Db2 Prefetching

Understanding, Configuring, Monitoring, Tuning

This document can be found on the web, www.ibm.com/support/techdocs
Search for document number WP102804 under the category of “White Papers”.

Version Date: 14. November 2024

IBM Germany Research & Development

Malte Schünemann

malte_schuenemann@de.ibm.com

Abstract

Unlike other database products, Db2 LUW© does not have all of its data in memory. This is a big cost factor that makes Db2 LUW© very attractive for customers.

Although the demand for memory resources is moderate, Db2 LUW© is still able to run peak load environments with excellent performance. The Db2 prefetching system is a key technology that ensures that requested data is available in memory at the time it is required.

Therefore, the understanding of the prefetching system, its configuration, monitoring, and basic tuning options have become of increasing importance for database administrators who need to ensure and improve the performance of a Db2 LUW© database.

This document explains the Db2 LUW© prefetcher model and shows how you configure and monitor asynchronous I/O, particularly for SQL query processing. It's for administrators of Db2 LUW© environments who have a fair knowledge of Db2 LUW© administrative topics and tasks. The paper is based on the author's experience with Db2 systems in SAP environments, but the information provided is expected to be helpful in all environments running on Db2 LUW©.

Trademarks

The following terms are registered trademarks of International Business Machines Corporation in the United States and/or other countries: AIX, AS/400, DB2, IBM, Micro Channel, MQSeries, Netfinity, NUMA-Q, OS/390, OS/400, Parallel Sysplex, PartnerLink, POWERparallel, RS/6000, S/390, Scalable POWERparallel Systems, Sequent, SP2, System/390, ThinkPad, WebSphere.

The following terms are trademarks of International Business Machines Corporation in the United States and/or other countries: DB2 Universal Database, DEEP BLUE, e-business (logo), ~, GigaProcessor, HACMP/6000, Intelligent Miner, iSeries, Network Station, NUMACenter, POWER2 Architecture, PowerPC 604,pSeries, Sequent (logo), SmoothStart, SP, xSeries, zSeries. A full list of U.S. trademarks owned by IBM may be found at http://www.ibm.com/legal/copytrade.shtml .
NetView, Tivoli and TME are registered trademarks and TME Enterprise is a trademark of Tivoli Systems, Inc. in the United States and/or other countries.
Oracle, MetaLink are registered trademarks of Oracle Corporation in the USA and/or other countries.
Microsoft, Windows, Windows NT and the Windows logo are registered trademarks of Microsoft Corporation in the United States and/or other countries.
UNIX is a registered trademark in the United States and other countries licensed exclusively through The Open Group.
LINUX is a registered trademark of Linus Torvalds.
Intel and Pentium are registered trademarks and MMX, Pentium II Xeon and Pentium III Xeon are trademarks of Intel Corporation in the United States and/or other countries.
Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States and/or other countries.
Other company, product and service names may be trademarks or service marks of others.

Table of Contents

Terminology

Term	Description
Regular data vs. *LONG/LOB data*	This document distinguishes between *regular data* and *LONG/LOB data*. Db2 agents access regular data using the buffer pool, while LONG/LOB data is held in the agent heap. Therefore, regular data include table data pages, index pages, and XDA pages.
Data pages vs. *index* (or *XDA, ...*) *pages*	A Db2 table can consist of the following objects that are stored separately in the database: • data objects • index objects • XDA objects • LONG/LOB objects The document refers to data pages, index pages etc. to distinguish between different kinds of object pages.
LONG/LOB object	The regular data portion of a table needs to fit into a single page of its tablespace. Larger data, known as LONG/LOB data, is stored in separate LONG/LOB objects. Db2 agents access regular data using the buffer pool (page cache), while LONG/LOB data is not cached and is held in temporary buffers stored in the agent heap. Note that the LONG data types are deprecated. In this document, when talking of LOB data, it still means LONG/LOB data unless explicitly mentioned otherwise.
RID	The indices point to the rows of a table using a row ID (= RID). Therefore, the RID is a reference from an index key to a table row.

Conventions and prerequisites

The information provided in this document is valid for Db2 LUW© 11.5 and higher.

If not specified otherwise, references to the IBM Db2 Documentation relate to Db2 LUW© 12.1.

This document uses the following conventions:

- Key terms are written in *italics* when used the first time.
- Parameter names, commands, function names, and other technical identifiers are written in `typewriter` characters.

1 The prefetching system

Physical data access performance is an essential feature of database systems that need to satisfy high load requirements. Since the time to retrieve data from disk usually is much higher than the time to process this data, an essential factor to improve database performance is to reduce the time that data processing needs to wait for physical I/O.

The purpose of the prefetching system is to retrieve data from the storage system to have it available at the time it is required by tasks running in the database.

1.1 The process model

The engine of a Db2 database, the *Db2 engine*, uses a variety of *engine dispatchable units* (*EDU*s) for different tasks. This document focuses on EDUs related to prefetching.

1.1.1 EDUs related to prefetching

EDUs are threads of the Db2 engine process. This paper mainly discusses the following EDUs:

- An EDU requesting data
  In most cases, this is the Db2 agent (`db2agent`), which is an engine thread processing queries requested by the database application. Beside Db2 agents, data is requested by the Db2 buffer manipulator (`db2bm`) in the context of Db2 backups.
- The *prefetchers* (`db2pfchr`)
  Prefetchers read data from the I/O subsystem to have it available at the time the requesting EDU needs it for processing. The prefetcher EDUs provide the data requested by Db2 agents or the Db2 buffer manipulators.

1.1.2 I/O considerations

The prefetching system performs a large part of I/O in a Db2 LUW© database. For many of the tools and utilities performing non-SQL data access, like backup and restore, the prefetching system uses *synchronous I/O*.

In database documentation, *synchronous I/O* is sometimes referred to as *direct I/O*. On the other hand, direct I/O also describes a file system access mode, often abbreviated as DIO. To avoid confusion, this document always uses the term *synchronous I/O*.

All prefetch requests satisfying SQL query processing of non-LONG/LOB objects use *asynchronous vectored I/O*.

The OS side of the I/O operations is the following:

- Vectored I/O uses a field of buffers to read from or write to. A vectored read operation reads data from a single stream and writes it to multiple buffers. Similarly, a vectored write operation gets the data from multiple buffers and writes it to a single output stream.
- Asynchronous I/O submits a request to the asynchronous I/O subsystem and returns immediately. This allows the calling process to continue processing while the I/O is being handled in the background.
- Synchronous I/O submits a request directly to the underlying I/O subsystem and waits for the data requested.

The upcoming chapters of this document deal with query processing (unless explicitly stated otherwise).

1.2 Using the prefetching system

The prefetching system is used by database backup and restore activities and during the execution of SQL statements.

1.2.1 Backup

Db2 backups access data from storage using the prefetching system. All physical read activity is performed as synchronous I/O.

Although the Db2 backup is not a topic discussed in this paper, it is important to know that online backups compete with other Db2 processing for prefetching resources.

1.2.2 Restore

Just like Db2 backups, the Db2 restore also uses the prefetching system. In fact, the Db2 restore is the only scenario where prefetchers do not read from, but write to disk. However, a Db2 restore rarely runs concurrently to normal operation, and prefetching issues usually are not relevant in restore scenarios. Therefore, this paper does not discuss the Db2 restore.

1.2.3 SQL query processing

The execution of SQL queries or statements often requires large portions of database objects that are provided by the prefetching system. Therefore, the analysis of SQL statements also requires that you consider prefetching.

To illustrate this, look at the simple SQL statement of Example 1. The statement accesses all entries of table TABSCHEMA0.TABLE0:

```
select * from tabschema0.table0
```

Example 1: SQL statement

The optimiser access plan, or explain plan (Figure 1), looks as follows:

```
                Rows
               RETURN
               (   1)
                Cost
                 I/O
                 |
               581567
               TBSCAN
               (   2)
               145685
               145392
                 |
               581567
           TABLE: TABSCHEMA0
               TABLE0
                 Q1
```

Figure 1: Table scan in the optimiser access plan

For information on how to get an explain plan, refer to appendix A.

The operator in node 2 (TBSCAN) reads all of table TABSCHEMA0.TABLE0 and therefore is a good candidate for prefetching. In the plan details of the explain output (Figure 2), you can find the corresponding information.

```
        2) TBSCAN: (Table Scan)
                ...
                MAXPAGES: (Maximum pages for prefetch)
                        ALL
                PREFETCH: (Type of Prefetch)
                        SEQUENTIAL
```

Figure 2: Sequential prefetching details

1.3 Prefetching strategies

The Db2 agent tries to anticipate which data portions are required to fulfil its tasks. The prefetching system provides the data that the Db2 agent assumes are required for processing.

One of the benefits of the prefetching approach is that a large I/O request is split up into many smaller I/O requests that are processed in parallel by the prefetching system. For more details, see section 2.3.

1.3.1 Basic concepts

There are different prefetching concepts, depending on how data is located at storage level:

- Sequential prefetching
  The prefetchers read consecutive pages. This is used if the database knows that all or most of the pages being read are really required as seen in the above SQL statement using a table scan.
- Readahead prefetching
  The prefetchers use indices to determine pages to be read.
- List prefetching
  The prefetchers read data pages to read data specified in a list of values. Typically in Db2 LUW© optimiser access plans we see several branches with an index access to retrieve a list of RIDs. After sorting, the query processing uses this list to prefetch the relevant data pages via the prefetching system.

While sequential prefetching for sure is the fastest concept, its efficiency heavily depends on the question whether consecutive pages are needed to satisfy the request being processed. Readahead prefetching and list prefetching require additional work but they limit the amount of data being read to the required minimum. Depending on the nature of the request the Db2 optimiser chooses the most efficient concept.

In Figure 3 you can see an example of list prefetching. The RID scan operator (`RIDSCN`) in the optimiser access plan is responsible for the retrieval of the RID list.

```
            555.066
            FETCH
            (  13)
            20.1155
            3.70901
           /---+----\
       725.587       18263
       RIDSCN   TABLE: TABSCHEMA0
       (  14)       TABLE0
       10.1622        Q5
       1.39117
         |
       725.587
       SORT
       (  15)
       10.1619
       1.39117
         |
       725.587
       IXSCAN
       (  16)
       9.98657
       1.39117
         |
        18263
    INDEX: TABSCHEMA0
       TAB0_INDEX0
         Q5
```

Figure 3: RID scan in an optimiser access plan snippet

The details on node 13 in the access plan snippet indicate the type of prefetching as seen in Figure 4.

```
      13) FETCH : (Fetch)
         ...
              PREFETCH: (Type of Prefetch)
                     LIST
```

Figure 4: List prefetching in the explain details

1.3.2 Smart prefetching strategies

Additional strategies, referred to as *smart data prefetching*, and *smart index prefetching* use sophisticated algorithms to decide which of the basic concepts listed above is to be used for a particular data access. The key feature used in these strategies is known as *sequential detection prefetching*.

When using this feature, the Db2 LUW© engine monitors the I/O of the prefetching system to determine if sequential prefetching is efficient for the current operation. If not, the process switches to readahead prefetching. The availability of sequential detection prefetching depends on the configuration of the database (see section 2.4).

Smart prefetching strategies combine I/O monitoring results and optimiser statistics to decide on the prefetching method to be used.

- Smart data prefetching
  This is either sequential detection or readahead prefetching, depending on the degree of data clustering.
- Smart index prefetching
  This is either sequential detection or readahead prefetching, depending on the density of the index[6] being used.

To identify smart prefetching for an SQL statement, consider a Db2 LUW© optimiser access plan snippet as shown in Figure 5:

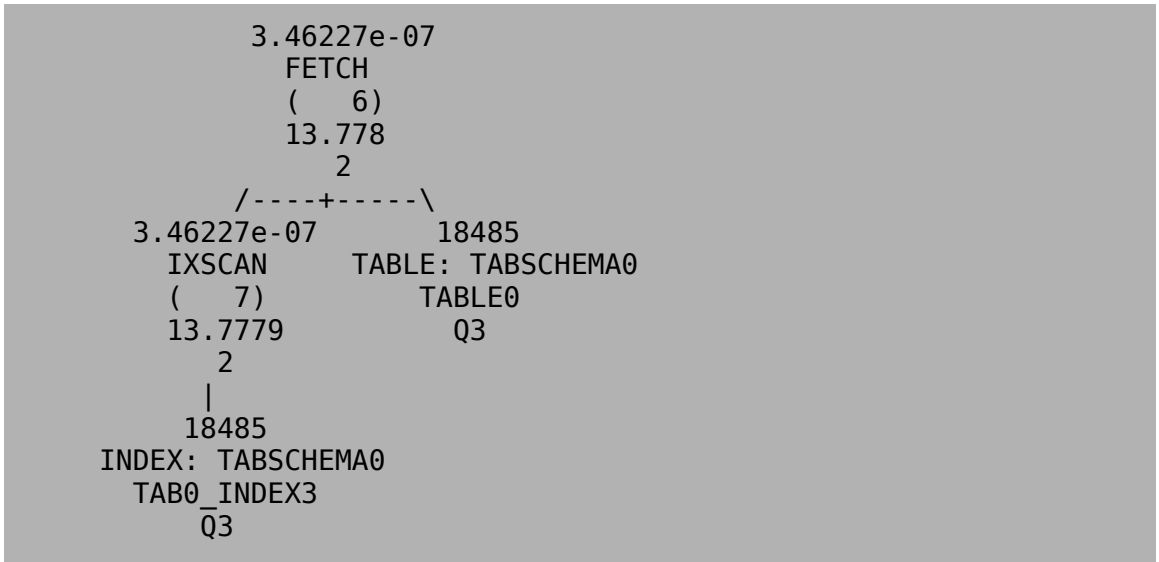


Figure 5: Index and table access in an optimiser access plan snippet

The detail plan section of the access plan (Figure 6) indicates smart data prefetching for the table access (node 6) and smart index prefetching for the index (node 7).

```
            6) FETCH : (Fetch)
                       ...
                  PREFETCH: (Type of Prefetch)
                         SEQUENTIAL,READAHEAD
                       ...
            7) IXSCAN: (Index Scan)
                       ...
                  PREFETCH: (Type of Prefetch)
                         SEQUENTIAL,READAHEAD
```

Figure 6: Smart prefetching in the explain details information

For additional information on smart prefetching strategies, refer to the IBM Db2 Documentation [(1)].

1.4 Requests and prefetching

Db2 agents submit prefetch requests by placing them into a prefetch queue. The prefetchers process these requests as follows:

- Verify if (part of) the requested data is already in the buffer pool. Db2 agents submit requests for all data they need for processing and do not check if they are already available.
- If data pages need to be read, reserve the required space in the buffer pool. The Db2 engine prefers to use consecutive pages in the buffer pool. However, the request can also be read into non-consecutive page areas.
- Submit read requests to the asynchronous I/O subsystem of the OS to read the data from disk into the buffer pool.

The prefetch requests that have been processed are deleted from the queue.

The following are typical scenarios that can occur between the Db2 agents and the prefetching system:

- The data requested by the Db2 agents is available by the time it is needed. This is the desired scenario.

- At the time the prefetchers read the data, the Db2 agent is attempting to access them. The Db2 agent then needs to wait for the prefetchers to provide the data. This wait time is referred to as *prefetch wait time*.
- The Db2 agent is attempting to access the data, but the prefetchers have not yet started processing the request. Then, the Db2 agent itself reads the data pages required for processing. At the time the prefetchers process the request, they will read only the pages of the request that have not yet been retrieved. Generally, data pages that the prefetchers skip as they are already in the buffer pool are referred to as *skipped pages*.
- There is a limit for the number of requests that a prefetch queue can hold. If the maximum size of the prefetch queue is reached, all subsequent requests are rejected until the number of requests drops below its maximum. Again, the Db2 agents will read required data themselves. The requests that have been rejected are referred to as *failed prefetch requests*.

Db2 agents perform synchronous reads to access data pages.

1.5 LOB data prefetching

Db2 LUW© uses the prefetching system to read LOB data that is larger than one extent. LOBs smaller than this size are read by the Db2 agent directly.

In SAP environments, the default extent size is 2 pages. With a page size of 16KB, which is the default for SAP environments, an extent is 32KB. However, the setting of the Db2 registry variable `DB2_WORKLOAD=SAP` implies that LOB data up to 90KB is read by the Db2 agents.

Since the prefetchers retrieve LOB objects using synchronous I/O, data retrieval is synchronous, and prefetchers need to wait for the data requested.

If table data that is defined as LOB data type is small enough to fit into a data page, it can potentially be stored in the data page itself rather than in a separate LOB object. The LOB is said to be *inlined*[7]. Inlined LOB data is handled just like regular data. Note that inlining is possible for LOB data, but not for LONG data.

You know from section 1.4 that in case the prefetching system falls behind, the regular data pages requested are being read by the Db2 agents directly. For LOB objects, this is

not the case. This means that even if the prefetching system is overloaded, the requests for LOB objects need to wait until the prefetchers are able to process the corresponding prefetch request.

1.6 Putting the things together

How do the various parts work together ? Figure 7 shows the path that the prefetch processing takes.

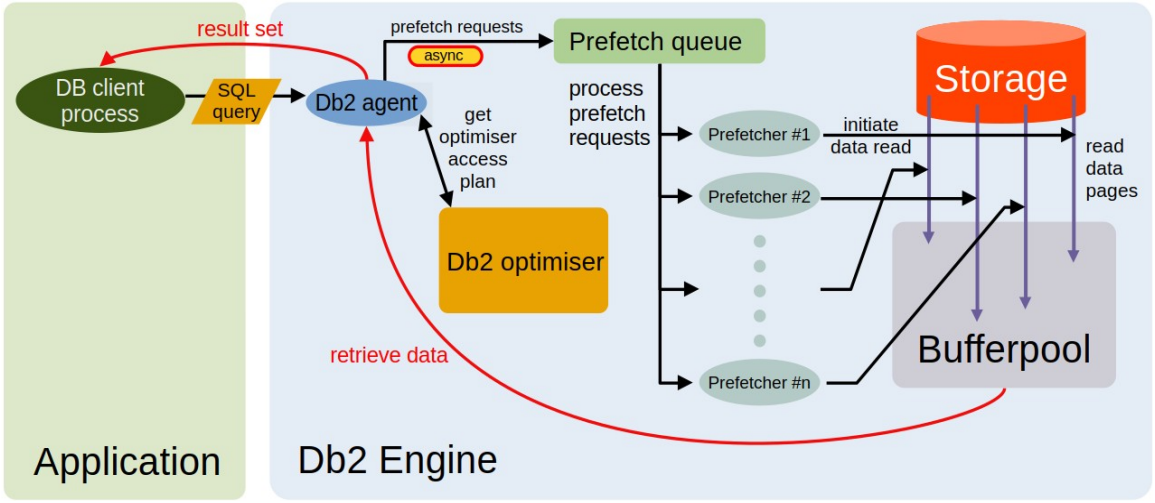


Figure 7: The prefetching process

Assume that the application is requesting information from the database by submitting an SQL query. At the database side, the counterpart of the application process is the Db2 agent. When the agent has received the application request it sends the query to the Db2 optimiser that generates an access plan. This plan includes information which data is required to satisfy the initial request from the application side. With this knowledge the agent starts sending requests for data to the prefetch queue. There is no check at this point if the data is already in the bufferpool, or not. Now the prefetcher EDUs start processing the requests found in the prefetch queue. Basically, when processing a request there are three tasks for the prefetchers:

- check if the data is already in the bufferpool - if so, then there is nothing to do, and the prefetch request can be discarded
- block an area in the Db2 bufferpool for the data to be retrieved
- read the data referred to in the prefetch request into the bufferpool

When the data is available in the bufferpool the agent can now read it for further processing. Finally, the Db2 agent returns the result set to the application process to satisfy the application request.

It remains to add that LOB prefetching looks a bit different, since no bufferpool is involved as shown in Figure 8.

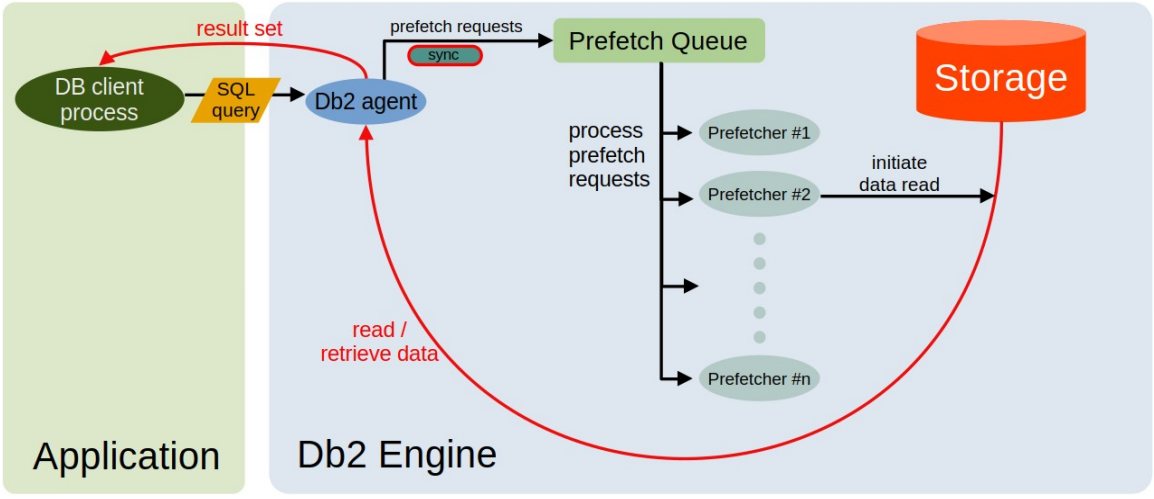


Figure 8: The prefetching process for LOB data

Rather, the prefetching process is synchronous, and the read requests address the application memory of the requesting Db2 agents directly.

2 Configuring

As prefetching is closely related to physical I/O, the performance of the prefetching system heavily depends on I/O characteristics. However, there are still several factors that influence the prefetching system and can be tuned in the Db2 environment.

They include

- the amount of data being read per request
- the number of threads available for prefetching
- the degree of parallelism when processing prefetch requests

2.1 Prefetch request size

The amount of data processed by a single prefetch request is limited by the prefetch size (in number of pages) of the tablespace from which the data is being read. The default

setting is AUTOMATIC. The physical size of a prefetch request is limited by the following configuration parameters of the tablespace holding the data:

```
PAGESIZE      (in bytes)
EXTENTSIZE    (in pages)
PREFETCHSIZE  (in pages)
```

Listing 1: Tablespace parameters relevant for prefetching

After tablespace creation, the parameters PAGESIZE and EXTENTSIZE are fixed. The extent size (in pages) is the minimum allocation unit for table data and indices. Valid pages sizes are 4KB, 8KB, 16KB, and 32KB.

To calculate the effective prefetch size, use Formula 1:

```
Effective prefetch size =
  if [ PREFETCHSIZE == AUTOMATIC ] then
    EXTENTSIZE * <# of storage paths>
  else
    <fixed value as configured>
```

Formula 1: Effective prefetch size in pages

2.2 Number of prefetchers

The number of prefetchers is configured in the database configuration by setting the parameter NUM_IOSERVERS. The value range is between 1 and 255. The setting to AUTOMATIC adjusts to the degree of I/O parallelism (see section 2.3).

In SAP environments, the setting of the Db2 registry variable DB2_WORKLOAD=SAP implies that NUM_IOSERVERS - if set to AUTOMATIC - adjusts to at least 104.

2.3 Prefetching parallelism

One of the factors driving prefetching performance is the degree of parallelism of a single prefetch request.

To illustrate this, let us consider an operation on a table or index within SQL query processing that implies some kind of prefetching. For prefetch requests, there is one

prefetcher thread assigned per storage path of the tablespace where the table or index is located.

The Db2 registry parameter DB2_PARALLEL_IO, if set, multiplies this default I/O parallelism. This means that the prefetch request is broken up into smaller pieces that are assigned to a number of prefetchers depending on the parallelism configured[5].

Note that the above parameter also multiplies the prefetch size if PREFETCHSIZE is set to AUTOMATIC.

- The Db2 registry variable DB2_PARALLEL_IO is particularly useful for environments that have just one storage path. If the parallelism is configured higher than 1, the use of this registry variable is to be considered carefully.
- DB2_PARALLEL_IO can specify the parallelism for individual tablespaces. A setting of DB2_PARALLEL_IO=* resolves to a value of 6 for all tablespaces. Refer to the IBM Db2 Documentation for further details.
- A natural upper limit for parallelism is the physical parallelism. Important factors are e.g. the number of disks in a RAID device (if used) and the number of different storage locations for storage paths.
- A parallelism higher than 32 is not beneficial.

2.4 Sequential detection

To decide about prefetching strategies (see section 1.3.2), Db2 LUW© monitors the I/O behaviour of the prefetchers. This is referred to as sequential detection and turned on by default.

You can turn off this feature using the database configuration parameter SEQDETECT. However, the Db2 Knowledge Center recommends that you leave this parameter turned on.

The parameter is configurable online, changes will become effective immediately.

3 Monitoring

The monitoring table functions (*MTF*s) of Db2 LUW© include fields that allow you to analyse the prefetching behaviour in a Db2 database. These fields are available in several MTFs providing the information at different levels of granularity. See appendix B) for details on how to query MTFs.

In the following, let's see what kind of information is available to monitor the prefetching system. For an overview of the metrics related to prefetching refer to Figure 9 in section 3.5.

3.1 Prefetching amount

To analyse prefetching performance, the amount of prefetching activity is key. But how do you define this quantity? For a good understanding it is essential to distinguish the following kinds of requests:

- Prefetch requests submitted by Db2 agents to the Db2 prefetching system
- Read requests that the Db2 prefetchers submit to the I/O subsystem

Another dimension is the type of information being prefetched, i.e. data, indices, XDA, or columnar. These categories exist for regular data and temporary data.

There are numerous monitoring elements to help evaluate the prefetching quality. It is useful to get a deeper insight into the available metrics and their meaning.

3.1.1 Basic prefetch request monitor elements

To get an idea about prefetching monitoring, first look at the prefetching activity for a single type of database pages, i.e. data pages of row-organized tables.

From previous chapters you know that the Db2 agents request data via prefetch requests. For data pages, the number of requests is counted in the metric POOL_QUEUED_ASYNC_DATA_REQS. The amount of data pages being requested is then kept in POOL_QUEUED_ASYNC_DATA_PAGES. To monitor the physical I/O, there are additional metrics as discussed in section 3.1.2.

As you can imagine, similar elements exist for other kinds of objects. These are indices, XDA objects, column-organized objects, and objects located in temporary tablespaces. For the names of metrics to count the amount of prefetch requests, see Listing 2.

```
POOL_QUEUED_ASYNC_<type>_REQS
POOL_QUEUED_ASYNC_TEMP_<type>_REQS
POOL_QUEUED_ASYNC_<type>_PAGES
POOL_QUEUED_ASYNC_TEMP_<type>_PAGES

where <type> = DATA,INDEX,XDA,COL
```

Listing 2: Metrics for prefetcher requests and requested pages

These metrics keep track of the requests that the requesters (Db2 agents in the case of SQL query processing) submit to the prefetching system. They do not directly reflect the amount of I/O.

From the number of requests submitted and the number of pages requested you can derive the average number of pages requested per prefetch request.

The metrics just discussed only track requests for read operations into the buffer pool. Non-buffer pool prefetching is tracked by POOL_QUEUED_ASYNC_OTHER_REQS. However, this metric refers to requests from non-SQL tools like the backup utility. These are not discussed in more detail in this paper.

3.1.2 Asynchronous read monitor elements

The prefetch requests just take account of the amount of prefetching at database level. The total physical prefetch I/O is accounted for in the metrics VECTORED_IOS and PAGES_FROM_VECTORED_IOS. Apart from prefetching I/O going into the buffer pool, these metrics also include the non-buffer pool I/O from SQL processing of LONG/LOB data as well as I/O from non-SQL access by tools and utilities.

To monitor the physical prefetching I/O coming from buffer pool read requests and the related single read operations at I/O level, use the metrics of Listing 3.

```
POOL_ASYNC_<type>_READ_REQS
POOL_ASYNC_<type>_READS
where <type> = DATA,INDEX,XDA,COL
```

Listing 3: Db2 LUW metrics for asynchronous read requests and reads

Note that the size of an individual read operation is one page.

Similar to prefetch requests and the number of pages per request, you can calculate the number of reads per read request at I/O level.

3.2 Latching

The performance of the prefetching system heavily depends on the throughput of the underlying I/O subsystem. However, the prefetchers not only read *from* some source, they also need to read *to* a destination. In other words, the prefetchers need a memory area where the data then can be accessed by the Db2 agents.

Therefore, the prefetching process implies a request for a memory area in the bufferpool that the data to be prefetched can be read into.

When prefetchers need to wait for free pages in the bufferpool, this is indicated by a latch named `SQLO_LT_SQLB_BP_AREA_INFO_pageArea__clockLatch`.

Impact due to this latch often is indicated by an average prefetch wait time that is noticeably higher than the average physical read time (see sections 3.3.1 and 3.3.2). In contrast, if prefetching is impacted by high I/O time, then the average prefetch wait time increases with the average physical read time.

3.3 Prefetching quality

To see if the prefetching system is working fine, you need to look at the following metrics:

- Average read time per OS read request issued by prefetchers
- Time that agents are waiting for prefetchers (prefetch wait time)
- Number of occurrences that agents were faster than prefetchers (UOW skipped)
- Number of prefetch failures (prefetch queue full conditions)

You can almost always observe a certain number of prefetch waits and skipped prefetch requests. This is due to the design of the prefetching model because query processing starts accessing data immediately after submitting requests. In contrast, the occurrence of prefetch failures always requires attention.

3.3.1 Average asynchronous read time

To evaluate the average read time, start with the buffer pool asynchronous read time (`POOL_ASYNC_READ_TIME`). This is the total amount of time for all prefetching into the buffer pool(s). Note that there is one element for different kinds of data (data, index, column-organized data). Use Formula 2 below.

```
av. async read time (µs) =
    POOL_ASYNC_READ_TIME * 1000 / (
        POOL_ASYNC_DATA_READS +
        POOL_ASYNC_INDEX_READS +
        POOL_ASYNC_XDA_READS +
        POOL_ASYNC_COL_READS + 1
    )
```

Formula 2: Average asynchronous read time (µs)

As the POOL_ASYNC_READ_TIME is reported in milliseconds (ms), the value returned by the formula is given in microseconds (µs) / read I/O. Note the small yet intended inaccuracy in the denominator introduced by adding "+1". If the number of asynchronous reads is not too small, then the formula is sufficiently correct. The benefit is that, in this way, you avoid division by zero.

However, we recommend to use Formula 3 below:

```
av. async read time (µs) =
    POOL_ASYNC_READ_TIME * 1000 /
        COALESCE( NULLIF(
            ( POOL_ASYNC_DATA_READS +
              POOL_ASYNC_INDEX_READS +
              POOL_ASYNC_XDA_READS +
              POOL_ASYNC_COL_READS ), 0), POWER( BIGINT(-2),63 )
        )
```

Formula 3: Average asynchronous read time (µs)

Here, the construction using functions COALESCE() and NULLIF() avoids the division by zero. As this formula is more accurate, it should be preferred.

If the sum of POOL_ASYNC_...READS becomes 0, the denominator becomes the maximum possible BIGINT value, which will make the fraction to become 0.

The average read time calculated by Formula 2 and Formula 3 provides insight into the performance of the underlying I/O subsystem. The higher the value, the more the prefetching is delayed. In turn, this leads to a performance impact on the Db2 agents. Find more on this topic in section 4.2.2.

3.3.2 Prefetch waits

To assess the quality of prefetching, first check the amount of time that agents are waiting for prefetchers to finish their job. The Db2 engine reports the total prefetch wait

time (in milliseconds) in the metric PREFETCH_WAIT_TIME. The number of wait occurrences is counted in PREFETCH_WAITS. You can easily calculate the average prefetch wait time as shown in Formula 4.

```
av. prefetch wait time (µs) =
    PREFETCH_WAIT_TIME * 1000 /
        COALESCE( NULLIF( PREFETCH_WAITS, 0 ),
                  POWER( BIGINT(-2),63 ) )
```

Formula 4: Average prefetch wait time (µs)

As discussed in section 3.2, an average prefetch wait time significantly higher that the average physical read time is an indication of a latching issue in the bufferpool.

3.3.3 Prefetching LONG/LOB data

As described in section 1.4, a prefetch wait situation implies that a prefetcher is currently assigned and is processing the request. Therefore, the average read time discussed earlier has a relation to the average prefetch wait time.

Prefetch waits are also an issue for LOB requests. However, the prefetching system is only used for LOB objects larger than a certain limit (see section 1.5). For related metrics, refer to Listing 4.

```
LOB_PREFETCH_REQS
LOB_PREFETCH_WAIT_TIME
```

Listing 4: Prefetch wait metrics for LOB access

LONG/LOB objects are read using synchronous I/O. Therefore, each LOB prefetch request automatically indicates a wait situation.

LOB prefetch metrics are available as of Db2 LUW© 11.1. Note that you need to track LOB prefetching independently from prefetching into the buffer pool.

The metric PREFETCH_WAIT_TIME relates to prefetching into the buffer pool and does not contain LOB_PREFETCH_WAIT_TIME. Similarly, LONG/LOB prefetch requests are not accounted for in metrics for POOL_QUEUED_ASYNC_ . . . _REQS.

3.3.4 Data retrieved but never used

Bad prefetching behaviour may be caused by pages that are unnecessarily read. The counter UNREAD_PREFETCH_PAGES collects the number of pages being read by

prefetchers but never being used by Db2 agents. The table functions `mon_get_database()` and `mon_get_tablespace()` provide this information at database and tablespace level.

In a well-tuned database environment, this number is small as compared to asynchronous physical reads (see section 3.1.2). A high number is an indication of additional load onto the I/O subsystem and the buffer pool.

A high `PREFETCHSIZE` of tablespaces may be the reason for a high value of this counter.

3.3.5 Skipped prefetching

Generally, skipped prefetching indicates that at the time when the prefetcher is processing a prefetch request, the requested pages are already available in the buffer pool. A reason may be that the pages have been retrieved already from the activity of other Db2 transactions or UOWs. At the time the request is to be processed, it is then skipped by the prefetchers.

If a Db2 agent that has requested database pages via prefetching is attempting to access one of the pages, it checks if the prefetchers have already started processing the request. If not, the Db2 agent retrieves the page itself. This means the prefetching system is too slow to provide the data in due time, which is an indication of a performance problem.

Therefore, you need to distinguish skipped prefetching due to activity in other UOWs from skipped prefetching within the same UOW.

For the latter case, use the metrics `SKIPPED_PREFETCH_UOW_DATA_P_READS`, `SKIPPED_PREFETCH_UOW_TEMP_DATA_P_READS`, and similar for `INDEX`, `XDA`,and `COL`.

The total amount of skipped prefetch requests is shown by the metrics for `SKIPPED_PREFETCH_DATA_P_READS` etc., i.e. you just omit the `_UOW` from the former metric names. With the total of these metrics you can derive Formula 5.

```
Prefetch request skip ratio (%) =
    (
        SKIPPED_PREFETCH_UOW_DATA_P_READS +
        SKIPPED_PREFETCH_UOW_INDEX_P_READS +
        SKIPPED_PREFETCH_UOW_XDA_P_READS +
        SKIPPED_PREFETCH_UOW_COL_P_READS +
        SKIPPED_PREFETCH_UOW_TEMP_DATA_P_READS +
        SKIPPED_PREFETCH_UOW_TEMP_INDEX_P_READS +
        SKIPPED_PREFETCH_UOW_TEMP_XDA_P_READS +
        SKIPPED_PREFETCH_UOW_TEMP_COL_P_READS
    ) * 100 / (
        COALESCE( NULLIF(
            ( SKIPPED_PREFETCH_DATA_P_READS +
              SKIPPED_PREFETCH_INDEX_P_READS +
              SKIPPED_PREFETCH_XDA_P_READS +
              SKIPPED_PREFETCH_COL_P_READS +
              SKIPPED_PREFETCH_TEMP_DATA_P_READS +
              SKIPPED_PREFETCH_TEMP_INDEX_P_READS +
              SKIPPED_PREFETCH_TEMP_XDA_P_READS +
              SKIPPED_PREFETCH_TEMP_COL_P_READS ), 0),
            POWER( BIGINT(-2),63 ) )
    )
```

Formula 5: Prefetch request skip ratio (in %)

Too high a value is an indication for bad prefetch performance. The percentage is expected to be a lower single digit number (i.e. well below 5%).

Note that there is a natural relation between page requests to the prefetching system, pages skipped and physical I/O:

```
( POOL_ASYNC_QUEUED_<type>_PAGES +
  POOL_ASYNC_QUEUED_TEMP_<type>_PAGES ) =
             ( SKIPPED_PREFETCH_<type>_P_READS +
               SKIPPED_PREFETCH_TEMP_<type>_P_READS +
               POOL_ASYNC_<type>_READS )
```

where **<type> = DATA,INDEX,XDA,COL**

Formula 6: Number of pages requested, skipped, read from I/O

This means that the amount of queued pages equals the number of skipped pages plus the number of pages read from the underlying I/O subsystem. The equation is valid per data type.

3.3.6 Failed requests

If the prefetch queue is full and cannot accept further requests, any attempt to submit a prefetch request returns an error referred to as *prefetch queue failure*. Frequent occurrence indicates severe performance problems. For options to address this kind of issue, refer to section 4.2.4.

A Db2 LUW© database running with good performance has close to zero prefetch queue failures.

The occurrence of request submission failures indicate that the prefetching system is not working properly. Such a situation has far-reaching consequences. In fact, as described in section 1.5, you can expect LOB access to be particularly delayed. Depending on the application and the data being processed, a high number of failed request submission potentially leads to virtual hang scenarios in the application.

Find below the relevant metrics (Listing 5).

```
POOL_FAILED_ASYNC_<type>_REQS
POOL_FAILED_ASYNC_TEMP_<type>_REQS
POOL_FAILED_ASYNC_OTHER_REQS
```

where **<type> = DATA,INDEX,XDA,COL**

Listing 5: Db2 LUW metrics for prefetch queue failures

Reasons for prefetch queue failures can be a contention for prefetcher resources and high I/O request times. For more information, see chapter 4.

3.4 Prefetcher usage

You have seen in section 2.2 how to configure the number of prefetchers in a Db2 LUW© environment. Therefore, the monitoring of a prefetching system requires checking if the configured number of prefetchers is sufficient or needs adjustment.

The MTF `env_get_db2_edu_system_resources()` provides the amount of user and system CPU for all threads of the Db2 engine. To confine them to prefetcher EDUs, use the following SQL:

```
select current timestamp as collection_timestamp, t.*
  from table(env_get_db2_edu_system_resources()) as t
  where t.edu_name like 'db2pfchr%'
```

Example 2: SQL for Db2 prefetcher information via table function

Refer to appendix B) for details on how to use MTFs. The next figure shows you the most relevant columns of this SQL statement:

```
EDU_ID EDU_NAME         CPU_USER_TIME_MS CPU_SYSTEM_TIME_MS
------ ---------------- ---------------- ------------------
   270 db2pfchr (...) 0       3904511291         3111677820
   269 db2pfchr (...) 0       3928320432         3160420541
   268 db2pfchr (...) 0       3940641625         3146765300
   267 db2pfchr (...) 0       3907127188         3131386818
   ...
   170 db2pfchr (...) 0      34675990726        29640813557
   169 db2pfchr (...) 0      34424363293        29698400291
   168 db2pfchr (...) 0      34593283502        29905526105
   167 db2pfchr (...) 0      34629504748        29114142044
   ...
```

Example 3: Db2 prefetcher info from monitoring table function

You can see that the prefetchers with the lowest EDU ID consume the highest amount of CPU. Note that a prefetch request is always processed by the first available prefetcher, i.e. of the idle prefetchers, the one with the lowest EDU ID. The data from Example 3 suggest a high usage of prefetchers.

As an alternative to the MTF, use the command `db2pd -edus` to yield roughly the same information.

```
$ db2pd -edus
...
EDU ID ... EDU Name         ...     USR (s)         SYS (s)
=======...==================...============= ==============
...
270    ... db2pfchr (...) 0 ...  3904.511291  3111.677820
269    ... db2pfchr (...) 0 ...  3928.320432  3160.420541
268    ... db2pfchr (...) 0 ...  3940.641625  3146.765300
267    ... db2pfchr (...) 0 ...  3907.127188  3131.386818
...
170    ... db2pfchr (...) 0 ... 34675.990726 29640.813557
169    ... db2pfchr (...) 0 ... 34424.363293 29698.400291
168    ... db2pfchr (...) 0 ... 34593.283502 29905.526105
167    ... db2pfchr (...) 0 ... 34629.504748 29114.142044
...
```

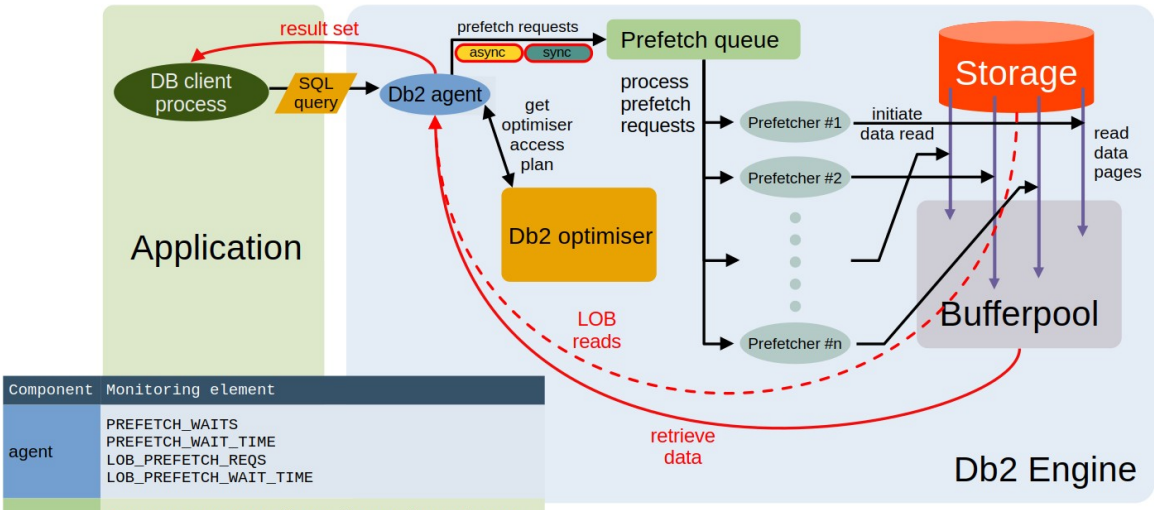
Example 4: Db2 prefetcher info in `db2pd -edus`

You can use the additional switch `interval=<number of seconds>` for db2pd to get the change rate of the CPU consumption within the time interval indicated.

3.5 Summary of the prefetch metrics

The amount of metrics that is available for monitoring the prefetcher activity indeed is overwhelming. It therefore is good to have an overview over the available metrics, and to know which part in the prefetch processing they belong to.

Recall the prefetching process as displayed in Figure 7. In Figure 9 below you can see which parts the various metrics are monitoring.



Component	Monitoring element
agent	PREFETCH_WAITS PREFETCH_WAIT_TIME LOB_PREFETCH_REQS LOB_PREFETCH_WAIT_TIME
prefetch queue	POOL_QUEUED_ASYNC_{[TEMP_]<type>|OTHER}_REQS POOL_QUEUED_ASYNC_[TEMP_]<type>_PAGES POOL_FAILED_ASYNC_{[TEMP_]<type>|OTHER}_REQS
prefetcher	SKIPPED_PREFETCH_[TEMP_]<type>_P_READS SKIPPED_PREFETCH_UOW_[TEMP_]<type>_P_READS
storage	POOL_ASYNC_<type>_READ_REQS POOL_ASYNC_<type>_READS POOL_ASYNC_READ_TIME VECTORED_IOS PAGES_FROM_VECTORED_IOS
bufferpool	UNREAD_PREFETCH_PAGES
latches	SQLO_LT_SQLB_BP_AREA_INFO_pageArea__clockLatch
<type> = DATA, INDEX, XDA, COL	

Figure 9: The prefetching process with monitoring metrics

4 Tuning

If monitoring leads to the conclusion that performance is not optimal, the next step is tuning. As there is no complete or comprehensive general tuning method, this chapter provides guidelines and examples based on experience. You may find your own best practices for Db2 tuning.

4.1 Guidelines

Proper tuning is a science on its own. In this chapter, you can find some tuning guidelines. However, keep in mind that every performance problem needs to be considered individually.

4.1.1 Interpretation of monitoring results

It is often tempting to quickly decide on the reason for bad performance. However, in most cases, there is more than one possible cause for bad results of a particular metric.

As an example, high values for prefetch wait time may be caused by a delay at I/O level. On the other hand, the prefetchers need to reserve pages in the buffer pool before they can start reading. Consequently, a lack of memory in the buffer pool may also increase the processing time of a prefetcher request.

Therefore, you should first get the complete picture of a problem before interpreting the results.

4.1.2 Tuning steps

Tuning is often an iterative approach. Therefore, you need to be able to backtrack the results of any configuration changes. Follow these golden tuning rules:

1. When making changes to the environment, take one step at a time.
2. Monitor the system after each change.
3. If the results are good, take the next step. If a change returned bad results, switch back to the previous setting.
4. Continue with the next change.

4.2 Prefetching system tuning areas

From a database user or application point of view, the prefetching system is hidden in the background. Nevertheless, it is of high relevance for the overall database performance and is interconnected with many other areas of the Db2 LUW© engine. You can tune the prefetching system mainly in the following ways:

- At the end of the read operation, that is, I/O system and buffer pool
- By configuring its resources

4.2.1 Buffer pool contention

Since prefetchers retrieve data directly into the buffer pool, the prefetching performance is sensitive to contention issues at buffer pool level.

Depending on the root cause of buffer pool contention, you have the following options to address the problem:

- Reduce the amount of data being read by SQL statements.
  For example, with an additional index on a table being queried frequently, you may be able to turn a table scan reading millions of rows into an index access

reading a small number of pages. You may also review the application and modify its design to reduce the amount of database data that is accessed.

- Reduce the configured amount of data to be read by prefetchers.
  As described in section 3.3.4, you can check if the amount of data that is read per prefetch request is too high. To reduce this amount, you can decrease the PREFETCHSIZE of the tablespace that the data is read from. Note that this change will affect all tables and indices located in that tablespace. If the problem is related to few tables or indices, it may be useful to create one or more new tablespaces with proper values of EXTENTSIZE and PREFETCHSIZE and move these tables to the new tablespaces. To move a table and its indices, use the Db2 LUW© procedure ADMIN_MOVE_TABLE() [8].
- Improve the cleanup of dirty buffer pool pages, known as *page cleaning*. Refer to the IBM Db2 Documentationfor more details on this topic [2][3].
- Increase the amount of memory assigned to the buffer pool.

As explained in section 3.2, bufferpool contention and delayed prefetching frequently is accompanied by latch SQLO_LT_SQLB_BP_AREA_INFO_pageArea__clockLatch. Possibly, you also see a low bufferpool hitratio, but this is not necessarily the case.

In summary, you can overcome this kind of problems by reducing the amount of data that is read into the buffer pool, or by increasing the buffer pool size.

4.2.2 I/O subsystem

If you find prefetching is often delayed, one of the first things to check is the average read I/O. A good value for a high load environment is below 1ms (or <1000 when using Formula 3). For environments with reduced load, you might find values of 2ms still acceptable.

Another option is to increase the amount of data prefetched for certain tablespaces because a single read access to retrieve a large amount of data is potentially faster than several consecutive read requests of small size. To do so, either

- increase the parameter PREFETCHSIZE of the relevant tablespace(s) or
- create one of more new tablespaces with larger EXTENTSIZE and PREFETCHSIZE and move relevant tables there.

You probably need larger I/O requests for only a limited number of tables. Again, a useful approach often is to move these tables to dedicated tablespaces that have proper values

of EXTENTSIZE and PREFETCHSIZE. As mentioned in the previous section, it is advisable to keep the balance between larger prefetch sizes and the potential of buffer pool contention.

Investigating I/O performance is a separate topic and outside the scope of this paper. I/O performance depends on many specific details as to hardware in use and type of environment. Particularly the use of network file systems adds another potential for delay, either on the file server, the client, in the network, or at software level.

4.2.3 Prefetcher contention

Increased values for the average prefetch wait time (Formula 4) and/or prefetch request skip ratio (Formula 5) indicate that prefetching is delayed and even may be behind the Db2 agents. A possible reason is that the number of prefetchers is too low (see section 3.4) and needs to be increased.

Similar to buffer pool contention (section 4.2.1), the amount of data being read by prefetchers may also lead to prefetcher contention. Therefore, the tuning steps described there also help to avoid possible prefetcher contention issues.

Keep in mind that the prefetching system is used not only by SQL query processing. Particularly if you regularly perform online backups or REORG operations in parallel to the normal database load you might quickly run into a prefetcher bottleneck.

To address prefetcher contention issues, you have the following options:

- Increase the number of prefetchers. Remember that the change becomes effective after the next database restart.
- Schedule non-SQL query activities like online backups or REORGs to run at times of reduced load. Particularly in the case of backups, you may also use a different strategy, e.g. a flash copy approach, to keep the related load off the database server.

Note that you may also run into contention issues if the I/O is delayed.

4.2.4 Prefetch queue full issues

As pointed out in section 3.3.6, prefetch queue full issues need to be addressed. The measures mentioned in the previous tuning sections also resolve failed requests (prefetch queue full) issues.

The key is to increase the processing speed of single requests while keeping their number limited so that the queue never becomes full.

4.3 Additional tuning

There are many other items that also influence prefetching. However, this paper restricts the discussion to things that directly impact prefetching such as the following:

- Access to the buffer pool may also suffer from delay due to latching. In this case, the investigation needs to turn towards the reason for latching. In most cases, the prefetching system is the victim rather than the source of the problem.
- Prefetching may suffer from a CPU bottleneck. However, the prefetching system depends on many other components like I/O. Therefore, a CPU bottleneck usually is far more impacting for other components like SQL processing.

In many cases, the tuning of other areas in the Db2 LUW© engine also influences the prefetching system. For example, adding CPU power may speed up backup processing and provide relief to the prefetching system as well.

Keep in mind that it is important to get the full picture before starting to tune the environment.

Appendix

A) Db2 LUW© explain utilities

With the EXPLAIN utilities, you see how the Db2 LUW© engine processes an SQL statement. The tools provide the *optimiser access plan* and the plan details.

To get detailed information on this topic, refer to the IBM Db2 Documentation [(9)Error: Reference source not found].

B) Use of Db2 LUW© Monitoring Table Functions (MTFs)

You can query all metrics discussed in this paper using the MTFs available in Db2 LUW©. For a list of MTFs and their usage, refer to the IBM Db2 Documentation [(4)].

A good starting point is to query metrics at database level. The following SQL statement retrieves prefetch wait metrics at database level for the current database partition.

```
select current timestamp as collection_timestamp,
       t.prefetch_waits, t.prefetch_wait_time
  from table(mon_get_database(-1)) as t
```

Example 5: Prefetch wait metrics at database level

Unlike snapshot functions, the MTFs do not include the time when the data is collected. Therefore, it is helpful to always add the `current timestamp` register to the query.

To display the same metrics per database connection, run the statement of Example 6.

```
select current timestamp as collection_timestamp,
       t.prefetch_waits, t.prefetch_wait_time
  from table(mon_get_connection(null,-1,0)) as t
```

Example 6: Prefetch wait metrics at connection level

Note that the table function `mon_get_connection()` retrieves information for current connections only. The information for connections that have been closed is not available any more. For example, if you sum up the values of `PREFETCH_WAITS` over all connections, you most probably get a value much smaller than what you see in `mon_get_database()`.

You can query the metrics of Example 6 also at transaction (UOW) level, at statement level, or per buffer pool of the database.

To retrieve the full set of metrics available at database level, run the SQL statement of Example 7.

```
select current timestamp as collection_timestamp, t.*
  from table(mon_get_database(-1)) as t
```

Example 7: Retrieve all metrics at database level

Note that for counters like the prefetch wait time or number of requests, you receive the quantity collected since start time of the monitored object. So the function `mon_get_database()` retrieves the total prefetch wait time and the number of prefetch waits since the last database start. Function `mon_get_connection()` retrieves this information per connection since connection start time.

When monitoring a database or a part of it, you want to see the difference between short time intervals rather than absolute values of counters between short time intervals. In examples 5 and 6, the essential information is how the prefetch wait time and the number of prefetch waits behave in collection iterations that are e.g. a few minutes apart. A monitoring approach that is able to display the changes of counters across short time intervals is discussed in a separate paper [11].

References

(1) IBM Db2 Documentation - Prefetching data into the buffer pool
(2) IBM Db2 Documentation - Improving update performance
(3) IBM Db2 Documentation - Proactive page cleaning
(4) IBM Db2 Documentation - Monitor procedures and functions
(5) IBM Db2 Documentation - Parallel I/O management
(6) IBM Db2 Documentation - Index structure
(7) IBM Db2 Documentation - Storing LOBs inline in table rows
(8) IBM Db2 Documentation - ADMIN_MOVE_TABLE procedure - Move tables online
(9) IBM Db2 Documentation - Guidelines for capturing explain information
(10) IBM Db2 Documentation - EXPLAIN_FROM_SECTION procedure
(11) Using Table Functions in Db2 LUW - A Db2 Monitoring Approach