KB

Notice that dynamic prefetch quantity is the same for SQL and utilities. In practice, dynamic prefetch is rarely used by utilities anyway. One exception where utilities use dynamic prefetch is when REORG must read from a non-partitioned secondary index (NPI). In that case, REORG can use dynamic prefetch to read the leaf pages of the NPI.

When using sequential detection, each getpage is judged to be either page-sequential or not page-sequential, based on its distance from the previous getpage. This distance criterion is half of the prefetch quantity. Hence, it is usually plus or minus 64 KB.

DB2 keeps a running count over the last eight getpages of the number of page-sequential getpages. When this count exceeds four, a getpage can schedule dynamic prefetch. In

addition to basic sequential detection, Data Manager applies Row Level Sequential Detection (RLSD). RLSD does not apply to indexes. RLSD tracks the last eight rows. When Data Manager advances to the next row in the same page as the previous row, the row can be counted as though it were a page-sequential getpage, even though there is no additional getpage. But, the row is only counted if the previous count was two or less. Thus, DB2 can accumulate only up to three rows for the first data page. The second getpage could increase the count to four and the third getpage could increase the count to five. Therefore, dynamic prefetch may be triggered by a transaction that touches a minimum of five rows and three pages. But, in the case of an index scan, the minimum is always five leaf pages.

The dynamic prefetch I/O quantity does not start out at the maximum. The initial dynamic prefetch quantity is one quarter of the maximum. (The definition of what constitutes a page-sequential getpage is still determined by the maximum prefetch quantity.) The second prefetch I/O doubles the prefetch quantity and thereafter the prefetch quantity is the maximum. This stepwise increase in the prefetch quantity is known as the progressive prefetch quantity. 32 KB pages are treated as an exception; in this case, the initial prefetch quantity is two pages and then progresses to four pages, and then ceases to increase.

There is one more aspect to dynamic prefetch. Which getpages trigger the next prefetch request? Dynamic prefetch maintains a prefetch window. A getpage that falls within that window may trigger the next prefetch. After the initial set of pages that are prefetched, the prefetch window becomes the second half of those prefetched pages. Subsequently, when the prefetch quantity reaches its maximum, the prefetch window is the entire set of pages that were last prefetched. But, unlike sequential prefetch, the next prefetch occurs before, not after, the getpage.

Sequential detection works well for skip sequential access to the data when the cluster ratio of the index is high and there is a screening predicate. For example, if a query needs to read every 10th page, all of the pages are page-sequential, and Data Manager prefetches every page. The first prefetch I/O reads eight pages (say 1 - 8), but the first getpage within the range of pages 4 - 7 triggers a prefetch for 16 pages (9 - 24). This also means that DB2 is bringing some pages into the buffer pool for which there are no getpages.

Like sequential prefetch, dynamic prefetch also has a read ahead algorithm. To illustrate, suppose that the progressive prefetch quantity has reached its maximum value of  $P$ . Also suppose that the query is sequential and the last getpage to trigger prefetch was pages  $N - P$ : the next page to trigger a prefetch I/O is page  $N$ . That page would have triggered a prefetch I/O for pages  $N$  through  $N+P-1$ . The getpage for page  $N$  will then trigger a prefetch for pages  $N+P$  through  $N+2P-1$ . However, unlike sequential prefetch, the prefetch occurs before the getpage of page  $N$ ; therefore, if the prefetch for pages  $N$  through  $N+P-1$  has not yet completed, the prefetch for pages  $N+P$  through  $N+2P-1$  commences before the getpage for page  $N$  is suspended. The result is two parallel I/Os of  $P$  pages each. Therefore, the query needs to have up to  $2P$  pages allocated. If  $VPSIZE \times VPSEQT$  is too small to accommodate  $2P$  buffers, the prefetch might not occur.

The above paragraph is also true when the prefetch quantity has not reached its maximum value, except that having previously prefetched  $P$  pages, the next prefetch I/O will read  $2P$  pages. Therefore, the getpage for page  $N$  reads pages  $N+P$  through  $N+3P-1$ .

Like sequential prefetch, dynamic prefetch also has a read ahead algorithm. A getpage for triggering page reads pages  $N+P$  through  $N+2P-1$ . However, unlike sequential prefetch, the prefetch occurs before the getpage of page  $N$ . Therefore, dynamic prefetch can schedule two parallel I/Os if the query is I/O bound. Whether the query does two parallel I/Os or not, the query needs to have up to  $2P$  buffers allocated.

Now let us look at an example of data getpages. Suppose that  $P=32$  and that the sequence of getpages is the following:

10(1 row), 11(2 rows), 13, 15, 18, 19, 21, 22, 25, 26, 29, 30, 33, 34, 36, 38, 41, 42, ....

The first time that the page-sequential count has reached 5 is as DB2 is about to do a getpage for page 15. Before doing the getpage, DB2 triggers prefetch for eight pages, numbered 15 - 22, and the prefetch window becomes 19 - 22. Page 19 falls within that window. Before the getpage for page 19, DB2 triggers prefetch for 16 pages, numbered 23 - 38, and the prefetch window becomes 31 - 38. Page 33 is the first accessed page that falls within that window. Before the getpage for page 33, DB2 triggers prefetch for 32 pages, numbered 39 - 70, and the prefetch window becomes 39 - 70. Before the getpage for page 41, DB2 triggers prefetch for 32 pages, numbered 71 - 102. Notice that now there could be two concurrent I/Os because the prefetch for pages 39 - 70 has not necessarily completed. In other words, dynamic prefetch uses sequential I/O parallelism.

Sequential detection does not eliminate synchronous I/Os. Unclustered pages are never prefetched. Even if the cluster ratio is 100%, if the skip distance is greater than  $P/2$  pages, the page is not considered page-sequential and a synchronous I/O may occur.

But, there is yet one other feature of DB2 sequential detection that deals with unclustered pages. When DB2 is determining whether the next page is page-sequential, it checks the last two page numbers. If the next page to be accessed is the exact same page as the one two pages ago, it is treated as page-sequential even if the page in between was an unclustered page, far away from the clustered pages. Thus, if DB2 does not encounter two consecutive unclustered rows, the page-sequential count for a range scan never drops below seven once it reaches eight. Dynamic prefetch is continued if there are not two consecutive unclustered rows on two different pages. This ensures that for a reasonably well clustered table, DB2 always prefetches the clustered pages. But, this particular access path never prefetches the unclustered pages. Nor does this access path prefetch the pages if a screening predicate is such that the data pages to be read are sparse (that is not page-sequential.)

## 2.3 List prefetch

When the pages accessed are not purely sequential, list prefetch I/O is generally more desirable than dynamic prefetch because it can target a specific set of pages. In the example where the query needs to read every tenth page, list prefetch would only read the specific pages needed, and no pages would be read into the buffer pool for which there is no getpage. Thus, even though reading 32 discontinuous pages is slower than reading 32 contiguous pages, if the getpages skip 90% of the pages, using list prefetch instead of dynamic prefetch can reduce the number of I/Os by 90% and reduce the number of buffers used by 90%. Furthermore, IBM storage supports List Prefetch Optimizer to optimize the performance of list prefetch, making list prefetch generally superior to dynamic prefetch at different levels depending on the device type.

The list prefetch I/O quantity is the same as for dynamic prefetch. See Table 2-3 on page 15.

Much of DB2 uses list prefetch I/O, but two instances make up the most common cases. The first case is that of a disorganized index scan. When an index scan begins, Index Manager starts to use sequential detection and it also monitors the number of buffer misses. If six of the last 32 getpages were buffer misses, Index Manager switches to using list prefetch, and then it uses the non-leaf pages to determine which leaf pages to prefetch. It initially schedules two parallel I/Os of 128 KB each. Thereafter, Index Manager mimics dynamic prefetch, potentially scheduling two parallel I/Os if the query is I/O bound.

The second most common instance of list prefetch I/O is a RID list scan (described below). Usually the RID list is sorted, but occasionally DB2 does not sort the RID list. A sorted RID list scan is always skip-sequential or sometimes sequential, but the sequence of page numbers is never descending. A sorted RID list scan provides very good I/O performance.

DB2 REORG also uses list prefetch to read pages from an NPI. Other instances of list prefetch I/O in DB2 include:

- ▶ LOBs
- ▶ Incremental Copy
- ▶ Fast Log Apply
- ▶ Runstats table sampling

Of all of the different types of I/Os in the system, none benefit from High Performance IBM FICON® more than DB2 list prefetch. However, at the current time (July 2014) only IBM storage devices support High Performance FICON for DB2 list prefetch. Thus, to get the best performance for DB2 list prefetch, you need to use IBM storage, which supports List Prefetch Optimizer. For more information about this subject, see *DB2 for z/OS and List Prefetch Optimizer*, REDP-4862.

## 2.4 RID pools

RID list processing is used in list prefetch, multiple-index access, hybrid join, dynamic index ANDing, and for UPDATE statements where otherwise it would be possible to update the same row more than once. Each RID block consists of 32 KB of memory. These RID blocks are managed in a RID pool, whose default size is 400,000 KB. The failure of a RID list when the RID pool is exhausted can be severe since DB2 runtime logic could fail over to a relational scan of the table. However, DB2 is capable of overflowing a RID list to a workfile, in which case it does not fall back to a relational scan. Alternatively, some people prefer to restrict how much workfile space a RID list can consume. The MAXTEMPS\_RID DSNZPARM controls this limit. For example, specifying a value of 10,000 for the MAXTEMPS\_RID limits the number of RIDs that are allowed to be stored in the work file to 6,524,000 (about 312.5 MB). The default value of this DSNZPARM is 0, which means that RID lists are unlimited as to the amount of workfile space that they can use.

Thus, if you permit a RID list to overflow into a workfile, DB2 rarely needs to fall back to a table space scan. In some cases, you might see work file use increase as a result. Large memory with a large work file buffer pool avoids I/O for the RID blocks. Even if there is not enough memory to avoid work file I/O, work file I/O is far better than paging I/O for an oversized RID pool.

The DB2 optimizer considers RID list access at bind or prepare time only when it assumes that no more than 50% of the RID pool would be consumed by this access path, and that it would not scan more than 25% of the rows (index entries) in the table. Since this decision is based on bind time statistics, and the RID list failures are runtime issues, obviously RUNSTATS can play a large role in the success of RID list processing because the statistics collected by RUNSTATS help the DB2 optimizer to determine if it should choose list prefetch.

With the ability to overflow to the workfile database (into 32 KB pages), the failures are not as degrading to the application, but many of these overflows could impact other work in the 32 KB buffer pool supporting the workfiles, as well as the workfile utilization. So, failures should still be monitored to avoid the failures. The field MAX RID BLOCKS OVERFLOWED statistic tells you how many 32 KB blocks ended up in a workfile, and you can then calculate the total storage used.

Although the DB2 optimizer tries to avoid a RID list scan, if it knows that the number of RIDs in the RID list exceeds the preceding thresholds, it cannot always predict if this will happen because of the use of host variables and the lack of frequency statistics. For that reason, sometimes the RID lists become larger than one would like and performance suffers.

Refer to *Subsystem and Transaction Monitoring and Tuning with DB2 11 for z/OS*, SG24-8182 for more information about performance monitoring that is related to the RID pool.

## 2.5 Sequential detection versus a sorted RID list

The DB2 optimizer prefers to perform a RID list scan when the data is disorganized or when a screening predicate causes the rows to be skipped. Sorting the RID list ensures that a query does not do two getpages for the same data page and it avoids all synchronous I/O (except for indirect references). The RID list is built by first processing an index and evaluating index predicates to determine which rows need to be read. After sorting the RID list, DB2 can then begin to scan the pages that are represented by the RID list in skip sequential order using list prefetch.

There are four downsides of a sorted RID list scan:

- ▶ If a query requires that the result set be ordered by the index key, a sorted RID list scan requires that the result set be sorted back into index key order. The larger the result set is, the costlier the sort is.
- ▶ A sorted RID list scan requires that all predicates be reevaluated against the data row, including the predicates that are already evaluated against the index.
- ▶ A sorted RID list requires memory to manage the RID blocks.
- ▶ If a query does not fetch the entire result set, creating a sorted RID list can be a waste of time.

The DB2 optimizer avoids a sorted RID list scan if OPTIMIZE FOR 1 ROW is coded, and an access path exists that can avoid an ORDER BY sort. OPTIMIZE FOR n ROWS (where n is > 1) also discourages the optimizer from choosing a sorted RID list scan, but does not discourage the optimizer as strongly as OPTIMIZE FOR 1 ROW. Nor does the optimizer choose a sorted RID list if a dynamic scrollable cursor is used. However, if the application does not fetch every row from the result set and fails to specify OPTIMIZE FOR 1 ROW, creating a sorted RID list and incurring a sort of the result set may be very costly.

The DB2 optimizer chooses whether to create a sorted RID list that is based on certain characteristics of the query. The DB2 optimizer also tends to avoid creating a sorted RID list if the query needs to return the rows in sorted order where the order is already determined by the index key. Since a sorted RID list loses the ordering of the index key, a sort of the result set would be required. List prefetch I/O might be very fast, but if the size of the result is large, the cost of the sort could outweigh the advantages of list prefetch.

Furthermore, sometimes an application quits early without reading the entire result set, in which case the overhead of scanning the index to build the result set is a waste of time, even if the query did not require the result set in sorted order. But, ordinarily these types of *early out* applications are associated with queries that specify ORDER BY.

In contrast, when sequential detection is used, the index key ordering is preserved, and there is no danger of wasting time when the application quits early.

Since a RID list does not contain the keys that were first obtained from the index, Data Manager cannot be sure that the key did not change since the key was first determined by Index Manager to qualify according to the index predicates of the SQL statement. For that reason, Data Manager must reevaluate the index predicates against the row. In contrast, when using sequential detection, Data Manager does know if the key changed because it can compare the key returned by Index Manager to the key in the row. Predicates only need to be reapplied if the key changed.

With all of these advantages of sequential detection and disadvantages of a sorted RID list, why does the DB2 optimizer ever choose list prefetch? As mentioned already, sequential detection often results in lots of synchronous I/O—possibly numbering in the thousands or millions, in which case the user might cancel the query. A sorted RID list avoids all of those synchronous I/Os.

That said, there is one problem that neither sequential detection or a sorted RID list copes with very well. That problem is indirect references.

## 2.6 Indirect references

An *indirect reference* consists of a pointer row and an overflow record on a different page from the pointer. These occur when an update causes a row to grow and it cannot fit on the original page. Overflows are never explicitly prefetched. That is why it is important to try to avoid indirect references. Consider using PCTFREE FOR UPDATE or MAXROWS to avoid indirect references.

Another problem with using RID lists is the storage that is associated with RID blocks. See the previous chapter on RID list scans to understand more about managing the RID pool and RID list usage of work file storage.

## 2.7 Index insert I/O parallelism and conditional getpages

When a table space is defined with three or more indexes, DB2 uses parallelism to update the leaf pages of all but the first and last index. Suppose that there are at least three indexes. The first index is the cluster index. All other indexes are secondary indexes. All of the getpages are random. For the first secondary index, index manager does an index probe starting from the root page. When it gets to the leaf page, it does a conditional getpage. If the getpage is a buffer hit, the behavior is the same as though it were an unconditional getpage. However, if the getpage is a buffer miss, buffer manager schedules a prefetch engine to asynchronously read the page, and then it returns control to index manager. This process is repeated for each secondary index until the last one. For the last index, an unconditional getpage is done, which might result in a synchronous I/O. Then, index manager cycles back through the leaf pages for which the conditional getpages were misses and then does unconditional getpages. If each I/O takes the exact same amount of time, only one of the secondary index getpages gets suspended.

The number of these conditional getpages that caused prefetch to be scheduled is counted in QTSTNGT in the IFCID 2 records. The total number of getpages is incremented once for the conditional getpage and once for the unconditional getpage. However, when DB2 OMEGAMON Performance Expert calculates the buffer hit ratio, it subtracts the prefetch I/Os from the getpage count so that it appears as though there was just one random getpage buffer hit and no random getpage buffer misses.



Although the getpages are random, the prefetch I/O to read the one leaf page is classified as sequential prefetch. Thus, if you observe numerous sequential prefetch I/Os that read a single page, it is probably Index Insert I/O Parallelism.

## 2.8 Declared global temporary tables prefetch

Declared global temporary tables (DGTTs) use work files, but unlike sort work files, Data Manager uses a prefetch technique that is called *command prefetch*, whereby Data Manager prefetches ahead the next segment. In this case, the prefetch quantity is the segment size. This technique efficiently deals with the fact that segments can be discontinuous. Buffer Manager classifies these types of prefetch requests as dynamic prefetch, even though it is not really dynamic prefetch. RLSD does not apply to it and there is no progressive prefetch quantity. Nor does Data Manager ever prefetch a segment from the wrong table.

These command prefetch operations are counted in the DB2 IFCID 2 records as though they were dynamic prefetch, even though the read ahead algorithm and the prefetch quantity are completely different.





## Buffer pool writes

The best I/O is the one that is not executed. It is desirable for a logical I/O to not require a physical I/O.

For read, if the data resides in the buffer pool, the getpage request is much faster than waiting for physical I/O to complete for synchronous I/O.

For write, updates to data are made in the buffer pools, and these changes are then written to DASD when fixed buffer pool threshold values are reached or at system checkpoint interval.

In this chapter, we describe the asynchronous write of DB2 and the impact on setting the buffer pool variable thresholds, by looking at the following topics:

- ▶ Deferred writes
- ▶ Format writes
- ▶ Write I/O metrics
- ▶ I/O problems and data set level statistics

## 3.1 Deferred writes

Ordinarily DB2 writes data asynchronously using deferred writes. Despite it being asynchronous, deferred write I/O is often one of the most difficult performance problems for DB2, particularly when data is being replicated remotely, because then write I/O can be very slow. Often it is best to try to trickle the write I/Os, or spread them out evenly in between successive system checkpoints. In other words, the objective is to avoid the hiccup effect every time a system checkpoint occurs. These hiccups flood the I/O subsystem with writes.

However, there are trade-offs to be made. On the one hand, forcing DB2 to write updated or *dirty* pages to DASD quickly, soon after the getpages are released, helps to avoid these hiccups. But alternatively, if the same page is updated many times in a short interval, it is best to defer the write until the application is done updating the page. In other words, accumulating dirty pages in the buffer pool maximizes the chances to minimize the amount of write I/O. Besides the stress on the I/O subsystem caused by too many needless write I/Os, write I/Os cost CPU time. The deferred write engines are System z Integrated Information Processor (zIIP) eligible, but nevertheless they consume zIIP capacity.

The problem is that DB2 does not know whether a dirty page is or is not going to be updated again. The user needs to help DB2. DB2 offers two techniques for managing the write I/O. One is the checkpoint interval. The other is the variable deferred write thresholds, *deferred write queue threshold* (DWQT) and *vertical deferred write queue threshold* (VDWQT).

In most cases updated pages are written asynchronously from the buffer pool to DASD. Update pages are written synchronously when the immediate write threshold is reached, or when no deferred write engines are available, or when more than two checkpoints pass without a page being written.

DB2 maintains a chain of updated pages for each data set. This chain is managed on a least recently used (LRU) basis. When a write is scheduled, DB2 takes up to 128 pages off the queue on an LRU basis (depending on the threshold that triggered the write operation), sorts them by page number, and schedules write I/Os for up to 32 pages (that is, 128 KB) at a time, but with the additional constraint that the page span be less than or equal to 180 pages. This could result in a single write I/O for all 32 pages or in 32 I/Os of one page each. Sequential inserts would result in each I/O writing 128 KB.

The 180 page span limitation is imposed because the more pages that DB2 writes per I/O, the longer these pages remain latched. A thread that tries to update one of these pages gets suspended, and that time is accounted as Other Write I/O.

DB2 maintains a counter of the total number of dirty pages in the pool at any one time. DB2 adds one to this counter every time a page is updated for the first time and then checks two thresholds to see whether any writes should be done:

- ▶ The first threshold is the DWQT. This threshold applies to all updated pages in the buffer pool and is expressed as a percentage of all pages in the buffer pool. If the number of updated pages in the buffer pool exceeds this percentage, asynchronous writes are scheduled for up to 128 pages per data set until the number of dirty pages is less than the threshold. The default for DWQT is 30%, which should be the starting point for most users.
- ▶ The second threshold is the VDWQT. This threshold applies to all pages in a single data set. VDWQT allows the specification in either percentage or pages: VDWQT(%,P). VDWQT=0 sets an internal threshold of 40 pages per object for triggering asynchronous writes. Setting VDWQT=0 is the same as setting VDWQT=(0,40) for a 4 KB pool.

Sometimes people set VDWQT to 0 to trickle the writes as much as possible. However, you should keep in mind that for a large buffer pool, there is a huge difference between

VDWQT(0) and VDWQT(1). For example, given that VPSIZE is 100000, while VDWQT(0) is only 40 pages, VDWQT(1) is 1000 pages, which is 25 times greater than 40. Therefore, VDWQT(1) has a vastly greater chance than VDWQT(0) of being able to reduce the number of deferred writes when the same buffers are being updated repeatedly, while only requiring that you allow 1% of the buffer pool to remain dirty. That is usually a good trade-off. Nevertheless, the default with VPSIZE of 100000 would allow up to 5000 dirty pages for one data set before the deferred writes would commence.

If DWQT is triggered, DB2 continues to write until the number of updated pages in the pool is less than (DWQT-10)%. So for DWQT=30%, writes continue until the number of dirty pages in the pool is 20%. If the VDWQT is triggered, DB2 writes up to 128 pages of the data set and continues to do so until the number of dirty pages drops below VDWQT.

Write operations are also triggered by checkpoint intervals. Whenever a checkpoint occurs, all updated pages in the pool at the time are scheduled to be written to disk. If a page is in use at the time of the scheduled I/O, it can be left dirty until the next checkpoint occurs. Writes are also forced by a number of other actions such as a **STOP DATABASE** command or stopping DB2.

The checkpoint interval can be specified in terms of minutes or the number of log records. If dirty pages can be kept in the buffer pool long enough to avoid deferred writes, the checkpoint interval writes all of the dirty pages to DASD. However, the checkpoint interval tends to flood the DASD subsystem, and that affects the performance of read I/O in a negative way. It is better to enable the writes to be *trickled out* to DASD uniformly over time. Keeping VDWQT small causes the writes to trickle out. However, if it is too small, it can also cause the same page to be written out multiple times, which could exacerbate the problem.

Above all else, remember that storage hardware does not all perform the same, especially when it comes to write I/O, and remote replication has an enormous impact on write performance. Finally, do not underestimate the value of large memory in helping to reduce write I/O. The bigger the buffer pool is, the more dirty pages DB2 can safely keep in its buffer pool.

## 3.2 Format writes

The DB2 utilities use format writes to write new pages to a page set. If VPSEQTxVPSIZE is less than 320 MB, these I/Os write 256 KB. If it is greater than or equal to 320 MB, these I/Os write 512 KB.

It is not quite accurate to call this the I/O quantity because it is only the amount of data that the DB2 buffer manager provides to the system in one request. When using the FICON channel protocol, the system sometimes needs to divide a 512 KB request into two I/Os. Using High Performance FICON, the system can always read or write 512 KB in a single I/O. That is why High Performance FICON is so important to the performance of DB2 utilities.

That is also why it is important that VPSIZExVPSEQT be at least 320 MB. For example, using a large buffer pool, it has been demonstrated that High Performance FICON can increase the throughput for the DB2 LOAD utility by 50% when using a 4 KB size page size.

Table 2-1 on page 14 for the sequential prefetch quantity is also used for DB2 utilities to determine the format write I/O quantity.

### 3.3 Write I/O metrics

There are two metrics to focus on for understanding write I/O performance.

The first one is:

Number of page updates for each page written = buffer page updates / pages written

where “buffer page updates” is the total number page updates in a given statistics interval (trace field QBSTWS) and “pages written” is the total number of pages written in a given statistics interval (trace field QBSTPWS).

The second metric to focus on is:

Pages written per write I/O = pages written / write I/Os

where “pages written” is the total number of pages written in a given statistics interval (trace fields QBSTPWS) and write I/Os is the total of asynchronous plus synchronous writes in a given statistics interval (trace field QBSTIMW+QBSTWIO).

This is a measure of the write process. Remember that a write I/O consists of up to 32 pages (64 or 128 for a utility from a single data set where the span of pages (first page to last page) is less than or equal to 180.) The lowest value for this metric is one. A high value is better, though it should be noted that a higher value can lead to increased OTHER WRITE I/O suspension as well as page latch contention suspension because those pages being written out are unavailable and all updates to those pages must be suspended.

The above two metrics depend on the type of workload being processed and can be expected to vary across a normal working day as the mix of batch and transaction processing varies. The metrics are more likely to increase when you perform the following actions:

- ▶ Increase the number of rows per page, for example by using compression
- ▶ Increase the size of the buffer pool
- ▶ Increase the deferred write thresholds
- ▶ Increase the checkpoint interval

All these increase the likelihood of a page being updated more than once before being written, and also of another page being updated that is within the 180-page span limit for a write I/O. Of course, there can be other consequences of taking any of the above actions.

### 3.4 I/O problems and data set level statistics

When system programmers are analyzing problems in the I/O subsystem, they typically start with IBM RMF™. RMF provides a system point of view with useful I/O statistics at the volume level. These I/O statistics provide much detail such as a breakdown of I/O response times into connect time, disconnect time, pend time, and IOS queue time. However, the I/O response time statistics do not distinguish reads from writes, or DB2 synchronous I/Os from asynchronous I/Os, and nor do they show any direct information about the I/O quantity. You can only make inferences from the connect time and from the cache statistics in the I/O Cache Activity Report.

The SMF 42-6 records provide data set level I/O statistics, also with a breakdown of the I/O components. These statistics also show the block counts, aggregated by read versus writes. The SMF 42-5 records provide similar statistics aggregated by SMS storage class. The storage class can be used in a similar fashion to the WLM reporting class to aggregate performance statistics. One thing lacking, however, is statistics that distinguish the read I/O

response time from write response time. Both the SMF 42-5 and 42-6 statistics require something other than RMF to produce reports.

Still, the SMF 42-5 and 42-6 statistics do not distinguish between DB2 synchronous and asynchronous I/Os. That is why only DB2 itself can distinguish between the two. DB2 systems programmers know that neither RMF or the 42-6 records are the best place to start when it comes to diagnosing I/O problems. The best place to start is with the DB2 accounting records. If there is a slow down, the accounting records show which specific transactions are being affected, or whether they are affected at all. The accounting records show how long the transactions are waiting on synchronous I/O or prefetch I/O, deferred write I/Os, and so on. The accounting records also contain the average synchronous I/O delays, including z/OS dispatch delays (which are not included in the I/O response times reported by RMF). When DB2 prefetch is used, the DB2 accounting and the system level DB2 statistics also help distinguish whether the pages being prefetched are contiguous, and exactly how many pages are being prefetched per I/O. Determining a clear profile of the prefetch I/Os is still not a perfect science using these methods, but DB2 provides much information that z/OS by itself does not provide.

If you needed to dig down even deeper, DB2 provides still more information. z/OS can trace I/Os, but few people know how to analyze them. DB2 provides much simpler trace records, namely the IFCID 6, 7, 8, 9 trace records. Together these IFCID traces show the start and stop times for reads and writes for each page set, and they show the specific page numbers. For example, for a DB2 list prefetch I/O, you can see the specific distances between the pages.

DB2 also provides some basic data set level statistics on the **DISPLAY BUFFERPOOL LSTATS** command. LSTATS shows the number of synchronous I/Os and asynchronous read and write delays by data set. These statistics can also be collected in IFCID 199 records and reported by OMEGAMON DB2 Performance Expert.







## Data sharing

No description of buffer pool tuning would be complete without considering data sharing and group buffer pools (GBPs). GBPs are stored in a coupling facility structure that is accessible from all of the members of a data sharing group. The size of that structure is what determines the size of the GBP.

This chapter provides, describes, discusses, or contains the following topics:

- ▶ GBPCACHE
- ▶ GBP cast out
- ▶ GBP write-around

## 4.1 GBPCACHE

You can specify **GBPCACHE** attribute both at the page set level and at the buffer pool level. This parameter specifies what pages of a page set are written to the group buffer pool in a data sharing environment when the page set becomes *GBP dependent*. Being GBP dependent means that inter DB2 read/write interest exists. At the page set level you can specify **CHANGED**, **ALL**, **SYSTEM** or **NONE** for this attribute. (If you specify **GBPCACHE** for a workfile, the parameter is ignored):

<b>CHANGED</b>	Is the default. This is the value that IBM recommends for most page sets. <b>CHANGED</b> means that only modified pages are stored in the GBP. (However, after a page gets cast out from the coupling facility to DASD, it remains on the GBP LRU chain until the buffer is stolen.)
<b>ALL</b>	Indicates that pages are to be cached as they are read in from DASD, except in the case of a single updating DB2 when no other DB2 members have any interest in the page set. To use <b>ALL</b> effectively, the structure needs to be very large relative to the size of the local buffer pools because if the local buffer pool hit ratio is high, using <b>ALL</b> merely adds to the CPU time without avoiding much I/O.
<b>SYSTEM</b>	Indicates that only changed system pages with a LOB table space are to be cached in the GBP. A system page is a space map page or any other page that does not contain actual data values. <b>SYSTEM</b> is the default for LOB table spaces. This is the value that IBM recommends for LOB table spaces.
<b>NONE</b>	Indicates that no pages are to be cached in the GBP, in which case the GBP is only used for cross-invalidation.

If a page set becomes GBP dependent and is defined with **GBPCACHE SYSTEM** or **NONE**, a Commit SQL statement causes the changed pages to be written directly to DASD. Commits takes longer, but this avoids flooding the group buffer pool. Thus, if it is unlikely that **GBPCACHE CHANGED** results in GBP read buffer hits (as is often the case with LOB table spaces), **GBPCACHE NONE** might be preferable. For example, although using **GBPCACHE NONE** for LOBs could elongate the LOB commit response times, doing so could help improve the GBP read hit ratio for non-LOB table spaces.

**CHANGED** is the default. This is the value that IBM recommends for most page sets. **CHANGED** means that only modified pages are written to the GBP. (However, after a page gets cast out from the coupling facility to DASD, it remains on the GBP LRU chain until the buffer is stolen.) Do not specify **GBPCACHE CHANGED** for LOBs if they are defined as **NOT LOGGED**.

At the buffer pool level, you can specify either **YES** or **NO**. **YES** is the default. Any no-data-caching attribute that is specified at the page set level takes precedence over a caching specification at the GBP level. For example, if you specify **GBPCACHE CHANGED** or **ALL** for the page set and specify **GBPCACHE(NO)** at the GBP level, **GBPCACHE(NO)** takes precedence. Or, if you specify **GBPCACHE NONE** for the page set and **GBPCACHE(YES)** for the GBP, **GBPCACHE NONE** takes precedence.

### 4.1.1 AUTOREC

**AUTOREC** specifies whether automatic recover by DB2 takes place when a structure failure occurs or when the connectivity of all members of the group to the group buffer pool is lost. **AUTOREC** can be set to **YES** or **NO**.

## 4.1.2 RATIO

RATIO changes the wanted ratio of the number of directory entries to the number of data pages in the GBP; that is, how many directory entries exist for each data page. The actual number of directory entries and data pages that are allocated depends on the size of the coupling facility structure, which is specified in the coupling facility policy definitions (CFRM policy). The default value is 5.

## 4.2 GBP cast out

*Cast out* is the process of writing pages from the GBP to DASD. Briefly stated, there are three GBP parameters to consider that are similar to the deferred write thresholds and the checkpoint interval for local buffer pools. CLASST is for a GBP what VDWQT is for a local buffer pool. GBPOOLT is for a GBP what DWQT is for a local buffer pool. As with the local buffer pool parameters, the default for CLASST is 5%, and the default for GBPOOLT is 30%.

All of the recommendations regarding VDWQT and DWQT apply equally to CLASST and GBPOOLT. In fact, it is even more important that CLASST drive the cast out process because cast out is not limited to a range of 180 pages and because cast out is also capable of writing up to 64 pages per I/O. Thus, cast out processing generates a lot less I/O than deferred write processing. Cast out processing also does not cause any Other Write I/O suspension because it does not latch any buffers in the local buffer pool.

As with VDWQT, you can also specify the CLASST threshold in terms of the absolute number of pages instead of a percentage. For example, CLASST(0,1000) means 1000 pages. You need to set the value in terms of absolute pages if 1% is too large for you. (1% of a very large GBP could be a very large number of buffers.) The maximum allowed number for the absolute number of pages is 32767. A value 0 - 32 is interpreted to mean 32 pages. Do not specify such a small number if you have some pages that are frequently updated because doing so causes extra cast I/Os.

GBPCHKPT is the time interval, in minutes, between successive checkpoints of the GBP. The default value is 4 minutes. The more frequently that checkpoints are taken, the less time it takes to recover the GBP if the coupling facility fails.

## 4.3 GBP write-around

There is one other aspect of cast out processing that is called *GBP write-around*. This is designed to keep data being cached in the GBP that is used by online transactions from being overrun by batch inserts that are writing new pages. This can help avoid application stalls that could otherwise occur during large concurrent batch insert/update/deletes. In severe cases, this can result in pages being written to the logical page list (LPL) and requires recovery execution.

Only deferred writes are eligible for GBP write-around. Commits and forced writes continue to write to the coupling facility. Also, the only pages that are eligible for write-around are pages that are registered in the coupling facility by the DB2 member who wrote the pages to the GBP and are not registered by any other DB2 member; in other words, pages that are not contained in the local buffer pools of the other members.

However, even if a page is eligible for write-around, DB2 first writes the page conditionally to the GBP. Conditional write means that if the page is already cached in the GBP, the write is allowed to proceed; if the page is not already cached in the GBP, the write fails and it is written via the local buffer to DASD.

A **DISPLAY GROUPBUFFERPOOL** command with the MDETAIL option shows the number of pages written via write-around processing in the DSNB777I informational message that is displayed as part of the output. OMEGAMON Performance Expert also has a counter called PAGES IN WRITE-AROUND.

GBP write-around requires an IBM zEnterprise® 196 (z196) or zEnterprise 114 (z114) server with CFLEVEL 17 and a microcode load (MCL). It also requires z/OS V2.1, or V1.12, or V1.13 with the PTF for APAR OA40966.

Here is how GBP write-around works. Say that CLASST is defaulted to 5%. When the number of changed pages in a class queue reaches 5%, DB2 begins to cast out the pages to DASD. At the same time that cast out is trying to write pages to DASD, large batch applications or DB2 utilities could be writing more changed pages to the GBP. If the rate of writing pages to the GBP is faster than DB2 can cast out the pages to DASD, the percentage of changed pages could well exceed 5%. When 20% of a class castout queue is occupied by changed pages, the DB2 deferred write engines stop writing more changed eligible pages to that castout queue and begin to write pages directly from the local buffer pool to DASD until the threshold drops to 10%. This 20% number is hardcoded; it cannot be changed.

Similarly, when 50% of the GBP contains modified pages, DB2 stops writing any eligible pages to the GBP until the threshold drops to 40%.

The benefit of GBP write-around is that DB2 automatically detects the flooding of writes to the GBP, and automatically responds by dynamically switching to the GBP write-around protocol for those objects that are causing the heaviest write activity. The other GBP-dependent objects using that GBP do not notice any difference in GBP access protocols; only the objects with the heaviest write activity are affected. When the GBP storage shortage has been relieved, DB2 resorts back to normal GBP write activity for all GBP-dependent objects.

## 4.4 Setting up your CFRM policies

The preceding information is just the basics on using GBPs, but there are more parameters that are associated with defining your CFRM policies. For more information about this subject, refer to *DB2 11 for z/OS Data Sharing: Planning and Administration*, SC19-4055; *Subsystem and Transaction Monitoring and Tuning with DB2 11 for z/OS*, SG24-8182; and *z/OS Parallel Sysplex Configuration Overview*, SG24-6485.



# Workfiles

This chapter describes the workfiles and their related buffer pools.

Workfiles table spaces, mostly defined to the DSNDB07 database, are used by for sort, star join, trigger, created temporary tables, view materialization, nested table expression, merge join, non-correlated subquery, sparse index, temporary tables, and so on.

This chapter contains the following topics:

- ▶ Controlling the growth of workfiles
- ▶ Performance considerations for workfiles
- ▶ Workfile buffer pools

## 5.1 Controlling the growth of workfiles

Some problems exist in having declared global temporary tables (DGTTs) share the same database as other work files because DB2 manages DASD storage differently for these types of tables. A normal workfile, such as a sort workfile, can span table spaces but DGTT workfiles cannot. Users want to be able to control the amount of workfile space that is used by each type of workfile. So, DB2 offers some controls to steer each object type to a different type of table space.

For non-DGTT work files such as sort files, DB2 prefers a DB2 managed table space that is defined with SECQTY 0 or a user managed table space (regardless of their secondary space units). For DGTT workfiles, DB2 prefers a DB2 managed table space that is defined with SECQTY>0 or -1 (or omitted). If a favored table space does not have room or is unavailable for any reason, DB2 still tries to use a non-favored table space, which could lead to sharing of space between non-DGTT and DGTT applications. This can be monitored by the following statistics fields:

QISTWFP3/4= #times DGTT had to be allocated from 0 secqty 4K/32K work file table space

QISTWFP5/6= #times non DGTT had to be allocated from >0 secqty 4k/32K work file table space

If you would rather fail the allocations with SQLCODE -904 rather than allowing the allocations to spill over into a non-favored table space, you can specify the online DSNZPARM WFDBSEP=YES.

Online DSNZPARM MAXTEMPS controls how much space in workfile database an agent can use (or a family of parallel tasks in case of query CP parallelism). The usage can be for sort workfiles, created global temporary tables (CGTTs), DGTTs, scrollable cursors result tables, trigger transition tables, and so on. The DSNZPARM is not granulated to any one type of these uses, instead it tracks the overall use by the agent.

Still, some people want to limit the growth of DGTTs, while allowing large sorts. MAXTEMPS is not good enough because it applies to both DGTTs and sorts. The problem is compounded by the fact that a small secondary quantity does not necessarily limit the growth when using system-managed storage because the system tries to consolidate contiguous extents. For example, specifying SECQTY 1 does not limit the growth to 123 extents of 1 KB each because if the system happens to consolidate the secondary extents, the data set could continue to grow beyond 123 KB. You can limit the size of a DGTT to as little as 1 GB by setting MAXPARTITIONS=1 and DSSIZE=1 GB, and you can use system managed storage with it. Conversely, you can also overcome the limit of 64 GB for a segmented table space if you want to, by specifying a MAXPARTITIONS greater than 1. Along with MAXPARTITIONS, you can set DSSIZE up to 256 GB. Furthermore, you can create table spaces larger than 2 GB with a single large data set rather than chopping up the table space into 2 GB data sets.

For more information about work file separation, see the following site:

<http://www.ibm.com/support/docview.wss?uid=isg1III14587&myns=apar&mynp=DOCTYP>

Even with these separations, the primary quantity specification could affect performance to some extent. DB2 managed workfiles' primary quantity can make a difference even when SECQTY= 0. DB2 might try to extend the table space even if the size is below 2 GB, which can lead to unnecessary space growth and CPU cycles, but at the same time you want to make the primary extent size as large as possible for efficient use of the table spaces.

DB2 11 contains some statistics that can be useful in tracking overall system utilization by sorting and DGTTS, as well as individual offenders. Having the ability to track the total storage configured versus the maximum used for DGTTS, sorts, and the grand total can help you:

- ▶ Better plan for capacity upgrades
- ▶ Trend usage and growth over time and correlate buffer pool statistics to the overall utilization of workfiles
- ▶ Monitor for drastic changes during application or release migrations that warrant further investigation

There are also DSNZPARMs for system and thread level monitoring that can issue warnings based on the percentage of the workfile or DGTT storage that is consumed, instead of just abending the thread. If the WFDBSEP DSNZPARM is set to YES, the threshold applies to both the DGTT and workfile storage individually, so it takes effect if either category of storage is infringed upon. If WFDBSEP = NO, the thresholds apply to the total storage configured. These DSNZPARMS are:

- ▶ **WFSTGUSE\_SYSTEM\_THRESHOLD**

% of workfile storage consumed at the system level when DB2 issues message:

```
DSNI052I csect-name AN AGENT HAS EXCEEDED THE WARNING THRESHOLD FOR STORAGE USE  
IN WORK FILE DATABASE.....FOR DECLARED GLOBAL TEMP TABLES or WORK FILES
```

- ▶ **WFSTGUSE\_AGENT\_THRESHOLD**

% of workfile storage consumed by a single agent when DB2 issues message:

```
DSNI053I csect-name THE DB2 SUBSYSTEM HAS EXCEEDED THE THRESHOLD FOR STORAGE  
USE IN WORK FILE DATABASE...
```

The IBM Redbooks publication *DB2 11 for z/OS Technical Overview*, SG24-8180, has details on these settings and the nuances of how the percentage of total storage is calculated (like needing all of the workfile data sets to be open to calculate it), and frequency of the messages.

These two DSNZPARMs are a more gentle and proactive approach to monitoring out-of-bounds conditions than using MAXTEMPS. This parameter would terminate any agent who used more storage than was specified in the installation parameter with an SQLCODE -904 and 00C90305 reason code. This is good as a last resort, but some customers are hesitant to set it for fear of disrupting service. To track whether service is being denied, instead you can now monitor the number of times that the MAXTEMPS parameter was hit by the 'NUMBER OF LIMIT EXCEEDED' field (QISTWFNE).

## 5.2 Performance considerations for workfiles

For any workfile record less than 100 bytes, DB2 prefers to use a 4 KB page table space. For larger records it prefers a 32 KB page table space. This, of course, affects the number and size of the workfile table spaces you must create for each page size. You can assess the need for each page size based on their use. IFCID 002 statistics shows the current storage in use broken down by page size. You should avoid having the larger record sizes needing to spill over to 4 KB pages and vice versa because this could hurt the performance.

HyperPAV is recommended to avoid IO operations queued up by the IO supervisor (IOSQ) time with your work files. If you observe IOSQ time for your workfiles even with HyperPAV (which is unlikely), you need to spread your workfiles across Logical Control Units. However, if

you see I/O disconnect time for your work files, it means that you also need to spread the workfiles across physical disk extent pools.

## 5.3 Workfile buffer pools

In some shops, the customer aggressively tries to ensure that all of the sorting and logical workfiles are contained in memory, in the sort pool and buffer pools, without spilling over to DASD. This would be a highly performance optimized design, which depends on a predictable workload in terms of number of concurrent sorts and the size of the sorts. However, it is likely not realistic if there are temporary tables or static scrollable cursors in the environment. The settings would look something like the following values:

- ▶ VPSEQT= 100%
- ▶ DWQT = 70-90%
- ▶ VDWQT= 60-90%

The downside here is that if you do see I/Os, the strategy is not working and it is likely that you hit other buffer pool thresholds like prefetch disabled, or even the data manager threshold, thus destroying the performance and affecting system stability.

A better approach is a more conservative one, with robust defensive settings. Where there is a mix of sort, sparse indexes, temporary tables, and unpredictable sort sizes, use the following values:

- ▶ VPSEQT= 90%
- ▶ DWQT = 50%
- ▶ VDWQT= 10%

With this approach, some performance is sacrificed for stability and versatility. Be ready to adjust up or down VPSEQT depending on the synchronous I/O activity in the sort BP.

If there is no synchronous I/O activity and also if QXSIWF<sup>1</sup> (Sparse Index Built Workfile) is > 0, you might want to leave it at 90%. If QXSIWF = 0, you can increase VPSEQT to 95 or higher.

But if synchronous I/O activity on sort buffer pool is high, check QXSISTOR<sup>2</sup>. If that is > 0, you either reduce MXDTCACH or reduce other pools (buffer pools or other memory allocations), or add more memory. If QXSISTOR=0, you might want to increase MXDTCACH or reduce VPSEQT.

QXSIWF and QXSISTOR are reported under Statistics - Miscellaneous by IBM Tivoli® OMEGAMON XE for DB2 Performance Monitor.

If you are concerned about reducing VPSEQT and its impact on sort, you can increase VPSIZE of the sort BP while also reducing VPSEQT so that the net for sequential sort activity is approximately the same number of buffer pool pages.

---

<sup>1</sup> The number of times that sparse-index built a physical work file for probing.

<sup>2</sup> The number of times that sparse index was disabled because of insufficient storage.





## Assigning page sets to buffer pools

This chapter describes the assignment of DB2 objects to different buffer pools, the buffer pool sizes, and their threshold settings.

This chapter provides, describes, discusses, or contains the following topics:

- ▶ Multiple buffer pools
- ▶ Choosing a page size

## 6.1 Multiple buffer pools

Indexes, data, large objects (LOBs), and work files each have different considerations, partly because the pages contain a different amount of information. Because index keys are small, one index page contains much information, and because of that, the likelihood of the page being re-referenced frequently is very high. Since rows are larger than keys, each data page contains less information and is less likely to be re-referenced frequently. LOBs are even larger. No two LOBs can occupy the same page and therefore the likelihood of a LOB page being referenced is very small. That is the motivation for separating indexes, data, and LOBs in separate buffer pools. If LOB pages are unlikely to be re-referenced, the LOB pages should be assigned to a small buffer pool. Data buffer pools should only be given buffers that are not needed for index buffer pools to achieve a very high index buffer hit ratio.

Another reason for separating some page sets into different buffer pools is to assign different buffer pool parameters. For example, table spaces that are mostly accessed sequentially can be assigned a buffer pool with a 90% or even 99% value for VPSEQT. This avoids 20% of the buffer pool to become *stale*, just because at one point there were some random getpages.

The same principle could be applied to workfiles since they are predominantly accessed sequentially. However, if not using WFDBSEP=YES, you would need random space in the workfiles for declared global temporary table (DGTT) access. Also, with DB2 11 workfiles should not be defined with 100% or 99% VPSEQT because DB2 11 uses workfiles for sparse indexing when in-memory caching and sorts overflow. When using workfiles for sparse indexing and similar, the access appears as random and VPSEQT would therefore affect that access. VPSEQT=100 would be dangerous in the sort buffer pool if there are numerous sparse indexes that spill out of MXDTCACH to the sort workfile.

Sometimes it is also useful to assign buffer pools based on update or insert activity. As noted previously, sequential inserts can devastate the buffer hit ratio for other page sets that share the buffer pool. Therefore, it is useful to place page sets that are used for sequential inserts into a separate buffer pool.

It is useful to specify PGFIX(YES) for I/O intensive buffer pools, particularly with small buffers such as those used for LOBs. Use PGFIX(NO) for buffer pools with a low getpage rate. If a buffer pool has a high getpage rate but a low I/O rate, it does not matter what you say for the **PGFIX** parameter. So, the optimal choice for PGFIX depends solely on the I/O rate, not the getpage rate. However, it is a good idea to use 1 MB page frames for I/O intensive page sets, whether you use PGFIX(YES).

## 6.2 Choosing a page size

Most page sets should use a 4 KB page size, which is the smallest page size that is supported by DB2, because 4 KB helps to maximize the random buffer hit ratio. Some exceptions are described below.

### 6.2.1 Indexes

The main reason for using a large page size for indexes is to minimize the number of index splits. Index splits cause synchronous log write I/O in a data sharing environment. Alternatively, a large page size tends to cause more deferred write I/O. Deferred writes are asynchronous, but oftentimes deferred write I/O is a major cause of poor I/O performance, in

which case it could affect the random buffer hit ratio. If the deferred writes are so slow that the dirty page count reaches the critical prefetch threshold of 90%, prefetch becomes disabled.

Even worse things can happen if the dirty page count reaches the data management threshold of 95%.

Use a large index page size for other reasons:

- ▶ To use index compression
- ▶ To reduce the number of index levels, and therefore to reduce the number of getpages
- ▶ To improve the performance of sequential scans

Free space is also one of the considerations that affects the buffer hit ratio. PCTFREE is the amount of space that is reserved on each page by the utilities. The default value for indexes is 10%.

Disorganized indexes often have about 25% free space because after a page is split, two pages are created and they each have 50% free space. Sometimes Index Manager does asymmetric page splits, but it does that for a reason; if the keys are ascending or descending, there is no reason to divide the free space equally.

It is recommended that you use the default PCTFREE value of 10% and do not use REORG for redistributing or re-establishing free space. Pages in an index that become disorganized by index splits average 25% free space. If you REORG such an index, you reduce the amount of free space, only to cause more index splits.

You can, however, remove all of the free space using ALTER and REORG if you do not expect the index to be updated again; doing so could improve the buffer hit ratio and save DASD space.

Choose a storage device such as the IBM DS8870 that optimizes list prefetch performance to avoid needing to keep your indexes organized.

## 6.2.2 Table spaces

The main reason for using a large page size for table spaces is to support large rows. Since a row cannot span pages, some space is often wasted at the end of each page. As a rule of thumb with variable length row sizes, expect the amount of wasted space to be the equivalent of half a row. For example, with 10 rows per page, expect to waste 5%. A larger page size that stores more rows reduces the space wastage, but it might hurt the random buffer hit ratio.

Another reason to use a large page size is to maximize the performance of DB2 prefetch and sequential writes. However, remember that one data page can contain at most 255 rows.

Free space is another consideration. PCTFREE is the amount of space reserved on each page by LOAD and REORG. PCTFREE is often used to maintain a high cluster ratio regarding the cluster index. The default value for table spaces is 5%. A high cluster ratio is necessary to avoid synchronous I/Os when using an index to locate and fetch rows within a specific range of key values. REORG can be used to ensure a high cluster ratio for the cluster index, but REORG is expensive and it will not improve the cluster ratio for most secondary indexes. If you choose a storage device that optimizes list prefetch performance, you might be able to use sorted record identifier (RID) list scans effectively and then you might be able to avoid these REORGs.

Another problem with table spaces is indirect references, which are associated with variable length rows. Compression and VARCHAR or VARGRAPHIC columns can result in variable length rows. Indirect references are created when a row is updated to a larger row size and it

can no longer fit on its original page. The original location of the row becomes a pointer to a new page, which is called an *overflow*. The pointer/overflow relationship is called an *indirect reference*. These cause more getpages and more synchronous I/Os. The most common cause of indirect references is nullable VARCHAR columns because applications sometimes like to insert null values and then supply the column values via update.

The only way to clean up indirect references when they occur is to use REORG. Sometimes PCTFREE is sufficient to prevent future indirect references, but inserts might consume the free space. That is why DB2 supports a second parameter, **PCTFREE FOR UPDATE** to reserve some space on each page that inserts cannot use. A value of -1 for this parameter is the autonomic option, whereby Data Manager derives a percentage of space to reserve for updates. You can also specify this parameter as a default in your DSNZPARMs.

## 6.2.3 Large objects

LOBs table spaces are used for columns that are too large to fit in the row, whose maximum size is 32 KB. When DB2 reads a LOB consisting of a single page, it does a random getpage, which can result in a synchronous I/O. For all subsequent pages in the LOB, the getpages are classified as sequential and DB2 uses list prefetch. Thus, all but the first LOB page is governed by VPSEQT.

Because each LOB page is connected with a single row, LOBs tend to be buffer unfriendly, meaning that the likelihood of a random buffer hit is small even if the VPSIZE is large. That is why it is usually best to assign LOB table spaces to a different buffer pool from other page sets, and dedicate a small buffer pool for it. Remember to size the buffer pool to accommodate concurrent prefetch I/O.

Choosing the page size for a LOB table space involves some trade-offs. To maximize the likelihood of fitting the entire LOB in one page, you could choose a 32 KB page size. Alternatively, if many of the LOBs are much smaller than 32 KB, a 32 KB page size wastes space in the LOB table space. Therefore, you must trade off DASD space utilization and I/O performance. To make this trade-off, you must analyze your LOB size distribution and judge the cost of DASD storage.

Also consider using inline LOBs if most of the LOBs are small. You chose LOB over VARCHAR because at least one column was larger than 32 KB. In most cases where you choose to use inline LOBs, you want to use a large page size for the base table space. For information about best practices for using inline LOBs, see the following site:

[https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/W7c1e3103820b\\_4d9e\\_add6\\_b5ade3280dcb/page/Inline+LOBs+%28Large+Objects%29](https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/W7c1e3103820b_4d9e_add6_b5ade3280dcb/page/Inline+LOBs+%28Large+Objects%29)



# Improving CPU performance

This chapter describes the most important attributes of buffer pools that are related to CPU usage.

The following topics are covered in this chapter:

- ▶ PGSTEAL(NONE/FIFO)
- ▶ PGFIX(YES)
- ▶ z/OS frame size

## 7.1 PGSTEAL(NONE/FIFO)

**PGSTEAL** is the buffer pool parameter that controls some of the buffer steal algorithms. Normally you should use the **PGSTEAL(LRU)** option, which is the default.

In rare cases you might expect page sets to remain in-memory, without any I/O at all. In such cases, it is desirable to avoid the CPU overhead of least recently used (LRU) management and the overhead of scheduling a prefetch engine, which would find all of the pages in the buffer pool anyway. **PGSTEAL(NONE)** is the best way to do that. When **PGSTEAL(NONE)** is used, opening of a page set causes the page set to be loaded into the buffer pool using sequential prefetch. Thereafter, prefetch will never again be used for that object.

If the aggregate of all of the objects used for a given buffer pool are too large to fit, unpredictable performance can occur. Therefore, you need to ensure that the buffer pool is large enough, and you need to monitor the size of the page sets if they are growing. Adjust the buffer pool size accordingly, and stop using **PGSTEAL(NONE)** if you start to see much synchronous I/O.

You can also use **PGSTEAL(FIFO)**. As with **PGSTEAL(NONE)**, **PGSTEAL(FIFO)** avoids the CPU overhead of LRU management. In the past, **PGSTEAL(FIFO)** was used in combination with **VPSEQT(0)**. **VPSEQT(0)** disables prefetching. The problem with **VPSEQT(0)**, however, is that when you disable prefetching this way, when you need to repopulate the buffer pool (such as after a DB2 restart), it is repopulated using synchronous I/O. Alternatively, if you use **PGSTEAL(FIFO)** without **VPSEQT(0)**, you will not save the CPU overhead of scheduling of prefetch engines.

## 7.2 PGFIX(YES)

When I/O is done using a DB2 buffer, z/OS requires that the buffer be *page fixed* or *pinned* in memory so that it does not get paged out to auxiliary storage during the I/O operation. But there is a CPU cost to page fixing and then page freeing the buffer after the I/O completes. This is especially costly with prefetch I/O because there is a cost for each buffer. To avoid these costs, use long-term page fixing (**PGFIX(YES)**) for any pool with a high buffer intensity.

The buffer intensity = (pages read + pages written) / buffer pool size

where:

pages read = total number of pages read in a given statistics interval, including synchronous read and all prefetch I/O

(trace fields QBSTRIO + QBSTSP + QBSTDPP + QBSTLPP)

pages written = total number of pages written in a given statistics interval including synchronous write and deferred write.

(trace field QBSTPWS)

When using data sharing, you should also include pages read from the buffer pool either synchronously or by prefetch. These statistics are in the Group Buffer Pool report:

- ▶ SYN.READ(XI)-DATA RETURNED (QBLXID)
- ▶ SYN.READ(NF)-DATA RETURNED (QBGLMD)
- ▶ NUMBER OF PAGES RETR.FROM GBP (QBGLAY)

It can be catastrophic to the performance of a production system if it does not have sufficient real memory to support the defined buffer pool sizes. If PGFIX(NO) is used and there is insufficient memory, your buffers may be paged out to DASD. Then each time a getpage miss needs to steal a buffer, z/OS pages the buffer back into real memory, only to have DB2 read a new page from the database into that buffer. Thus, each getpage miss suffers from two I/Os, of which two are synchronous, in addition to the asynchronous page-out operation. This is expensive in terms of both CPU time and response time. DB2 provides statistics to track whether page-in operations are occurring. It has sometimes been said that less than 1% of the getpages should suffer from page-ins. To be more precise, 1% may be acceptable, but you should see 0% most of the time. If you consistently see page-ins, you have oversized your buffer pool. If you see this, you should either shrink the buffer pool size, or you should purchase more memory.

If you succeed at avoiding the paging I/O operations, the virtual pages of your buffer pools always occupy real memory. In that case, you can use PGFIX(YES) to save on CPU consumption. It cannot be emphasized too strongly that using PGFIX(YES) must be agreed upon with the person who manages the z/OS storage for the logical partition (LPAR) because running out of storage can have catastrophic effects for the whole of z/OS. By page fixing the DB2 buffer pool, you may inadvertently cause other DB2 storage or other application storage to do paging, just because you oversized your DB2 buffer pool. Whether or not you use PGFIX(YES), you should never oversize your buffer pool, meaning that the buffer pool should never be so large that it is the cause of DB2 or some other application doing paging.

One disadvantage of using PGFIX(YES) to be aware of is that the buffer pool consumes real storage even if the buffer pool becomes dormant. Had you used PGFIX(NO) with a dormant buffer pool, the buffers could get paged out and then paged back in when the buffer pool becomes active again. If that is what you want, use PGFIX(NO). However, if you are willing to do a little micro managing, you could shrink the buffer pool size when the buffer pool becomes dormant, and then increase the size when it becomes busy again. Better yet, AUTOSIZE(YES) might be the best way to manage buffer pools whose usage intensity varies significantly throughout the day or week.

## 7.3 z/OS frame size

In addition to basic 4 KB frames, z/OS supports large-size frames that can be either 1 MB or 2 GB. Using 1 MB page frames saves CPU consumption by improving the hit ratio of the system translation lookaside buffer (TLB). Ordinarily, 1 MB frames can be non-pageable and can be used with PGFIX(YES). To use 1 MB or 2 GB frames, you must specify the LFAREA system parameter when you IPL the system. By using LFAREA, you can specify either some 1 MB frames or 2 GB frames, or a mixture of the two.

If LFAREA is defined, DB2 automatically tries to use the 1 MB frames for buffer pools defined with PGFIX(YES). By default, when LFAREA is defined, DB2 tries to use 1 MB frames. You can override the default by specifying FRAMESIZE(4 K) or FRAMESIZE(2 GB) for the DB2 buffer pool. You can also explicitly specify FRAMESIZE(1 MB).

Frames that are 2 GB require the zEC12 processor. They also require that the buffer pool be non-pageable (that is, PGFIX(YES)). Currently, performance advantages of 2 GB frames have not been observed. However, it is expected that as memory sizes become much larger, 2 GB frames can become more advantageous.

Frames that are 1 MB can also be pageable if the processor is configured with Flash Express memory and if the z/OS release is V1R13 (with the requisite PTFs) or higher. The zEC12 processor is required to use Flash Express. For buffer pools with high I/O intensity, IBM

recommends using non-pageable buffer pools with PGFIX(YES). See 7.2, “PGFIX(YES)” on page 42.

With PGFIX(NO), if Flash Express is configured in the processor, DB2 tries to use 1 MB frames for the buffer pool control blocks and they are pageable. (If there is insufficient memory, they page out to Flash Express.) However, with PGFIX(NO), DB2 never uses large frames for the buffers themselves. The **FRAMESIZE** parameter is ignored with PGFIX(NO).

There are many reasons to purchase Flash Express for your processor, such as to improve paging performance or to make it less disruptive to take system dumps. However, IBM does not recommend purchasing Flash Express for the specific purpose of using them for DB2 buffer pools. (IBM recommends that you use PGFIX(YES), which does not need Flash Express.) Alternatively, given that you have Flash Express installed, if you must use pageable buffer pools with PGFIX(NO) for any reason, IBM recommends using 1 MB frames. Still, IBM recommends that you not oversize your buffer pools. If your buffer pools are paging, it means that your buffer pools are too large, meaning that there is not enough real storage to support the buffer pool size.





## Index and data organization

Running REORG can cause some level of unavailability. You must consider several factors before you reorganize your data because new functions allow REORG avoidance.

Data reorganization is still advisable when data is in REORG-pending or Advisory REORG-pending status while several DB2 enhancements have diminished the impact of disorganization and others have removed the reasons for it.

If application performance is not degraded, you might not need to reorganize data. Even when some statistics indicate that data is becoming disorganized, a REORG utility job is not always required, unless the lack of organization exceeds a specified threshold.

In this section, we examine the reasons for reorganization of indexes and data.

This chapter contains the following topics:

- ▶ Index reorganization
- ▶ Data reorganization

## 8.1 Index reorganization

Let us review some of the supposed reasons for keeping an index organized:

1. Enable dynamic prefetch to be used instead of list prefetch for an index scan.
2. Re-establish some free space to ensure that newly inserted keys and record identifiers (RIDs) will not disrupt the ability of DB2 to use dynamic prefetch for an index scan.
3. Re-establish some free space to ensure that newly inserted keys and RIDs will not suffer poor performance due to index splits, which cause synchronous writes in a data sharing environment.
4. Clean up pseudo deleted index entries.
5. Reclaim free space.
6. Materialize online schema changes.

The first two bullets in the list presume that list prefetch is much slower than dynamic prefetch. As the performance of list prefetch catches up to the performance of dynamic prefetch, this motivation for REORG is reduced. The combination of solid-state disks and IBM's List Prefetch Optimizer achieves that goal.

The third bullet can be misleading. If you never run REORG and you insert random keys into the index, the average amount of free space per page is 25%. Because the default value of PCTFREE for an index is 10%, running REORG shrinks the size of the index and increases the likelihood of more index splits in the future.

The fourth bullet, that is the cleanup of pseudo deleted index entries, was largely solved by DB2 11. See *DB2 11 Technical Overview*, SG24-8180 for more details. Now DB2 can automatically clean up the pseudo deleted index entries that have the most effect on performance. DB2 autonomically cleans up those index pages that are touched.

The fifth bullet has the most merit. If you do not expect your indexes to change anymore, you could set both PCTFREE and FREEPAGE to 0, and then run REORG. Now your index is organized, it uses less DASD space, and your index scans read fewer pages. The fewer pages that DB2 needs to read, the faster the scan is.

Incidentally, although the free pages that are associated with FREEPAGE can be used to preserve dynamic prefetch when there is an index page split, these free pages do not help to avoid the splits.

Lastly, consider that the third bullet and fifth bullet conflict with each other. So, unless you are sure that your index will not change anymore, it is probably best to let it remain disorganized and save the cost of running REORG.

The sixth bullet lists that REORG INDEX is needed to materialize index pending changes. Pending changes do not cause object unavailability so the REORG utility can be scheduled at more convenient times.

## 8.2 Data reorganization

Now let us review some of the supposed reasons for keeping the data organized:

1. Clean up indirect references.

This first bullet was discussed in 2.6, “Indirect references” on page 20. MAXROWS and with DB2 11 PCTFREE FOR UPDATE can provide a solution for avoiding indirect references. Thus, that reason for running REORG no longer exists.

2. Eliminate synchronous I/Os for existing unclustered rows when doing a range scan, for example.

This second bullet still has merit. Row level sequential detection helps to ensure that DB2 can use dynamic prefetch for clustered pages, but it does not help DB2 to use prefetch for unclustered pages. As the cluster ratio degrades, the performance of a range scan is degraded unless the access path is changed to use a RID list. Given a RID list, DB2 uses list prefetch. Nevertheless, even if list prefetch is as fast as dynamic prefetch, organizing the rows reduces the number of pages that DB2 must read for a range scan. That is assuming, of course, that the range is based on the cluster index key. Unless a secondary index is correlated with the primary cluster index, REORG does not affect the cluster ratio of a secondary index.

3. Re-establish free space in the table space to avoid creating more unclustered rows and indirect references.

The third bullet is very similar to the previous bullet. However, it is only a factor if there will be more inserts in the future.

Here we are talking about range scans based on the cluster key where there is no filtering. An example of this is:

```
SELECT PHONE_NUMBER FROM PHONE_BOOK WHERE LASTNAME = 'Smith';
```

where LASTNAME, FIRSTNAME, STATE is the cluster key.

However, suppose that the query filters the rows that are based on the state as well, such as the following:

```
SELECT PHONE_NUMBER FROM PHONE_BOOK WHERE LASTNAME='SMITH' AND STATE='Alaska';
```

In this case, the qualifying rows are either skip-sequential or random, and even if the table was reorganized, there is a good chance that the query will do sync I/Os anyway because the Alaskan Smiths might be too far apart in the phone book from each other to enable dynamic prefetch I/O to be used. Alternatively, a query that searched for California Smiths might successfully use dynamic prefetch for all of the qualifying rows in an organized table.

4. Materialize online schema changes

The fourth bullet lists that REORG is needed to materialize schema pending changes, mostly for universal table spaces. REORG needs to be scheduled for when the change needs to take effect.



# Related publications

The publications listed in this section are considered particularly suitable for a more detailed description of the topics covered in this paper.

## IBM Redbooks

The following IBM Redbooks publications provide additional information about the topic in this document. Note that some publications referenced in this list might be available in softcopy only:

- ▶ *DB2 11 for z/OS Performance Topics*, SG24-8222
- ▶ *DB2 11 for z/OS Technical Overview*, SG24-8180
- ▶ *Subsystem and Transaction Monitoring and Tuning with DB2 11 for z/OS*, SG24-8182
- ▶ *z/OS Parallel Sysplex Configuration Overview*, SG24-6485
- ▶ *DB2 for z/OS and List Prefetch Optimizer*, REDP-4862
- ▶ *DB2 9 for z/OS: Buffer Pool Monitoring and Tuning*, REDP-4604
- ▶ *How does the MIDAW Facility Improve the Performance of FICON Channels Using DB2 and other workloads?*, REDP-4201

You can search for, view, download, or order these documents and other Redbooks, Redpapers, Web Docs, drafts, and additional materials, at the following website:

[ibm.com/redbooks](http://ibm.com/redbooks)

## Other publications

These publications are also relevant as further information sources:

- ▶ *DB2 11 for z/OS Data Sharing: Planning and Administration*, SC19-4055
- ▶ *DB2 11 for z/OS Installation and Migration Guide*, GC19-4056
- ▶ *DB2 11 for z/OS Command Reference*, SC19-4054
- ▶ *DB2 11 for z/OS Messages*, GC19-4062

## Online resources

This website is also relevant as a further information source:

DB2 11 for z/OS

<http://www.ibm.com/software/data/db2/zos/family/db211>

## Help from IBM

IBM Support and downloads

[ibm.com/support](https://ibm.com/support)

IBM Global Services

[ibm.com/services](https://ibm.com/services)



# IBM DB2 11 for z/OS Buffer Pool Monitoring and Tuning



**Describe the functions of the DB2 buffer pools**

**Check metrics for read and write**

**Set up and monitor the DB2 buffer pools**

IBM DB2 buffer pools are still a key resource for ensuring good performance. This has become increasingly important as the difference between processor speed and disk response time for a random access I/O widens in each new generation of processor. An IBM System z processor can be configured with large amounts of storage, which if used wisely, can help compensate by using storage to avoid synchronous I/O. Several changes in buffer pool management have been provided by DB2 10 and DB2 11.

The purpose of this IBM Redpaper is to cover the following topics:

- ▶ Describe the functions of the DB2 11 buffer pools
- ▶ Introduce a number of matrixes for read and write performance of a buffer pool
- ▶ Provide information about how to set up and monitor the DB2 buffer pools

The paper is intended to be read by DB2 system administrators, but it might be of interest to any IBM z/OS performance specialist. It is assumed that the reader is familiar with DB2 and performance tuning.

In this paper, we also assume that you are familiar with DB2 11 for z/OS performance. See DB2 11 for z/OS Technical Overview, SG24-8180; and DB2 11 for z/OS Performance Topics, SG24-8222, for more information about DB2 11 functions and their performance.

## INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

### BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

**For more information:**  
[ibm.com/redbooks](http://ibm.com/redbooks)