Data Manipulation

All regular table data and indexes for a database are manipulated within the database’s buffer pool area. A database must have at least one buffer pool associated with it but can use multiple buffer pools. The buffer pools are the work area for the database, and all searching for and manipulation of the data and indexes must take place within the buffer pools. In order for DB2 to scan a table or an index, the pages of the table or index must be in the database’s buffer pool (or buffer pools). If the required page is already in the buffer pool, DB2 can start to work on the page immediately. If the page is not in the buffer pool, DB2 must read the page from disk and position it in the buffer pool(s) before it can be manipulated. Because most data manipulation takes place in the database buffer pools, configuring the buffer pools is the single most important tuning area for DB2 Universal Database (UDB) databases.

The database buffer pools have a profound effect on the overall performance of the database and, as such, there have been a number of enhancements to the manner in which buffer pools can be created and managed in DB2 UDB Version 8. This chapter will focus on the creation, management, and monitoring of buffer pools in DB2 UDB Version 8 to provide optimal performance for the database.

**The Buffer Pools**

The database buffer pool area is a piece of memory used to cache a table’s index and data pages as they are being read from disk to be scanned or modified. The buffer pool area helps to improve database system performance by allowing data to be accessed from memory instead of from disk. Because memory access is much faster than disk access, the less often that DB2 needs to read from or write to a disk, the better the system will perform.
When a database is created, there will be one default buffer pool created for the database. This buffer pool is named IBMDEFAULTBP; it has a page size of 4 KB; and it will be sized depending on the operating system. For Windows, the default buffer pool will be 250 pages or 1 MB, whereas for UNIX, the default buffer pool will be 1,000 pages or 4 MB. The default buffer pool cannot be dropped; however, the size of the default buffer pool can be changed, using the ALTER BUFFERPOOL statement.

A database’s buffer pool(s) are allocated in memory when the database is activated or when the first application connects to the database. When all applications disconnect from the database and it is deactivated, the buffer pool memory is deallocated and freed back to the operating system. With DB2 UDB Version 8, buffer pools can now be created, dropped, and resized while the database is active.

**Buffer Pool Usage**

When a DB2 agent acting on behalf of an application retrieves a row in a table for the first time, DB2 UDB must first read the page containing that row from disk and place it in the buffer pool. The next time any application requests data, DB2 UDB will look for it in the buffer pool. If the requested data is in the buffer pool, it can be retrieved without disk access, resulting in faster performance.

Once a page is read into the buffer pool, it will remain in the buffer pool until the database is stopped or until the space used by the page in the buffer pool is needed to read another page into the buffer pool. When reading a page into the buffer pool, it will be placed into an empty buffer pool page if one exists. If the buffer pool is full (i.e., there are no empty pages available), DB2 will choose a victim page, based on the following:

- How recently the page was last accessed
- The probability that the page will be read again by the last agent that accessed it
- The type of data on the page (i.e., index or data)
- Whether the page was changed in the buffer pool but has not yet been written to disk

To guarantee the integrity of the data and transactions, changed pages must first be written to disk before they can be replaced in the buffer pool. Because the transaction must wait for the page to be written to disk, this can slow down the performance of the system. To overcome this condition and improve performance, DB2 UDB uses buffer pool cleaners periodically to scan the buffer pool for dirty pages (i.e., pages that have changed since being read into the buffer pool) and asynchronously write them to disk. To optimize the access to the data on these pages, they will not be removed from the buffer pool after they have been written to disk; however, they will no longer be dirty, so they can be replaced in the buffer pool without the need to write them to disk.

Reading pages from disk is an expensive, time-consuming operation. Reading several consecutive pages into the buffer pool using a single I/O operation can greatly reduce this overhead and allow applications to process their work faster. In addition, multiple parallel I/O operations to
read several ranges of pages into the buffer pool can help reduce I/O wait time. In DB2 UDB this is accomplished using I/O prefellers to read pages from disk into the buffer pool in the expectation that they will be required by an application. Prefetching of index and data pages into the buffer pool can help improve performance by reducing the I/O wait time and further taking advantage of parallel I/O operations.

**NOTE** Prefetching and buffer pool cleaning will be discussed in more detail later in this chapter.

Creating Buffer Pools

The CREATE BUFFERPOOL statement will create a buffer pool for the database that the user is connected to. In previous versions of DB2 UDB, the buffer pool would not be allocated until the database was stopped and restarted (i.e., all current applications were disconnected from the database and another connection was established). With DB2 UDB Version 8, the buffer pool can be allocated immediately, if there is enough memory available to handle the request.

The CREATE BUFFERPOOL statement has options to specify the following:

- **bufferpool-name:** Specifies the name of the buffer pool. The name cannot be used for any other buffer pools and cannot begin with the characters SYS or IBM.
- **IMMEDIATE:** Specifies that the buffer pool will be created immediately if there is enough memory available on the system. If there is not enough reserved space in the database shared memory to allocate the new buffer pool, a warning is returned, and buffer pool creation will be DEFERRED, as described below.
  - This is the default.
- **DEFERRED:** Specifies that the buffer pool will be created the next time the database is stopped and restarted.
- **ALL DBPARTITIONNUMS:** Specifies that the buffer pool will be created on all partitions in the database. This is the default if no database partition group is specified.
- **DATABASE PARTITION GROUP:** Specifies the database partition group(s) on which the buffer pool will be created. The buffer pool will be created on all database partitions that are part of the specified database partition groups.
- **SIZE:** Specifies the size of the buffer pool and is defined in number of pages. In a partitioned database, this will be the default size for all database partitions where the buffer pool exists.
  - The EXCEPT ON DBPARTITIONNUMS clause described below allows the buffer pool to have different sizes on the different database partitions.
- **EXCEPT ON DBPARTITIONNUMS:** Specifies the database partition or partitions where the buffer pool will be sized differently than specified by the SIZES clause. This can be specified as a single partition, a list of partitions, or a range of partitions. To specify a list of partitions, separate the database partitions in the list by commas. To
specify a range of partitions, use the TO clause between the partition numbers. When specifying a range of partitions, the second database partition number must be higher than the first database partition number, and the buffer pool must exist on all of the database partitions. After the partitions or partitions have been specified, their size must also be specified.

- **NUMBLOCKPAGES**: Specifies the number of pages to be created in the block-based area of the buffer pool. The actual value of NUMBLOCKPAGES may differ from what was specified because the size must be a multiple of the BLOCKSIZE. The block-based area of the buffer pool cannot be more than 98% of the total buffer pool size. Specifying a value of 0 will disable block I/O for the buffer pool.

- **BLOCKSIZE**: Specifies the number of pages within a given block in the block-based area of the buffer pool. The block size must be between two and 256 pages, and the default value is 32 pages.

- **PAGESIZE**: Specifies the page size used for the buffer pool. The default page size is 4 KB, or 4096 bytes. The page size can be specified in both bytes and KB.

- **EXTENDED STORAGE/NOT EXTENDED STORAGE**: Specifies whether buffer pool victim pages will be copied to a secondary cache called extended storage. Extended storage is more efficient than retrieving data from disk but less efficient than retrieving data from the buffer pool, so it is not applicable to 64-bit environments.

**NOTE** Once defined for a buffer pool, the page size and buffer pool name cannot be altered.

Enabling block-based I/O by setting NUMBLOCKPAGES to a value greater than 0 can help performance for applications that do sequential prefetching. If the applications do not perform sequential prefetching of pages into the buffer pool, this area of the buffer pool will be wasted.

**NOTE** NUMBLOCKPAGES does not need to be set to allow prefetching to occur.

Block-based buffer pools have the following limitations:

- A buffer pool cannot be made block-based and also use extended storage.
- A buffer pool cannot be made block-based and also support Address Windowing Extensions (AWE) on Windows.
- If a buffer pool is enabled for AWE support and block-based I/O is enabled, the block-based I/O support will be disabled.
The Buffer Pools

The following statement:

```
CREATE BUFFERPOOL BP1
SIZE 25000
```

will create a buffer pool named BP1 with a size of 100 MB (25,000 4-KB pages = 100 MB). Because the page size is not specified, the buffer pool will use the default page size of 4 KB. Because the IMMEDIATE option is the default, the buffer pool will be allocated immediately and available for use, as long as there is enough memory available to fulfill the request.

The following statement:

```
CREATE BUFFERPOOL BP2
SIZE 25000
PAGESIZE 8 K
```

will create a buffer pool named BP2 with a size of 200 MB (25,000 8-KB pages = 200 MB). The buffer pool will use an 8-KB page size. Because the IMMEDIATE option is the default, the buffer pool will be allocated immediately and available for use, as long as there is enough memory available to fulfill the request.

The following statement:

```
CREATE BUFFERPOOL BP3
DEFERRED
SIZE 1000000
```

will create a buffer pool named BP3 with a size of 4 GB (1,000,000 4-KB pages = 4 GB). Because the page size is not specified, the buffer pool will use the default page size of 4 KB. Because the DEFERRED option is specified, the buffer pool will not be allocated until the database is stopped and restarted.

The following statement:

```
CREATE BUFFERPOOL BP4
IMMEDIATE
SIZE 100000
NUMBLOCKPAGES 32000
BLOCKSIZE 256
```

will create a buffer pool named BP4 with a size of 400 MB (100,000 4-KB pages = 400 MB). Because the page size is not specified, the buffer pool will use the default page size of 4 KB. The buffer pool will have 128 MB set aside for block-based I/O to help optimize sequential prefetch activity. Because the IMMEDIATE option is specified, the buffer pool will be allocated immediately and available for use, as long as there is enough memory available to fulfill the request. Because the block size is set to 256 pages, it is recommended that the extent size for the table spaces assigned to this buffer pool also be set to 256 pages.
The proportions of the page area and block area are not maintained. The block-based area of the buffer pool will be altered only if:

- NUMBLOCKPAGES is specified in the ALTER BUFFERPOOL statement
- The buffer pool SIZE is altered to something that would cause NUMBLOCKPAGES to be greater than 98% of SIZE.

For example, consider the buffer pool BP1, defined as:

```sql
create bufferpool BP1 size 9600 numblockpages 4800 blocksize 32
```

This gives a page-based area of 4,800 pages and a block-based area of 4,800 pages.

The statement:

```sql
alter bufferpool BP1 size 6400
```

would change the buffer pool to have a page-based area of 1,600 pages and a block-based area of 4,800 pages.

In the following statement, the specified SIZE is less than the current NUMBLOCKPAGES value of 4,800:

```sql
alter bufferpool BP1 size 3200
```

This would cause the following error to be returned.

```
DB21034E  The command was processed as an SQL statement because it was not a valid Command Line Processor command. During SQL processing it returned: SQL20150N  The number of block pages for a buffer pool is too large for the size of the buffer pool.  SQLSTATE=54052
```

In the following statement, the specified SIZE is less than the current NUMBLOCKPAGES value of 4,800. However, because the BUFFPAGE parameter is used, the buffer pool will be reduced in size, but the block-based area will have to be reduced accordingly. This is done to
allow for a certain number of pages to exist in the page-based area for workloads that do not consist of sequential prefetching. Use this statement:

```sql
update db cfg for TEST using BUFFPAGE 3200
alter bufferpool BP1 size -1
```

for a partitioned database with 12 partitions and partition group DPG1 defined to include database partitions 0, 1, 2, 3, 4, 5, 6, and 7. To create a buffer pool named BP5 with a size of 25,000 pages and a page size of 32 KB that will be allocated only on the partitions where DPG1 is defined, use the following statement:

```sql
CREATE BUFFERPOOL BP5
DATABASEPARTITIONGROUP DPG1
SIZE 25000
PAGESIZE 32K
```

For the same database and partition groups as described above, to create a buffer pool named BP6 with a page size of 32 KB and a size of 25,000 pages on partitions 0, 1, 2, 3, 4, and 5 and a size of 35,000 pages on partitions 6 and 7, use either of the following statements:

```sql
CREATE BUFFERPOOL BP6
DATABASEPARTITIONGROUP DPG1
SIZE 25000
PAGESIZE 32K
EXCEPT ON DBPARTITIONNUMS 6 TO 7 SIZE 35000
CREATE BUFFERPOOL BP6
DATABASEPARTITIONGROUP DPG1
SIZE 25000
PAGESIZE 32K
EXCEPT ON DBPARTITIONNUM 6 SIZE 35000
EXCEPT ON DBPARTITIONNUM 7 SIZE 35000
```

How Many Buffer Pools Should a Database Have?

There are two main schools of thought regarding the number of buffer pools that should be configured for a database:

- Use one big buffer pool and let DB2’s aging algorithm take care of which pages are in the buffer pool.
- Use multiple buffer pools and assign them to specific table spaces to ensure that highly active pages are kept in the buffer pool.

If DB2 is installed in 32-bit mode, there are limitations on the maximum amount of shared memory that can be addressed by a process. Because the buffer pool is allocated in shared memory, there are limits on the maximum size of the buffer pool(s) that can be created for a database,
regardless of the amount of physical memory on the system. If DB2 is installed in 64-bit mode, the shared memory limit is much higher, and most systems will run out of real memory before hitting the operating system limit.

If they are properly sized and assigned to the right table spaces, multiple buffer pools will outperform a single buffer pool. However, multiple buffer pools will require constant monitoring and tuning in order to keep them performing optimally.

Other than its size, a single buffer pool needs no tuning. DB2 has a highly optimized algorithm for aging pages in the buffer pool that uses several techniques designed to optimize the buffer pool hit ratio by:

- Favoring important pages, such as index pages
- Placing pages that are unlikely to be accessed again on “Hate stacks,” which are used to identify victim pages quickly

**When to Consider Multiple Buffer Pools**

Multiple buffer pools should be considered under the following conditions:

- There are multiple applications accessing the database, and the user wants to favor a particular application.
- Certain tables do not need large buffer pools, for example:
  - Tables that are always appended to, such as journal or history tables.
  - Huge tables (i.e., bigger than the buffer pool) that are only ever fully scanned.
    - These tables will likely always require disk I/O, regardless of buffer pool size.
    - Assign these tables to a dedicated buffer pool; otherwise, when they are scanned, they will likely cause other important pages to be flushed from the buffer pool.
    - Be careful to make these tables’ dedicated buffer pool large enough for effective prefetching.
- If tables are usually scanned concurrently, separate them into their own dedicated buffer pools.
  - UDB will try to keep these pages in memory; but in aggregate, this could swamp the buffer pool and interfere with the access to other objects.
- To isolate high-priority or high-usage tables or indexes from other database activity.
- If people occasionally run large reports on OLTP systems and the reports do large joins, sorts, etc. that cause large temporary tables to be created.
  - These temporary tables can sometimes overwhelm the buffer pool, so it can be beneficial to create a buffer pool for the temporary table space.
  - Size this buffer pool according to the typical overflow sorts, if there are any, under normal operations.
  - This buffer pool typically does not need to be large because it is normally accessed sequentially.
This way, the reports have less impact on the overall performance of the online system.

Buffer Pool Overhead

For every page of buffer pool space and/or extended storage, DB2 creates a 100-byte descriptor in the database heap to store a descriptor of the page. Before creating large buffer pools, it may be necessary to increase the size of the database heap (dbheap) configuration parameter to hold the descriptors for the buffer pool pages.

For a 1-GB buffer pool with a page size of 4 KB, there must be 262,144 pointers allocated in the database heap. This will use up approximately 25 MB of database heap.

For a 32-GB buffer pool with a page size of 4 KB, there must be 8,388,608 pointers allocated in the database heap. This will use up approximately 800 MB of database heap.

32- and 64-bit Considerations

DB2 UDB Version 8 can be installed as a 32-bit or a 64-bit application. When DB2 is installed on the 64-bit version of UNIX, setting the DATABASE_MEMORY configuration parameter to AUTOMATIC allows DB2 to grow its memory usage as needed, when the buffer pools grow, or when additional memory is needed for database control blocks.

When DB2 is installed in 32-bit mode, the amount of shared memory available and the memory management options are more limited than in 64-bit mode. To guarantee that memory is available to create a new buffer pool or enlarge an existing buffer pool with the IMMEDIATE option, memory must be reserved when the database is started. If the database configuration parameter DATABASE_MEMORY is set to a numeric value, when the database is activated, DB2 will allocate the minimum of the following:

- The total shared memory used by the buffer pools, lock list, database heap, shared sort heap threshold, package cache, and utility heap.
- The amount of memory specified by DATABASE_MEMORY.

Note: The db2level command now displays whether DB2 is installed in 32- or 64-bit mode, in addition to the version and fixpack level.

Shared Memory/Buffer Pool Limits

When DB2 is installed in 32-bit mode, there are operating system limitations to the maximum size of a shared memory that a process can allocate. Because each database will have all of its shared memory allocated in the same segment, there is a maximum amount of shared memory that can be allocated per database.
It is important to note that the limitation is on the total amount of shared memory that can be allocated/addressed, and the buffer pool is just one of the items that gets allocated in the database’s shared memory. The following items are allocated in shared memory:

- Buffer pools (buffpage or BP size)
- Lock list (locklist)
- Package cache (pckcachesz)
- Shared sorts
  - If intra-partition parallelism is enabled, reserve the shared sort heap threshold (SHEAPTHRES_SHR).
- Database heap (dbheap)
  - Log buffer (logbufsz)
  - Catalog cache (catalogcache_sz)
- Utility heap (util_heap_sz)

In 32-bit AIX, there are a maximum of 16, 256 MB memory segments addressable by a process. Of these, only seven can be used by DB2 for shared memory. Of these seven segments (or 1.75 GB), one is used for memory mapped I/O, and one may be used by the fast communications manager (FCM) for inter- or intra-partition communication (if there are multiple database partitions or intra-partition parallelism is enabled). To maximize the amount of shared memory that can be used for the buffer pool, memory-mapped I/O can be disabled, and the FCM can be forced to use the network instead of shared memory.

To disable memory mapped I/O, set the following DB2 registry variables to no.

- DB2_MMAP_READ
- DB2_MMAP_WRITE

Using memory-mapped I/O is most beneficial for systems that contain table spaces with few file containers, i.e., SMS or DMS files. Memory-mapped I/O helps to avoid i-node latching by spreading the I/Os across more files (and more underlying i-nodes). If memory-mapped I/O is turned off, make sure that any SMS table space (or DMS table space using file containers) has enough containers (i.e., 3+ per file system) to avoid i-node contention. Otherwise, any performance gain from the extra memory will be lost due to contention on disk.

To force the FCM to use the network instead of shared memory, set the following DB2 registry variable to no.

- DB2_FORCE_FCM_BP

The maximum addressable amount of shared memory for the 32-bit version of DB2 varies, depending on the operating system. Table 2.1 shows shared memory limits.
Maximizing Buffer Pool Size on Windows

When working with Windows 2000, the total addressable memory can be up to 64 GB; therefore, the maximum buffer pool sizes that can be created on Windows equals 64 GB minus the memory used by the operating system and other DB2 memory allocations, assuming that the server is dedicated to DB2. The support for large memory addressing on Windows is provided by the Microsoft Address Windowing Extensions (AWE). Through AWE, Windows 2000 Advanced Server provides support for up to 8 GB of memory addressing, whereas Windows 2000 Data Center Server provides support for up to 64 GB of memory.

To take advantage of memory addresses above 2 GB, both DB2 and Windows 2000 must be configured correctly to support AWE. To be able to address up to 3 GB of memory, the /3GB Windows 2000 boot option must be set. To enable access to more than 4 GB of memory via the AWE memory interface, the /PAE Windows 2000 boot option must be set. To verify that you have the correct boot option selected, under Control Panel, select System, then select “Startup and Recovery.” From the drop-down list, you can see the available boot options. If the boot option (/3GB or /PAE) you want is selected, you are ready to proceed to the next task in setting up AWE support. If the option you want is not available for selection, you must add the option to the boot.ini file on the system drive. The boot.ini file contains a list of actions to be done when the operating system is started. Add /3GB or /PAE, or both (separated by blanks), at the end of the list of existing parameters. Once you have saved this changed file, you can verify and select the correct boot option, as mentioned above.
Windows 2000 also has to be modified to associate the right to “lock pages in memory” with the userid that was used to install DB2. To set the “lock pages in memory” correctly, once you have logged on to Windows 2000 as the user who installed DB2, under the Start menu on Windows 2000, select the “Administrative Tools” folder, then the “Local Security Policy” program. Under the local policies, you can select the user rights assignment for “lock pages in memory.”

DB2 also requires the setting of the DB2_AWE registry variable to be able to take advantage of the larger memory addressing. This registry variable must be set to the buffer pool ID of the buffer pool that will be larger than 3 GB and have a need for AWE support, as well as the number of physical pages and the address window pages to be allocated for the buffer pool.

The buffer pool ID is found in the BUFFERPOOLID column in the catalog view SYSCAT.BUFFERPOOLS. The number of physical pages to allocate should be less than the total number of available pages of memory, and the actual number chosen will depend on the working environment. For example, in an environment where only DB2 UDB and database applications are used on the server, normally select a value between one-half of the available memory up to 1 GB less than the available memory. In an environment where other nondatabase applications are also running on the server, these values will need to be reduced to leave memory available for the other applications. The number used in the DB2_AWE registry variable is the number of physical pages to be used in support of AWE and for use by DB2. The upper limit on the address window pages is 1.5 GB, or 2.5 GB when the /3GB Windows 2000 boot option is in effect.

Hidden Buffer Pools

When a database is activated or started (i.e., during the first connection to the database), DB2 automatically creates four hidden buffer pools for the database, in addition to the IBMDEFAULTBP and any user-created buffer pools. These buffer pools are hidden and do not have entries in the system catalog tables. In addition, these buffer pools cannot be used directly by assigning table spaces to them and cannot be altered.

There will be one hidden buffer pool per page size (i.e., 4 KB, 8 KB, 16 KB, and 32 KB) to ensure that there is a buffer pool available under all circumstances. DB2 UDB will use these buffer pools under the following conditions:

- When the CREATE BUFFERPOOL statement is executed and the IMMEDIATE option is specified, but there is not enough memory available to allocate the buffer pool.
  - If this occurs, a message is written to the administration notification log.
  - Any table spaces that are using the buffer pool will be remapped to the hidden buffer pool with the same page size.
- When the IBMDEFAULTBP and/or any of the user-created buffer pools cannot be allocated when the database is activated or started.
  - If this occurs, a message is written to the administration notification log.
Any table space that is using a buffer pool that was not allocated will be remapped to the hidden buffer pool with the same page size.

DB2 will be fully functional because of the hidden buffer pools, but performance will be drastically reduced.

- When a table space is created and its page size does not correspond to the page size of any of the user created buffer pools.
- During a roll forward operation if a buffer pool is created and the DEFERRED option is specified.
  - Any user-created table spaces that are assigned to this buffer pool will be remapped to the hidden buffer pool with the same page size for the duration of the roll forward operation.

By default, the hidden buffer pools will be created with a size of 16 pages. This can be changed, using the DB2_OVERRIDE_BPF registry variable. To change the size of the hidden buffer pools to use 64 pages each, set the registry variable as follows:

```
DB2SET DB2_OVERRIDE_BPF=64
```

**Altering Buffer Pools**

The ALTER BUFFERPOOL statement will change the defined attributes for the specified buffer pool in the database that the user is connected to. In previous versions of DB2 UDB, the buffer pool size could not be changed until the database was stopped and restarted (i.e., all current applications were disconnected from the database and another connection was established).

With DB2 UDB Version 8, the buffer pool size can be changed immediately, provided that there is enough memory available to handle the request. However, changes to any other buffer pool configuration parameters will be deferred until the database is stopped and restarted.

The ALTER BUFFERPOOL statement has options to specify the following:

- **bufferpool-name**: Specifies the name of the buffer pool. The name cannot be used for any other buffer pools and cannot begin with the characters SYS or IBM.
- **IMMEDIATE**: Specifies that the buffer pool will be created immediately if there is enough memory available on the system. If there is not enough reserved space in the database shared memory to allocate the new buffer pool, a warning is returned, and buffer pool creation will be DEFERRED. This is the default.
- **DEFERRED**: Specifies that the buffer pool will be created the next time that the database is stopped and restarted
- **DBPARTITIONNUM**: Specifies the database partition group on which the ALTER BUFFERPOOL statement will take effect.
- **SIZE**: Specifies the size of the buffer pool and is defined in number of pages. In a partitioned database, this will be the default size for all database partitions where the
buffer pool exists. The EXCEPT ON DBPARTITIONNUMS clause allows the buffer pool to have different sizes on the different database partitions.

- **EXTENDED STORAGE/NOT EXTENDED STORAGE:** Specifies whether buffer pool victim pages will be copied to a secondary cache called extended storage. Extended storage is more efficient than retrieving data from disk but less efficient than retrieving data from the buffer pool so it is not applicable to 64-bit environments.

- **ADD DATABASE PARTITION GROUP:** Specifies the database partition group(s) to which the buffer pool will be added. The buffer pool will be created on all database partitions that are part of the specified database partition group(s).

- **NUMBLOCKPAGES:** Specifies the number of pages to be created in the block-based portion of the buffer pool. The actual value of NUMBLOCKPAGES may differ from what was specified because the size must be a multiple of the BLOCKSIZE. The block-based portion of the buffer pool cannot be more than 98% of the total buffer pool size. Specifying a value of 0 will disable block I/O for the table space.

- **BLOCKSIZE:** Specifies the number of pages used for the block-based I/O. The block size must be between 2 and 256 pages, and the default value is 32 pages.

A buffer pool cannot be altered to use both block-based I/O and extended storage at the same time.

**NOTE** Before altering the number of pages in a buffer pool, it is important to understand and assess the impact on applications accessing the database. A change in buffer pool size may result in a different access plan.

For the buffer pool created with the following statement:

```sql
CREATE BUFFERPOOL BP1
SIZE 25000
```

In order to change the size of the buffer pool to 200 MB use the following statement:

```sql
ALTER BUFFERPOOL BP1
SIZE 50000
```

For the buffer pool created with the following statement:

```sql
CREATE BUFFERPOOL BP2
IMMEDIATE
SIZE 100000
NUMBLOCKPAGES 32000
BLOCKSIZE 256
```
In order to change the size of the block-based area of the buffer pool to 200 MB use the following statement:

```sql
ALTER BUFFERPOOL BP2
NUMBLOCKPAGES 50000
```

**NOTE** Changes to the size of the block-based area of the buffer pool will not take effect immediately. They will take effect when the database is stopped and restarted.

For the buffer pool created with the following statement:

```sql
CREATE BUFFERPOOL BP3
DATABASEPARTITIONGROUP DPG1
SIZE 25000
PAGESIZE 32K
```

In order to allocate this buffer pool on the database partitions in the partition group DPG2, use the following statement:

```sql
ALTER BUFFERPOOL BP3
ADD DATABASEPARTITIONGROUP DPG2
```

For the buffer pool created with the following statement:

```sql
CREATE BUFFERPOOL BP4
IMMEDIATE
SIZE 100000
NUMBLOCKPAGES 32000
BLOCKSIZE 256
```

the following statement:

```sql
ALTER BUFFERPOOL BP4
EXTENDED STORAGE
```

would produce an error because block-based I/O and extended storage cannot both be enabled at the same time.

**Block-Based Buffer Pools Can Improve Sequential Prefetching**

The DB2 UDB buffer pools are page-based; therefore, when contiguous pages on disk are prefetched into the buffer pool, they likely will be placed into noncontiguous pages within the buffer pool. Sequential prefetching can be enhanced if contiguous pages can be read from disk into contiguous pages within the buffer pool.
This can be accomplished by enabling block-based I/O for the buffer pool, using the NUMBLOCKPAGES and BLOCKSIZE parameters for the buffer pool. A block-based buffer pool will contain a page-based area, as well as a block-based area, with sizes based on the NUMBLOCKPAGES parameter. The page-based area will be used for any I/O operation that is not performed using sequential prefetching. The block-based area of the buffer pool will be made up of a number of “blocks,” where each block will contain a set number of contiguous pages, defined by the BLOCKSIZE.

To make the block-based I/O as efficient as possible, it is important to try to match the extent size for the table spaces, using the buffer pool with the block size for the buffer pool. Because the prefetch operation will attempt to read an extent from the table space into a block in the buffer pool, having these use the same size allows for the most optimal use of the memory. DB2 will still use the block-based area of the buffer pool for prefetching if the extent size is larger than the block size of the buffer pool. If the extent size is smaller than the block size, DB2 may still use the block-based area for prefetching, but if there is too much difference between the extent and block sizes, DB2 may choose to use the page-based area of the buffer pool, depending on the size difference.

If some of the pages that have been requested in the prefetch request have already been read into the page-based area of the buffer pool, using page-based I/O, the prefetcher may or may not read the data into the block-based area, depending on how much of the block would be wasted. The I/O server allows some wasted pages in each buffer pool block, but if too much of a block would be wasted, the I/O server will prefetch the data into the page-based area of the buffer pool. Space is considered as wasted under the following conditions:

- The page would be empty because the extent size is smaller than the block size
- The page already exists in the page-based area of the buffer pool

**System Catalog Views Relating to Buffer Pools**

There are two system catalog views that can be used to retrieve information about the buffer pools defined within a database:

- `SYSCAT.BUFFERPOOLS`
- `SYSCAT.BUFFERPOOLDBPARTITIONS`

**SYSCAT.BUFFERPOOLS View**

This view contains a row for each buffer pool defined in each database partition group in the database. It contains the following columns:

- **BPNAME**: Name of the buffer pool. The name is defined when you create the buffer pool with the `CREATE BUFFERPOOL` statement.
- **BUFFERPOOLID**: The unique identifier assigned to the buffer pool when it is created
The Buffer Pools

- **DBPGNAME**: The name of the database partition group where the buffer pool is defined. Will be blank if the buffer pool is on all partitions.
- **NPAGES**: The size of the buffer pool, in number of pages.
- **PAGESIZE**: The page size for the buffer pool.
- **ESTORE**: Indicates whether the buffer pool is using extended storage as a secondary cache.
  - Y = Yes
  - N = No
- **NUMBLOCKPAGES**: The number of pages in the buffer pool set aside for block-based I/O, in number of pages.
  - Zero if block-based I/O is not enabled
- **BLOCKSIZE**: The block size of a given block in the block-based area of the buffer pool.
  - Zero if block-based I/O is not enabled

The following is an example of the contents of this view:

```sql
select * from SYSCAT.BUFFERPOOLS
```

<table>
<thead>
<tr>
<th>BPNAME</th>
<th>BUFFERPOOLID</th>
<th>DBPGNAME</th>
<th>NPAGES</th>
<th>PAGESIZE</th>
<th>ESTORE</th>
<th>NUMBLOCKPAGES</th>
<th>BLOCKSIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBMDEFAULTBP</td>
<td>1</td>
<td>-</td>
<td>250</td>
<td>4096</td>
<td>N</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BP3</td>
<td>2</td>
<td>PG2</td>
<td>25000</td>
<td>32768</td>
<td>N</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BP4</td>
<td>3</td>
<td>-</td>
<td>10000</td>
<td>4096</td>
<td>N</td>
<td>3200</td>
<td>256</td>
</tr>
</tbody>
</table>

3 record(s) selected.

To calculate the sizes of the buffer pools in MB, use the following statement:

```sql
select BPNAME, NPAGES*PAGESIZE/1024/1024 AS SIZE_in_MB from SYSCAT.BUFFERPOOLS
```

The following is an example of the output of this statement:

<table>
<thead>
<tr>
<th>BPNAME</th>
<th>SIZE_in_MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBMDEFAULTBP</td>
<td>9</td>
</tr>
<tr>
<td>BP3</td>
<td>781</td>
</tr>
<tr>
<td>BP4</td>
<td>39</td>
</tr>
</tbody>
</table>

3 record(s) selected.
SYSCAT.BUFFERPOOLDBPARTITIONS View

This view contains a row for each database partition in the database partition group, where the size of the buffer pool is different from the default size specified in the column NPAGES in the SYSCAT.BUFFERPOOLS view. It contains the following columns:

- **BUFFERPOOLID**: The unique identifier assigned to the buffer pool when it is created
- **DBPARTITIONNUM**: The database partition number where the buffer pool has a different size.
- **NPAGES**: The size of the buffer pool, in number of pages on the specified database partition.

The following is an example of the contents of this view:

```
select * from SYSCAT.BUFFERPOOLDBPARTITIONS
```

<table>
<thead>
<tr>
<th>BUFFERPOOLID</th>
<th>DBPARTITIONNUM</th>
<th>NPAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>30000</td>
</tr>
</tbody>
</table>

1 record(s) selected.

What Is Prefetching?

When an agent acting on behalf of an application needs to access table or index pages, it will first look for the pages in the database buffer pool area. If the page cannot be found in the buffer pool area, it will be read from disk into the buffer pool. I/O is very expensive, and in this case, the agent cannot do anything but wait for the read request to finish before it can access the page. These page reads are typically done one page at a time, and if the application also needs to access subsequent pages within the table or index, this is not an efficient method for reading the pages into the buffer pool.

In many situations, DB2 UDB can anticipate the pages that will be requested by an application and read them into the buffer pool before the agent actually attempts to access them. This is referred to as *prefetching*. Prefetching can improve the database performance because the pages will be found in the buffer pool when the agent accesses them, reducing or eliminating the time the application must wait for the page to be read from disk into the buffer pool. This is more relevant to DSS-type workloads that scan large indexes and tables than for OLTP-type workloads that involve less scanning and more random insert/update/delete activity.

Prefetching can be enabled by the DB2 optimizer when it is building the access plan for a statement and determines that it will be scanning a large portion of a table or index. It can also be enabled or triggered when DB2 is executing an access plan and detects that it has read a number of pages in sequence and will likely continue to do so. This is known as *sequential detection* and can be enabled or disabled, using the database configuration parameter SEQDETECT.
Restart recovery will automatically use the prefachers to improve restart time; however, prefetching can be disabled during restart recovery by setting the DB2 registry variable DB2_AVOID_PREFETCH to ON.

How Does Prefetching Work?
Prefetching will be requested by a DB2 agent if the access plan has specified that prefetching should be done or if sequential detection is enabled and a number of pages are read in sequence. The DB2 agent will make prefetch requests that will be examined by the prefetch manager and assigned to the DB2 I/O servers (also known as prefetchers).

The prefetch requests are sent to a queue where they are examined by the prefetch manager. The prefetch manager examines the requests to determine whether any of the prefetch requests from different applications can be combined and handled more efficiently in one prefetch operation. If no prefetch requests can be combined, the queue acts in a first in-first out (FIFO) manner.

To ensure that the prefetchers do not work too aggressively, reading data into the buffer pool and overwriting pages before they can be accessed, the amount of prefetching is controlled by the size of the buffer pool, as well as the prefetch size and the access plan generated by the optimizer. For smaller buffer pools, prefetching may be scaled back to ensure that prefetched pages do not flood the buffer pool and kick out pages that are currently being used or have been prefetched into the buffer pool and have not been accessed.

Within DB2 UDB, there are two main types of data and index prefetch requests handled by the prefetchers: range and list prefetching.

Range prefetching is used to prefetch a sequential range of pages from a database object and is used during table scans and when triggered by sequential detection. If a block-based area is set aside in the buffer pool, DB2 will perform a “big block” read to read a contiguous series of pages into a single private memory buffer, then will copy the block of pages to the block-based area of the buffer pool with one memory copy operation.

Internally, the mechanism used is big block read or “vectored” read (on platforms that support an efficient vectored read API). A big block read involves reading a contiguous block of pages into a single, private memory buffer. Each individual page is then copied into its own buffer pool slot (and all of these slots are likely to be scattered throughout the buffer pool). A vectored read API will read a contiguous block of pages from disk into different memory buffers (a different memory buffer for each page of the block). In this case, those memory buffers can actually be the individual buffer pool slots, and the overhead of the copy from one memory buffer to another is avoided (hence, an efficient vectored read API is preferred over big block read). The block-based area of the buffer pool overcomes this by setting aside contiguous blocks of the buffer pool equal in size to the extent size to make prefetching more efficient, especially on operating systems that do not support vectored reads.
**List prefetching** is used to prefetch a list of pages into the buffer pool area when an index scan produces a list of record IDs (RIDs) that must be prefetched into the buffer pool. List prefetching is normally decided on and specified in the access plan when the SQL statement is optimized. A list prefetch request may get converted to a range prefetch request if the pages are found to be sequential.

There is one **prefetch queue** that is monitored by the prefetch manager and shared between all of the configured prefetchers. The prefetch queue can handle up to 100 prefetch requests.

When a prefetch request is made, the request will be broken up into a number of smaller I/O requests. By default, the number of I/O requests will be determined by the number of containers that are in the table space. However, if the DB2_PARALLEL_IO registry variable is set, DB2 will start up a number of prefetchers equal to the prefetch size divided by the extent size. This allows multiple prefetchers to be working on the requests in parallel.

**Choosing the optimal prefetch size**  The prefetch size for a table space can be changed using the ALTER TABLESPACE statement so it can be adjusted if the prefetchers are either over-prefetching or under-prefetching. Over-prefetching results in wasted space in the buffer pool and/or overwriting pages in the buffer pool that are likely to be reused by other applications. Under-prefetching results in high prefetch wait times and usually causes the DB2 agents to perform the I/O themselves.

As a general rule of thumb, set the prefetch size as follows:

For a table space with multiple containers:

\[
\text{prefetch size} = \text{extent size} \times \text{number of containers}
\]

For a table space with a single container on a Redundant Array of Independent Disks (RAID) or striped disk with the DB2_PARALLEL_IO variable set:

\[
\text{prefetch size} = \text{extent size} \times \text{number of disks in the stripe set}
\]

If the containers are on separate physical disks, this will allow parallel I/O that will enable the prefetchers to read all extents simultaneously. The prefetching can be made more aggressive by increasing the prefetch size to be an integer multiple of the above formula. However, for this to be effective, there must be sufficient disk bandwidth to support it. In addition, the number of prefetchers may need to be increased (but only if the registry variable DB2_PARALLEL_IO is enabled, because the number of requests will be (prefetch size/extent size). With the registry variable DB2_PARALLEL_IO disabled (which is the default), the number of requests will be based on the number of containers in the table space, so increasing the prefetch size will not result in more prefetch requests and will increase the amount of I/O requested in each request.

Consider the following example:

A table space has two containers with a prefetch size set to twice the extent size. With the registry variable DB2_PARALLEL_IO disabled, there will be two prefetch requests since there are
two containers in the table space, and each prefetch request will be for one extent. With the reg-
istry variable DB2_PARALLEL_IO enabled, there will be two prefetch requests because the
prefetch size divided by the extent size equals 2, and each prefetch request will be for one extent.

If the prefetch size is increased to be four times the extent size, with the registry variable
DB2_PARALLEL_IO disabled, there will still only be two prefetch requests because there are
two containers in the table space. However, each prefetch request will now be for two extents.
With the registry variable DB2_PARALLEL_IO enabled, there will be four prefetch requests
because the prefetch size divided by the extent size equals 4, and each prefetch request will still
be for one extent.

Increasing the prefetch size when the registry variable DB2_PARALLEL_IO is disabled will nor-
mally result in the same number of prefetch requests but with more I/O assigned to each request.

Choosing the number of prefec ters  If prefetch size is set as suggested above (i.e.,
prefetch size = extent size  \times  number of containers) for all of the table spaces in the database, and
all of the table spaces are being scanned at the same time, then the number of prefec ters should
be equal to the number of disks belonging to the database. However, if one or more of the table
spaces has been set up using more aggressive prefetching (i.e., the prefetch size is a multiple
of this value) and/or some of the table spaces are not being scanned at the same time as the others,
then the calculation becomes more complicated. To determine the number of prefec ters
required in this case:

- Determine the number of table spaces that will potentially be scanned at the same time.
- For each of these table spaces, determine the number of prefec ters required to service
  a scan of it (based on the formulas above).
- Sum these values to determine the total number of prefec ters required.

Prefec ting example  When the optimizer determines that a table scan can benefit from
prefec ting, DB2 will not prefetch the entire table in at once because this could potentially flood
the buffer pool and result in important pages being removed; it does the prefec ting in stages.

In the example shown in Figure 2.1, the prefetch size is equal to four times the extent size. Each
block represents an extent.

![Figure 2.1  DB2 prefetching.](image-url)
New prefetch requests are made each time a trigger-point page is read by the agent. Trigger-point pages are located at every multiple of the prefetch size within the table.

When the table scan starts, the first trigger point will be hit when the first page in the table is read. The first trigger point will request that DB2 prefetch the first two prefetch-sized blocks in the table. The first block will actually be one page less than the regular prefetch size and will start on the second page in the table to avoid prefetcher/agent contention on that first page. All subsequent trigger points will make a prefetch request to bring in the next prefetch-sized block of pages. If the system is configured and tuned properly, the agent should never have to wait on I/O because the prefetcher will have already read the page into the buffer pool.

If there is more than one container in the table space and/or DB2_PARALLEL_IO is enabled, it will be much more efficient to have multiple prefetchers performing the I/O, rather than one prefetcher. In this case, the prefetch request is broken up into smaller requests, and each of the requests is placed onto the prefetch queue. The prefetchers can then service the requests in parallel, potentially performing much faster than a single prefetcher doing all of the work.

In the example above, if DB2_PARALLEL_IO is enabled, the larger request would be broken into four (prefetch size / extent size = 4) smaller prefetch requests. If DB2_PARALLEL_IO is disabled (which is the default), the number of prefetch requests would be based on the number of containers in the table space.

In the example, the last trigger point will not result in prefetching because there are not enough pages remaining to fill an entire prefetch request.

**Other roles of the prefetchers**  
In addition to the true prefetching work, prefetchers have some other key functions. These functions include:

1. **DROPPING TEMPORARY OBJECTS:** During query processing, one or more system temporary tables may be created to hold intermediate or final result sets. When a system temporary table is no longer required, it is dropped. This involves purging (removing) its pages from the buffer pool, and this can take a relatively long time. So, rather than having the agent drop the table and wait for it to complete, it instead makes a prefetcher request that will handle the drop table work. This reduces the agent's response time because it does not have to do the work itself.

2. **ADDING DMS CONTAINERS:** When more than one container is added to a DMS table space in a single ALTER TABLESPACE statement, the prefetchers assist the agent in allocating the disk space for those containers. The agent handles the allocation for one container, and for each of the remaining containers it creates a prefetch request to add the container. This allows the container allocation to be done in parallel, speeding up the process.
3. RESIZING DMS CONTAINERS: This request type is similar to adding DMS containers, but it handles allocation of new space when the RESIZE or EXTEND options of the ALTER TABLESPACE statement are used.

4. TRAVERSING INDEX LEAF PAGES: This request type is mainly used by the REORG utility when scanning an index object. Given the root page of an index, the leaf level is traversed using the prefetchers.

What Is Page Cleaning?

When an agent acting on behalf of an application needs to access table or index pages, it will first look for the pages in the database buffer pool area. If the page cannot be found in the buffer pool area, it will be read from disk into the buffer pool. If the buffer pool is full, DB2 must select a “victim” page to be overwritten in the buffer pool. If the victim page is dirty (i.e., it was changed since it was read into the buffer pool and has not been written to disk), it must first be written to disk before it can be overwritten. During the write operation, the application must wait. To reduce the likelihood that the victim page will be dirty, DB2 uses page cleaners to write dirty pages asynchronously to disk before they are chosen as victims.

An additional benefit of the page cleaners writing the dirty pages to disk is the reduction in the amount of work required in the event that there is a problem with DB2 and a database restart or recovery is required.

Each buffer pool has what is called a “dirty list” that contains a list of all dirty pages contained in the buffer pool. When the page cleaners are triggered, they will read the dirty lists for all table spaces in the database and will asynchronously write the dirty pages to disk.

How Are the Page Cleaners Triggered?

The page cleaners can be triggered in three different manners.

Dirty Page Threshold

When a page in the buffer pool is changed, it is added to the buffer pool’s dirty list. At this time, DB2 checks to see whether this addition to the dirty list exceeds the changed page threshold (aka dirty page threshold) for the buffer pool. If the changed page threshold is exceeded, the page cleaners will be triggered.

The changed page threshold database configuration parameter (CHNGPGS_THRESH) represents the percentage of the buffer pool that can be dirty before the page cleaners are triggered.

LSN Gap

When transactions are occurring against the database, they will be logged. To reduce the amount of work required in the event of a problem, DB2 will trigger the page cleaners as it writes to the log file(s).
The percentage of the log file reclaimed before soft checkpoint database configuration parameter (SOFTMAX) represents the percentage of a log file that is written before the page cleaners are triggered.

Dirty Page Steals
When an agent requests a page that must be read from disk and DB2 chooses the victim page, if the page is dirty, the page must first be written to disk before it can be used to read the new page that the agent has requested. After a number of dirty victim pages have been selected, DB2 will automatically trigger the page cleaners to write the dirty pages to disk.

How the Page Cleaners Work
When the page cleaners are triggered, all of the page cleaners are triggered at the same time. They will each gather up to 400 pages from the dirty lists for the database buffer pools. The pages from the dirty list will then be written to disk one page at a time until the page cleaner has processed its assigned dirty pages. Once it has written all of the pages, it will check to see whether there are more pages to be written or whether there have been any new triggers. If so, it will gather a new list of pages to process; if not, it will wait for the next page cleaner to trigger.

Choosing the Number of Page Cleaners
Because all page cleaners are started whenever a page cleaner trigger is hit, having too many page cleaners can overwhelm the run queue on the server and cause a significant performance impact on the system. Therefore, as a rule of thumb, set the number of page cleaners equal to the number of CPUs in the database server.

Changed Page Threshold Tuning
The default value for the changed page threshold is 60%. For systems with large buffer pools, this can cause a noticeable slowdown of the system when the page cleaners are triggered. For example, with a 2-GB buffer pool, the changed page threshold will trigger the page cleaners when there is 1.2 GB of dirty pages in the buffer pool. Starting up a number of page cleaners to write out 1.2 GB of data to disk can cause a very noticeable slowdown. Setting the changed page threshold to a smaller value (i.e., 20–30%) will trigger the page cleaners more frequently, but they will have a smaller amount of data to write to disk and will have a much smaller impact on the performance of the system.

Monitoring Buffer Pool Activity
The database monitor can be used to monitor the various buffer pool and prefetching activities. The following entries in the buffer pool snapshot and/or the table space snapshot provide information about the page reads and writes occurring in the database:
• **Buffer Pool Data Physical Reads:** The number of data pages that were physically read from disk into the buffer pool. This includes synchronous reads done by the agents, as well as asynchronous reads that are done by the prefetchers.

• **Buffer Pool Data Logical Reads:** The number of data pages that were read from the buffer pool and from disk. To determine the number of read requests that were satisfied by the buffer pool, subtract the buffer pool data physical reads from the buffer pool data logical reads.

• **Buffer Pool Asynchronous Data Reads:** The number of data pages that were read into the buffer pool asynchronously by the prefetchers.

• **Buffer Pool Index Physical Reads:** The number of index pages that were physically read from disk into the buffer pool. This includes synchronous reads done by the agents, as well as asynchronous reads that are done by the prefetchers.

• **Buffer Pool Index Logical Reads:** The number of index pages that were read from the buffer pool and from disk. To determine the number of read requests that were satisfied by the buffer pool, subtract the buffer pool index physical reads from the buffer pool index logical reads.

• **Buffer Pool Asynchronous Index Reads:** The number of index pages that were read into the buffer pool asynchronously by the prefetchers.

• **Buffer Pool Data Writes:** The number of data pages that were written out from the buffer pool to disk. This includes synchronous writes done by the agents, as well as asynchronous writes done by the page cleaners and agents as a result of victim selection.

• **Buffer Pool Asynchronous Data Writes:** The number of data pages that were written out from the buffer pool to disk asynchronously by the page cleaners as a result of victim selection.

• **Buffer Pool Index Writes:** The number of index pages that were written out from the buffer pool to disk. This includes synchronous writes done by the agents, as well as asynchronous writes done by the page cleaners as a result of victim selection.

• **Buffer Pool Asynchronous Index Writes:** The number of index pages that were written out from the buffer pool to disk asynchronously by the page cleaners as a result of victim selection.

• **Total Buffer Pool Physical Read Time:** The total elapsed time spent processing read requests that caused data or index pages to be physically read from disk into the buffer pool. This includes synchronous reads done by the agents, as well as asynchronous reads done by the prefetchers.

• **Buffer Pool Asynchronous Read Time:** The total elapsed time spent by the prefetchers processing read requests that caused data or index pages to be physically read from disk into the buffer pool.

• **Total Buffer Pool Physical Write Time:** The total elapsed time spent processing write requests that caused data or index pages to be written out from the buffer pool to disk.
This includes synchronous writes done by the agents, as well as asynchronous writes done by the page cleaners as a result of victim selection.

- **Buffer Pool Asynchronous Write Time:** The total elapsed time spent writing data or index pages from the buffer pool to disk by page cleaners as a result of victim selection.

- **Buffer Pool Asynchronous Read Requests:** The total number of asynchronous read requests handled by the prefetchers.

- **Time Waited for Prefetch:** The total elapsed time that an agent spent waiting for a prefetcher to finish reading pages into the buffer pool.

In DB2 Version 8, the snapshot information can also be captured using SQL table functions. For each of the database snapshot monitors, there is an equivalent snapshot table function that can be used to obtain the same information. The SQL table functions normally require two input parameters, the database name and the database partition number. However, the database manager snapshot SQL table functions require only the partition number.

If no value is specified for the database name, the information will be captured for the database to which the application is currently connected. If a value of negative one (−1) is specified for the database partition number, the information will be captured for the database partition to which the application is currently connected. If a value of negative two (−2) is specified for the database partition number, the information will be captured for all of the database partitions in the database.

For example:

```sql
SELECT BP_NAME, 
     (((1 - ((FLOAT(pool_Index_P_Reads + pool_data_P_Reads)) / 
        (FLOAT(Pool_Index_L_Reads + Pool_data_L_Reads)))) * 100)) AS BPool_Hit_Ratio 
FROM TABLE(SNAPSHOT_BP('SAMPLE',-1 ))
```

will capture the snapshot information for the database partition the application is connected to, and

```sql
SELECT BP_NAME, 
     (((1 - ((FLOAT(pool_Index_P_Reads + pool_data_P_Reads)) / 
        (FLOAT(Pool_Index_L_Reads + Pool_data_L_Reads)))) * 100)) AS BPool_Hit_Ratio 
FROM TABLE(SNAPSHOT_BP('SAMPLE',-2 ))
```

will capture the snapshot information for all database partitions in the database.

**Buffer Pool Tuning**

After obtaining a buffer pool and/or table space snapshot for the database, the entries described above need to be analyzed to determine whether the database is operating efficiently. The information can also be examined using SQL table functions.
The buffer pool hit ratios are measures of the effectiveness of the buffer pool. The hits ratios reflect the number of times that a page request was able to be handled by the buffer pool directly without the need to read the page from disk. The more often that the request for a page can be satisfied from the buffer pool, the better that the overall performance of the system will be.

The following is a sample of the output of the buffer pool snapshot:

<table>
<thead>
<tr>
<th>Database Snapshot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database name                          = SAMPLE</td>
</tr>
<tr>
<td>Database path                          = /v1/db2/NODE0000/SQL00001/</td>
</tr>
<tr>
<td>Input database alias                   = SAMPLE</td>
</tr>
<tr>
<td>Database status                        = Active</td>
</tr>
<tr>
<td>Catalog database partition number      = 0</td>
</tr>
<tr>
<td>Catalog network node name              =</td>
</tr>
<tr>
<td>Operating system running at database server = NT</td>
</tr>
<tr>
<td>Location of the database               = Local</td>
</tr>
<tr>
<td>First database connect timestamp       = 09-04-2002 13:21:52.797473</td>
</tr>
<tr>
<td>Last reset timestamp                   =</td>
</tr>
<tr>
<td>Last backup timestamp                  =</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bufferpool Snapshot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bufferpool name                        = IBMDEFAULTBP</td>
</tr>
<tr>
<td>Database name                          = SAMPLE</td>
</tr>
<tr>
<td>Database path                          = /v1/db2/NODE0000/SQL00001/</td>
</tr>
<tr>
<td>Input database alias                   = SAMPLE</td>
</tr>
<tr>
<td>Buffer pool data logical reads         = 523956</td>
</tr>
<tr>
<td>Buffer pool data physical reads        = 33542</td>
</tr>
<tr>
<td>Buffer pool data writes                = 288</td>
</tr>
<tr>
<td>Buffer pool index logical reads        = 257949</td>
</tr>
<tr>
<td>Buffer pool index physical reads       = 11323</td>
</tr>
<tr>
<td>Total buffer pool read time (ms)       = 12012</td>
</tr>
<tr>
<td>Total buffer pool write time (ms)      = 720</td>
</tr>
<tr>
<td>Asynchronous pool data page reads      = 5227</td>
</tr>
<tr>
<td>Asynchronous pool data page writes     = 276</td>
</tr>
<tr>
<td>Buffer pool index writes               = 255</td>
</tr>
<tr>
<td>Asynchronous pool index page reads     = 451</td>
</tr>
<tr>
<td>Asynchronous pool index page writes    = 239</td>
</tr>
<tr>
<td>Total elapsed asynchronous read time   = 819</td>
</tr>
<tr>
<td>Total elapsed asynchronous write time  = 663</td>
</tr>
<tr>
<td>Asynchronous read requests             = 3553</td>
</tr>
<tr>
<td>Direct reads                           = 69664</td>
</tr>
</tbody>
</table>
Buffer Pool Hit Ratio

If DB2 needs to read an index or data page and the page is already in the buffer pool, the ability to access the page will be much faster than if the page has to be read from disk. The buffer pool hit ratio describes how frequently a request for an index or data page is handled directly from the buffer pool. The buffer pool hit ratio is calculated using the following formula:

\[
BPHR = \left(1 - \frac{(\text{Data Physical Reads} + \text{Index Physical Reads})}{(\text{Data Logical Reads} + \text{Index Logical Reads})}\right) \times 100\%
\]

Based on the above buffer pool snapshot, the buffer pool hit ratio would be:

\[
BPHR = \left(1 - \frac{(33542 + 11323)}{(523956 + 257949)}\right) \times 100\% \\
BPHR = 94.26\%
\]

In this case, 94.26% of the data and index page requests were able to be handled by the buffer pool, without the need for a physical I/O request.

The buffer pool hit ratio can also be calculated using the SQL table function as follows:

```sql
select BP_NAME, 
  (((1 - (FLOAT(pool_Index_P_Reads + pool_data_P_Reads)) / 
  (FLOAT(Pool_Index_L_Reads + Pool_data_L_Reads))) * 100)) AS 
  BPool_Hit_Ratio FROM TABLE(SNAPSHOT_BP('SAMPLE',-1 ))
```

```sql
```
The output of this statement would be the following:

<table>
<thead>
<tr>
<th>BP_NAME</th>
<th>BPOOL_HIT_RATIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBMDEFAULTBP</td>
<td>+9.42652009389671E+001</td>
</tr>
</tbody>
</table>

To make the output more readable, the buffer pool hit ratio can be cast as an integer. This will round the value to the integer portion of its value but is close enough for most purposes. In this case, the SQL statement to calculate the buffer pool hit ratio can be changed to:

```
select BP_NAME,
    (INT((1 - ((FLOAT(pool_Index_P_Reads + pool_data_P_Reads)) / (FLOAT(Pool_Index_L_Reads + Pool_data_L_Reads))) * 100)) AS BPool_Hit_Ratio
FROM TABLE(SNAPSHOT_BP('SAMPLE',-1 ))
as SNAPSHOT_BP;
```

The output of this statement would be the following:

<table>
<thead>
<tr>
<th>BP_NAME</th>
<th>BPOOL_HIT_RATIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBMDEFAULTBP</td>
<td>94</td>
</tr>
</tbody>
</table>

**Index Hit Ratio**

It is also important to examine the individual hit ratios, such as the data and index hit ratios. The index hit ratio is calculated using the following formula:

\[
IHR = (1 - (\text{Index Physical Reads} / \text{Index Logical Reads})) \times 100\%
\]

The index hit ratio can also be calculated using the SQL table function as follows:

```
select BP_NAME,
    (INT((1 - ((FLOAT(Pool_Index_P_Reads)) / (FLOAT(Pool_Index_L_Reads)))) * 100)) AS Index_Hit_Ratio
FROM TABLE(SNAPSHOT_BP('SAMPLE',-1 ))
as SNAPSHOT_BP;
```

**Data Hit Ratio**

The data hit ratio is calculated using the following formula:

\[
DHR = (1 - (\text{Data Physical Reads} / \text{Data Logical Reads})) \times 100\%
\]
The data hit ratio can also be calculated using the SQL table function as follows:

```sql
select BP_NAME,
   (INT((1 - ((FLOAT(pool_Data_P_Reads)) / (FLOAT(Pool_Data_L_Reads)))) * 100))
   AS Data_Hit_Ratio
FROM TABLE(SNAPSHOT_BP('SAMPLE',-1 ))
as SNAPSHOT_BP;
```

Based on the above buffer pool snapshot, the index and data hit ratios would be:

```
IHR = (1 - (11323) / 257949)) * 100%
IHR = 95.61%

DHR = (1 - (33542) / 523956)) * 100%
DHR = 93.60%
```

In general, a buffer pool hit ratio above 80% is considered good. However, for an OLTP system, it is important to have the buffer pool hit ratio as high as possible (especially the index hit ratio) to be able to respond to requests efficiently and quickly. For large DSS systems, it is very unlikely that the buffer pool hit ratio will be high, due to the vast amounts of data that are normally read. Therefore, it is important to understand the workload on the system when analyzing the buffer pool hit ratios, so that they can be examined in the right context.

**Asynchronous Read Ratio**

Another important aspect of the buffer pool performance that can be analyzed using the snapshot information is the amount of synchronous versus asynchronous I/O. The percentage of asynchronous read requests (or Asynchronous Read Ratio) is calculated using the following formula:

```
ARR = ((Asynch Data Reads + Asynch Index Reads) / ((Data Logical Reads + Index Logical Reads)) * 100%
```

The asynchronous read ratio can also be calculated using the SQL table function as follows:

```sql
select BP_NAME,
   (INT(((FLOAT(pool_Async_data_Reads + pool_async_index_Reads)) / (FLOAT(Pool_Index_L_Reads + Pool_data_L_Reads))) * 100))
   AS Asynch_Read_Ratio
FROM TABLE(SNAPSHOT_BP('SAMPLE',-1 ))
as SNAPSHOT_BP;
```

Based on the above buffer pool snapshot, the asynchronous read ratio would be:

```
ARR = ((5227 + 451) / (523956 + 257949)) * 100%
ARR = 0.73%
```
This is a very small value and would indicate that there is very little prefetch activity occurring for this database. This could be due to a number of reasons, such as:

1. The workload is reading and writing single rows, so it cannot take advantage of prefetching.
2. There are too few prefachers configured for the database.
3. The table spaces in the database are set up with only one container each so that prefetching cannot normally take place.

| NOTED | When DB2_PARALLEL_IO is set to YES, prefetching can occur within a single container table space if the prefetch size is a multiple of the extent size. |

For a system with multiple buffer pools, it is normally a good idea to separate tables with a high percentage of asynchronous reads from those with a low percentage of asynchronous reads. The asynchronous read ratio can also be examined for each table space to help separate the table spaces with high and low asynchronous read ratios. For the following table space snapshot information, we see that there are four table spaces with different access patterns:

<table>
<thead>
<tr>
<th>Tablespace name</th>
<th>Buffer pool data logical reads</th>
<th>Asynchronous pool data page reads</th>
<th>Buffer pool index logical reads</th>
<th>Asynchronous pool index page reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSPC1</td>
<td>1200</td>
<td>32</td>
<td>3400</td>
<td>128</td>
</tr>
<tr>
<td>TSPC2</td>
<td>15000</td>
<td>14000</td>
<td>90000</td>
<td>86000</td>
</tr>
<tr>
<td>TSPC3</td>
<td>9000</td>
<td>8600</td>
<td>6250</td>
<td>5975</td>
</tr>
<tr>
<td>TSPC4</td>
<td>7200</td>
<td>1400</td>
<td>800</td>
<td>770</td>
</tr>
</tbody>
</table>
In this case, the asynchronous read ratios would be:

<table>
<thead>
<tr>
<th>Table Space</th>
<th>Read Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBSPC1</td>
<td>3.5%</td>
</tr>
<tr>
<td>TBSPC2</td>
<td>95.2%</td>
</tr>
<tr>
<td>TBSCP3</td>
<td>95.6%</td>
</tr>
<tr>
<td>TBSPC4</td>
<td>27.1%</td>
</tr>
</tbody>
</table>

Because the table spaces TBSPC1 and TBSPC4 both have low asynchronous read ratios, they should not be placed in the same buffer pool as table space TBSPC2 or TBSPC3. Because the table spaces TBSPC2 and TBSPC3 both have a high asynchronous read ratio, they could be placed in the same buffer pool; however, because DB2 places an internal limit on the number of pages that can be prefetched into a buffer pool before they are accessed by a DB2 agent, having two table spaces with a high asynchronous read ratio in the same buffer pool may have an adverse effect. It may be more optimal to place the table spaces TBSPC2 and TBSPC3 in their own buffer pools.

**Physical Read Rate**

Also important is the rate at which DB2 is reading pages from disk. This should be calculated for all table spaces and, when compared, will show whether the I/O is spread evenly across all table spaces or whether the workload on certain table spaces is causing more I/O than in other table spaces.

The rate at which pages are read from disk (or Page Read Rate) is calculated using the following formula:

$$PRR = \frac{(\text{Data Physical Reads} + \text{Index Physical Reads})}{(\text{Time since monitor switches reset or activated})}$$

Based on the above buffer pool snapshot, the page read rate would be:

$$PRR = \frac{(33542 + 11323)}{(23.53 \text{ seconds})}$$

Examining the table space snapshot for the table spaces using the identified buffer pool may provide additional information to help determine which table space(s) is being read most often. Any table space(s) with a significantly higher I/O rate than other table spaces can be examined to determine whether the performance could be improved by assigning the table space to its own buffer pool or by adding containers to the table space to improve the I/O bandwidth.

**Read Time**

For every millisecond that a DB2 agent spends waiting for a page to be read into the buffer pool, the application is also waiting. The database snapshots do not provide information on the amount of time taken by each read request; however, they do provide enough information to cal-
calculate the average time taken per read request. The average read time is calculated using the following formula:

\[
\text{ART} = \frac{\text{Total Buffer Pool Read Time}}{\text{Data Physical Reads} + \text{Index Physical Reads}}
\]

The average read time can also be calculated using the SQL table function as follows:

```sql
select BP_NAME,
    (INT(((FLOAT(pool_read_time)) / (FLOAT(Pool_Index_p_Reads + Pool_data_p_Reads))) * 100))
    AS Avg_Read_Time_in_ms
FROM TABLE(SNAPSHOT_BP('SAMPLE',-1 ))
as SNAPSHOT_BP;
```

The SQL table function can also be used to calculate the buffer pool, data and index hit ratios, as well as the asynchronous read ratio and average read time in one SQL statement. This can be done using the following statement:

```sql
select BP_NAME,
    (INT(((1 - ((FLOAT(pool_Index_P_Reads + pool_data_P_Reads)) / (FLOAT(Pool_Index_L_Reads + Pool_data_L_Reads)))) * 100))
    AS BPool_Hit_Ratio,
    (INT((1 - ((FLOAT(pool_Data_P_Reads)) / (FLOAT(Pool_Data_L_Reads)))) * 100))
    AS Data_Hit_Ratio,
    (INT((1 - ((FLOAT(pool_Index_P_Reads)) / (FLOAT(Pool_Index_L_Reads)))) * 100))
    AS Index_Hit_Ratio,
    (INT(((FLOAT(pool_Async_data_Reads + pool_async_index_Reads)) / (FLOAT(Pool_Index_L_Reads + Pool_data_L_Reads))) * 100))
    AS Asynch_Read_Ratio,
    (INT(((FLOAT(pool_read_time)) / (FLOAT(Pool_Index_p_Reads + Pool_data_p_Reads))) * 100))
    AS Avg_Read_Time_in_ms
FROM TABLE(SNAPSHOT_BP('SAMPLE',-1 ))
as SNAPSHOT_BP;
```

The output of this statement looks like the following:

<table>
<thead>
<tr>
<th>BP_NAME</th>
<th>BPOOL_HIT_RATIO</th>
<th>DATA_HIT_RATIO</th>
<th>INDEX_HIT_RATIO</th>
<th>ASYNCH_READ_RATIO</th>
<th>AVG_READ_TIME_IN_MS</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBMDEFAULTBP</td>
<td>69</td>
<td>78</td>
<td>63</td>
<td>0</td>
<td>362</td>
</tr>
</tbody>
</table>

1 record(s) selected.
Page Cleaner Triggers

It is also important to understand which of the three triggers is causing the page cleaners to be activated and write the dirty pages from the buffer pools to disk. This information is available in the database snapshot information or through an SQL table function. The entries that describe the page cleaner triggers in the database snapshot are:

- LSN Gap cleaner triggers = 142
- Dirty page steal cleaner triggers = 2
- Dirty page threshold cleaner triggers = 396

The SQL table function that will return the page cleaner triggers would look like the following:

```sql
SELECT DB_NAME,
      POOL_LSN_GAP_CLNS,
      POOL_DRTY_PG_STEAL_CLNS,
      POOL_DRTY_PG_THRSH_CLNS
FROM TABLE(SNAPSHOT_DATABASE('SAMPLE',-1))
```

The output of the SQL function would look like the following:

<table>
<thead>
<tr>
<th>DB_NAME</th>
<th>POOL_LSN_GAP_CLNS</th>
<th>POOL_DRTY_PG_STEAL_CLNS</th>
<th>POOL_DRTY_PG_THRSH_CLNS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAMPLE</td>
<td>142</td>
<td>2</td>
<td>396</td>
</tr>
</tbody>
</table>

In this case, the page cleaners have been triggered by the “good” triggers, i.e., changed page threshold or LSN gap (softmax) well over 99% of the time. As was explained earlier, a dirty page steal trigger is done only after a number of pages have been synchronously written to disk and their associated clients forced to wait. If the number of “bad” page cleaner triggers (i.e., dirty page steal triggers) is more than a small percentage of the total number of triggers, the values set for changed page threshold and softmax, as well as the number of page cleaners, should be examined.

The percentage of bad page cleaner triggers is calculated as follows:

\[
PBPCT = \left( \frac{\text{Dirty page steal cleaner triggers}}{\text{Dirty page steal cleaner triggers} + \text{Dirty page threshold cleaner triggers} + \text{LSN Gap Cleaner Triggers}} \right) \times 100\%
\]

Based on the snapshot information above, the percentage of bad page cleaner triggers equals:

\[
PBPCT = \left( \frac{2}{142 + 396 + 2} \right) \times 100\%
\]

\[
PBPCT = 0.37\%
\]

This ratio is very good and indicates that the system is primarily writing dirty pages to disk, using the asynchronous page cleaners.
The Buffer Pools

However, based on the following snapshot information for the page cleaner triggers, the percentage of bad page cleaner triggers is much higher.

<table>
<thead>
<tr>
<th>DB_NAME</th>
<th>POOL_LOSN_GAP_CLNS</th>
<th>POOL_DRTY_PG_STEAL_CLNS</th>
<th>POOL_DRTY_PG_THRSH_CLNS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAMPLE</td>
<td>17</td>
<td>2034</td>
<td>1192</td>
</tr>
</tbody>
</table>

1 record(s) selected.

The percentage of bad page cleaner triggers equals:

\[
PBPCT = \frac{(2034)}{(17 + 1192 + 2034)} \times 100\% 
\]

\[
PBPCT = 62.7\% 
\]

In this case, the page cleaners are rarely being triggered by the pool LSN gap trigger, which indicates that the database configuration parameter softmax may be set too high. To determine the value of the softmax configuration variable, use the command:

```
get db cfg for sample | grep –i softmax
```

This returns the following:

```
Percent log file reclaimed before soft chckpt (SOFTMAX) = 100
```

In this case, the page cleaners are being triggered each time a log file is filled. Because this value is not abnormally high, next examine the log file size, using the command:

```
get db cfg for sample | grep –i logfilsiz
```

This returns the following:

```
Log file size (4KB) (LOGFILSIZ) = 250000
```

The log file size for this database is 250,000 4-KB pages, or 1 GB. Therefore, the page cleaners are being triggered only after 1 GB of log information has been written. If the log file size cannot be reduced, the softmax configuration parameter can be reduced to cause the page cleaners to be triggered more frequently. To update the softmax configuration parameter to cause the page cleaners to trigger after 10% of a log has been written, use the following command:

```
update db cfg for sample using softmax 10
```

If the log files do not need to be this large and can be reduced, the log file size can be changed to 250 4-KB pages, or 1 MB, using the following command:

```
update db cfg for sample using logfilsiz 250
```

Asynchronous Pages per Write

When the page cleaners are triggered, it is important that they be writing to disk as efficiently as possible. Having the page cleaners triggered too infrequently and writing a large number of
pages to disk will cause the system to slow down. Likewise, having the page cleaners triggered frequently but writing a small number of pages to disk is also inefficient.

The number of pages written per page cleaner trigger is not captured in any of the DB2 snapshots; however, the average number of pages written per asynchronous write request can be calculated using the database base and buffer pool snapshot information. The average pages per asynchronous write can be calculated using the formula:

\[
\text{APPAW} = \frac{(\text{Asynchronous pool data page writes} + \text{Asynchronous pool index page writes})}{(\text{Dirty page steal cleaner triggers} + \text{Dirty page threshold cleaner triggers} + \text{LSN Gap Cleaner Triggers})}
\]

Based on the following information from the database and buffer pool snapshots:

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSN Gap cleaner triggers</td>
<td>142</td>
</tr>
<tr>
<td>Dirty page steal cleaner triggers</td>
<td>2</td>
</tr>
<tr>
<td>Dirty page threshold cleaner triggers</td>
<td>396</td>
</tr>
<tr>
<td>Asynchronous pool data page writes</td>
<td>167660</td>
</tr>
<tr>
<td>Asynchronous pool index page writes</td>
<td>178944</td>
</tr>
</tbody>
</table>

the average pages per asynchronous write would be:

\[
\text{APPAW} = \frac{167660 + 178944}{142 + 2 + 396}
\]

\[
\text{APPAW} = 641.9
\]

In this case, the page cleaners wrote an average of 641.9 pages, or 2.5 MB, each time they were triggered. This value needs to be examined in the context of the size of the buffer pool that is being examined. For a 1-GB buffer pool, this is a small value, and perhaps the page cleaners are being triggered too aggressively. For a 100-MB buffer pool, this value is much more reasonable.

**PARALLELISM**

The exploitation of parallelism within a database and within an application accessing a database can also have a significant benefit for overall database performance, as well as the normal administrative tasks. There are two types of query parallelism that are available with DB2 UDB: *inter-query* parallelism and *intra-query* parallelism.

*Inter-query* parallelism refers to the ability of multiple applications to query a database at the same time. Each query will execute independently of the others, but DB2 UDB will execute them at the same time.

*Intra-query* parallelism refers to the ability to break a single query into a number of pieces and replicate them at the same time using either *intra-partition* parallelism or *inter-partition* parallelism, or both.
Intra-Partition Parallelism

Intra-partition parallelism refers to the ability to break up a query into multiple parts within a single database partition and execute these parts at the same time. This type of parallelism subdivides what is usually considered a single database operation, such as index creation, database load, or SQL queries into multiple parts, many or all of which can be executed in parallel within a single database partition. Intra-partition parallelism can be used to take advantage of multiple processors of a symmetric multiprocessor (SMP) server.

Intra-partition parallelism can take advantage of either data parallelism or pipeline parallelism. Data parallelism is normally used when scanning large indexes or tables. When data parallelism is used as part of the access plan for an SQL statement, the index or data will be dynamically partitioned, and each of the executing parts of the query (known as package parts) is assigned a range of data to act on. For an index scan, the data will be partitioned based on the key values, whereas for a table scan, the data will be partitioned based on the actual data pages.

Pipeline parallelism is normally used when distinct operations on the data can be executed in parallel. For example, a table is being scanned and the scan is immediately feeding into a sort operation that is executing in parallel to sort the data as it is being scanned.

Figure 2.2 shows a query that is broken into four pieces that can be executed in parallel, each working with a subset of the data. When this happens, the results can be returned more quickly than if the query was run serially. To utilize intra-partition parallelism, the database must be configured appropriately.

Intra-partition parallelism must be enabled for the DB2 instance before the queries can be executed in parallel. Once intra-partition parallelism is enabled, the degree of parallelism, or number of pieces of the query that can execute in parallel, can be controlled using database configuration parameters.

![Figure 2.2](Snow_Ch02.fm Page 113 Friday, June 13, 2003 10:16 AM)

Figure 2.2
Intra-partition parallelism.
Configuring Intra-Partition Parallelism

Intra-partition parallelism in DB2 UDB is enabled or disabled using the database manager configuration parameter INTRA_PARALLEL. To enable intra-partition parallelism in DB2 UDB, the INTRA_PARALLEL configuration must be set to YES. This can be done using the following command:

```
UPDATE DBM CFG USING INTRA_PARALLEL YES
```

The degree of parallelism can then be controlled at the instance level, the database level, the application, or the statement level. The degree of parallelism can be set to a specific value or to ANY. If the degree of parallelism is set to ANY, the optimizer will determine the degree of parallelism for each individual SQL query that is submitted, based on the query itself and the number of CPUs available to the database or database partition.

Table 2.2 gives an overview of the parameters and options that are related to intra-partition parallelism in DB2 UDB.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRA_PARALLEL</td>
<td>YES/NO</td>
</tr>
<tr>
<td></td>
<td>Defaults to NO on uni-processor machine</td>
</tr>
<tr>
<td></td>
<td>Defaults to YES on SMP machine</td>
</tr>
<tr>
<td></td>
<td>If changed, packages already bound will automatically be rebound at next execution.</td>
</tr>
<tr>
<td>MAX_QUERYDEGREE</td>
<td>1–32767, ANY</td>
</tr>
<tr>
<td></td>
<td>Defaults to ANY; allows optimizer to choose degree of parallelism based on cost.</td>
</tr>
<tr>
<td></td>
<td>No SQL executed on a database in this instance can use a degree of parallelism higher than this value.</td>
</tr>
<tr>
<td>DFT_DEGREE</td>
<td>1–32767, ANY</td>
</tr>
<tr>
<td></td>
<td>Defaults to 1 (no parallelism)</td>
</tr>
<tr>
<td></td>
<td>Provides the default value for:</td>
</tr>
<tr>
<td></td>
<td>CURRENT DEGREE special register</td>
</tr>
<tr>
<td></td>
<td>DEGREE bind option</td>
</tr>
<tr>
<td></td>
<td>Maximum for any SQL in this database</td>
</tr>
<tr>
<td>CURRENT DEGREE</td>
<td>1–32767, ANY</td>
</tr>
<tr>
<td></td>
<td>Sets degree of parallelism for dynamic SQL</td>
</tr>
<tr>
<td></td>
<td>Defaults to DFT_DEGREE</td>
</tr>
</tbody>
</table>
Parallelism

The maximum degree of parallelism for an active application can be specified using the SET RUNTIME DEGREE command. The application can set its own runtime degree of parallelism by using the SET CURRENT DEGREE statement. The actual runtime degree used is the lower of:

- MAX_QUERYDEGREE instance configuration parameter
- Application runtime degree
- SQL statement compilation degree

More information on parallelism support in DB2 Universal Database can be found in the *DB2 UDB Administration Guide: Performance*.

For a multi-partitioned database on a large SMP server, the maximum degree of parallelism for each partition should be limited so that each partition does not attempt to use all of the CPUs on the server. This can be done using the MAX_QUERYDEGREE instance configuration parameter. For a 32-way SMP server with eight database partitions, the maximum degree of parallelism for each partition could be limited to four, as follows:

```
UPDATE DBM CFG USING MAX_QUERYDEGREE 4
```

For an SMP server with 16 CPUs, running two separate DB2 instances, the maximum degree of parallelism for each partition could be limited to eight, as follows:

```
UPDATE DBM CFG USING MAX_QUERYDEGREE 8
```
for a DB2 instance with two databases that has intra-partition parallelism enabled. The benefit of intra-partition parallelism is very different if one database is a data mart or data warehouse and has large, complex queries scanning a large amount of data and the other database is used as a journal and is accessed using only INSERT statements. In this case, the default degree of parallelism can be set to different values for each database. If the databases are named DMDB and JRNLDB, this can be done as follows:

```
UPDATE DB CFG for DMDB USING DFT_DEGREE 8
UPDATE DB CFG for JRNLDB USING DFT_DEGREE 1
```

For CLI, ODBC, and JDBC applications, the degree of parallelism is controlled using the `db2cli.ini` file. The following is an example `db2cli.ini` file where any application that connects to the SAMPLE database will use a degree of parallelism of four.

```
[common]
TRACE=1
TRACECOMM=1
TRACEFLUSH=1
TRACEPATHNAME=d:\trace

; Comment lines start with a
semi-colon.

[sample]
DBALIAS=MYSAMP
DB2DEGREE=4
autocommit=0
```

To change the degree of parallelism for a currently executing SQL statement, the SYSADM can change the run time degree for an application. For an application with an application ID of 130, to change the degree of parallelism to 2, the following command can be used:

```
SET RUNTIME DEGREE FOR (130) to 2
```

**NOTE** This change cannot affect the currently executing SQL statement but will be effective for all subsequent SQL statements.

### Applications and Intra-Partition Parallelism

When an application connects to a nonpartitioned database, the connection is assigned to a coordinating agent, and all subsequent SQL statements and DB2 commands are executed by the coordinating agent on behalf of the application. If intra-partition parallelism is enabled, when an SQL statement is executed, the coordinating agent will determine the degree of parallelism to be used to execute the statement based on the access plan and the database and instance configura-
tion parameters. The coordinating agent will then make \( n \) copies of the access plan, where \( n \) is the degree of parallelism and will execute each copy of the plan on its own subagent. Therefore, for an access plan with a degree of parallelism of eight, there will be eight subagents and one coordinating agent used to execute the plan and return the data.

On a UNIX system, the coordinating agent and subagent processes will have different names. The coordinating agent process will be named `db2agent`, and the subagent process will be named either `db2agntp` or `db2agnts`, depending on whether the process is currently idle or working. An idle subagent will be named `db2agnts`, and a working subagent will be named `db2agntp`. In Figure 2.3, coordinating agent A has two subagents associated to it; therefore, the statement that it is currently executing has a degree of parallelism of 2. If coordinating agent A next runs an SQL statement with a degree of parallelism of 3, it must acquire another subagent in order to execute the access plan, as depicted in Figure 2.4.

If another application connects to the database and executes an SQL statement that has a degree of parallelism of 1, the agent pool would look like Figure 2.5. In this case, there will be three subagent processes associated with coordinating agent A and one subagent process associated with coordinating agent B.

The subagents are called *associated* because they have an affiliation with the coordinating agent and have control blocks and shared inter process communication (IPC) blocks set up to communicate between the coordinating agent and its associated subagents.
Figure 2.4
Degree of parallelism of 3.

Figure 2.5
Two applications executing with varying degrees of parallelism.
If the application that is connected to coordinating agent B finishes its current SQL statement, its associated subagent will become idle but will remain associated with its coordinating agent. It remains associated because there is a chance that it may be needed in the future for a subsequent statement, and it is inefficient to break its association to release it back to the idle agent pool if it may be needed again.

If the application connected to coordinating agent A completes the current statement and runs a subsequent statement with a degree of parallelism of 2, one of its subagents will become idle but will remain associated with coordinating agent A. The agent pool would then look like that in Figure 2.6.

Figure 2.6 shows all of the states that agents and subagents can be in within the agent pool. Within the pool, there are three idle agents waiting to be called on by either of the existing coordinating agents or by a subsequent application connecting to the database. Coordinating agent B is idle, but its application has not yet disconnected from the database, so the agent and its associated subagents will remain to service any subsequent SQL statements or until the application disconnects. If the application disconnects, the coordinating agent and its subagents will be returned to the agent pool as idle agents.
Coordinating agent A is currently executing an SQL statement with a degree of parallelism of 2; therefore, the coordinating agent and two of its associated subagents are busy. The third subagent is idle but remains associated to coordinating agent A.

If the current statement executing in coordinating agent A completes and the next SQL statement has a degree of parallelism of 8, the coordinating agent must make active and/or acquire a number of additional subagents. The activation and acquiring of these subagents will be done in an order that helps ensure the optimal usage of the agents and available resources. The agents will be acquired in the following order:

1. Convert any idle subagents currently associated to the coordinating agent to active.
2. Convert any idle subagents not currently associated to another coordinating agent to active (i.e., in the idle agent pool).
3. Convert any idle subagents currently associated to another coordinating agent to active.
4. Create a new agent process.

For this example, the idle subagent currently associated with coordinating agent A would be converted to active. The three idle agents in the pool would be associated with coordinating agent A and made active. The idle subagent currently associated with coordinating agent B would be unassociated from coordinating agent B, associated with coordinating agent A, and made active. One additional agent process would need to be created, associated with coordinating agent A and made active; then the SQL statement could be processed.

For a large, complex query, intra-partition parallelism is good, and the result set can normally be returned much quicker this way. However, an insert of a single does not require nine (eight subagents plus the coordinating agent) agents. In a typical OLTP environment, there are normally hundreds (if not more) of applications running in parallel. If each of them were assigned nine agent processes to handle their simple tasks, there would be far too many agent processes running on the system, causing a great deal more system overhead than is required.

In addition, when intra-partition parallelism is enabled, an instance level control block is set up by DB2, and each statement that is performed on any database in the DB2 instance will require a check of this control block during the statement’s optimization and execution. Even if intra-partition parallelism is disabled for a database by setting the maximum degree of parallelism or default degree of parallelism to 1, this control block will still be checked during the optimization and execution of each statement. Therefore, it is best to disable intra-partition parallelism, using the database manager configuration parameter INTRA_PARALLEL.

**Inter-Partition Parallelism**

Inter-partition parallelism refers to the ability to break up a query into multiple parts across multiple partitions of a partitioned database on a single server or between multiple servers. The query will be executed in parallel on all of the database partitions. Inter-partition parallelism can
be used to take advantage of multiple processors of an SMP server or multiple processors spread across a number of servers.

Figure 2.7 shows a query that is broken into four pieces that can be executed in parallel, with the results returned more quickly than if the query was run in a serial fashion in a single partition. In this case, the degree of parallelism for the query is limited by the number of database partitions.

**Combined Intra-Partition Parallelism and Inter-Partition Parallelism**

In a partitioned database, intra-partition parallelism and inter-partition parallelism can be combined and used at the same time (Figure 2.8). This combination provides, in effect, two dimensions of parallelism. This results in an even more dramatic increase in the speed at which queries are processed. Combined intra-partition and inter-partition parallelism can be used to take advantage of multiple processors of a single or multiple SMP servers.

**Why Partition a Database on a Large SMP Server?**

Although DB2 can break a query into a number of pieces that can be executed in parallel, the scalability of DB2 with intra-partition parallelism can be limited or restricted only by the operating system and hardware on which DB2 is installed.

Creating multiple database partitions within a larger SMP server has proven to provide better scalability than intra-partition parallelism alone. As the number of CPUs grows, the scalability
decreases when using intra-partition parallelism alone. By creating multiple database partitions within the SMP server, the scalability will remain almost linear. In a recent test of DB2 with multiple database partitions on a single SMP server growing from 6 to 24 CPUs (Figure 2.9), DB2 was able to deliver a speedup of 23.9 times on query response time.

**Considerations with Multiple Database Partitions**

In a partitioned database, the partition or partitions that will accept connection requests from clients are known as *coordinator partitions*. Because DB2 will automatically route all requests to the underlying database partitions, not all partitions need to be configured as coordinator partitions. In
fact, when DB2 is configured with multiple database partitions within a single server, only one database partition per server can be a coordinator. The first database partition on the server listed in the db2nodes.cfg file is the only database partition that can be a coordinator partition.

Because the database partitions share a common install path and home directory, the listener port for the partitions, as defined in the database manager configuration, is the same for all database partitions on the same server. Because the database partitions cannot use the same listener process, only one partition per server can be a coordinator.

This should be taken into consideration when choosing where to create database partition groups within the database. Database partition groups that will be created on a single partition may benefit from being placed on database partitions that can be coordinators. Accessing tables in a single partitioned database partition group that is not a coordinator requires the coordinator to receive the SQL statement, then to pass it to the database partition where the SQL statement will be executed. The extra overhead of the coordinator partition “handling” the request and sending it to the affected database partition is not significant if the operation is complex and accesses a large amount of data. A transactional environment, where only a single row may be updated, can result in increased response time. However, DB2 provides a method to achieve maximum scalability and throughput, even for single-row selects or updates, known as local bypass.

Local Bypass

There is an API that can be called within the application to determine on which database partition the particular row is stored so that the application can then call another API to connect directly to the identified partition and read/update/delete the row.

These APIs are:

- `sqlugtpi` – Get partition information for the table being worked on
- `sqlugrpn` – Determine the partition to connect to

There are some important considerations when using these APIs that need to be noted:

1. As noted previously, only one database partition per server can be a coordinator; therefore, even if the application is directed to another partition on the server, it will be handled by the first database partition on the server. The communication between partitions within the same server uses shared memory to reduce the overhead associated with TPC/IP.

2. These APIs can be invoked from only one of the servers identified in the db2nodes.cfg file.
   a. They cannot be called in an application running on Windows if the database is on a UNIX server.
   b. The application must be running on the same server as one of the database partitions.
Some pseudo-code to show the use of the above APIs:

```c
//create an array of connections as the connection pool to use.  
SQLHDBC hDbc[NumOfNodes];   // will have one connection for each logical node  
SQLAllocEnv(&hEnv);

for (NodeNum=0; NodeNum < NumOfNodes; NodeNum++)
{
   SQLAllocHandle(SQL_HANDLE_DBC, henv, hdbc);   // used to be SQLAllocConnect
   SQLSetConnectAttr(hDbc[NodeNum], SQL_ATTR_CONNECT_NODE, NodeNum);
   SQLConnect(hDbc[NodeNum], "Your_DB_NAME");
} // Now you have one connection for each logical node ready.

// Need to get the partition information for the table you are working on  
// See last sample for more information about this part  
strcpy(table_name, dbname);  
strcat(table_name, "table_name");

sqlugtpi( table_name, &part_info, &sqlca_inst );  
if(check_error("sqlugtpi failed", &sqlca_inst)!=0){
   exit(-1);
}
 
// Say you got your primary key value and want to see which node you want to connect and do your SQL  
// See last sample for more information about this part

key_value[0] = malloc(10);  
sprintf(key_value[0], "%d", YouKeyValue);  
key_len[0] = strlen(key_value[0]);

sqlugrpn( part_info.sqlid, key_value, key_len, ctrycode, codepage,  
&part_info,  
&part_number, &node_number, chklvl, &sqlca_inst,  
SQL_CHARSTRING_FORMAT,  
(void*)0, (void *)0 );
if(check_error("sqlugrpn failed", &sqlca_inst)!=0){
   exit(-1);
}
```

---

//create an array of connections as the connection pool to use.  
SQLHDBC hDbc[NumOfNodes];   // will have one connection for each logical node  
SQLAllocEnv(&hEnv);

for (NodeNum=0; NodeNum < NumOfNodes; NodeNum++)
{
   SQLAllocHandle(SQL_HANDLE_DBC, henv, hdbc);   // used to be SQLAllocConnect
   SQLSetConnectAttr(hDbc[NodeNum], SQL_ATTR_CONNECT_NODE, NodeNum);
   SQLConnect(hDbc[NodeNum], "Your_DB_NAME");
} // Now you have one connection for each logical node ready.

// Need to get the partition information for the table you are working on  
// See last sample for more information about this part  
strcpy(table_name, dbname);  
strcat(table_name, "table_name");

sqlugtpi( table_name, &part_info, &sqlca_inst );  
if(check_error("sqlugtpi failed", &sqlca_inst)!=0){
   exit(-1);
}
 
// Say you got your primary key value and want to see which node you want to connect and do your SQL  
// See last sample for more information about this part

key_value[0] = malloc(10);  
sprintf(key_value[0], "%d", YouKeyValue);  
key_len[0] = strlen(key_value[0]);

sqlugrpn( part_info.sqlid, key_value, key_len, ctrycode, codepage,  
&part_info,  
&part_number, &node_number, chklvl, &sqlca_inst,  
SQL_CHARSTRING_FORMAT,  
(void*)0, (void *)0 );
if(check_error("sqlugrpn failed", &sqlca_inst)!=0){
   exit(-1);
}
When configuring multiple database partitions on the same server, it is important that each database partition have sufficient resources in order to perform optimally. Each database partition contains a number of processes (or threads on Windows), and it is important to ensure that there are enough CPU cycles available to execute these processes. To ensure that there are sufficient CPU resources available, it is recommended that the ratio of database partitions to CPUs not exceed 1:1, i.e., do not create more database partitions on a server than there are CPUs in the server.

Depending on the speed and relative processing power of the CPUs on the server, a ratio of CPUs to database partitions of 1:1 or 2:1 will normally give the best performance. For most servers, a ratio of CPUs to database partitions of 2:1 will provide the best performance; however, with faster processors, such as the IBM pSeries P690 (Regatta), it is possible to reduce this ratio to 1:1 if needed.

For a database with multiple database partitions on the same server, DB2 UDB can use shared memory to communicate between the database partitions instead of using the network interconnect. The use of shared memory improves the performance of inter-partition communication but must be enabled using the DB2 registry variable DB2_FORCE_FCM_BP. Any inter-partition communication between database partitions on different servers will still use the dedicated interconnect; however, all inter-partition communication between database partitions on the same server will use shared memory, as shown in Figure 2.10.

One of the features available with the IBM pSeries P690 server is the ability to create logical servers within a large SMP server, using logical partitions (LPARs). This is a good way to separate different DB2 UDB instances within a single server and ensure that each partition gets dedicated resources. However, it may not be an optimal way of separating database partitions for the

```c
// node_number is the node you need to connect
// Then you just need to use the connection handle hDbc[node_number]
// here is more CLI stuff for running your sql, consulting the CLI documentation

SQLAllocStmt(hDbc[node_number], & hStmt);
SQLPrepare(hStmt, "Your SQL Statement String like select * from ....");
<< ... more statement specific calls to bind the columns and parameters, etc. >>
SQLExect(hStmt);
SQLFetch(.....)
SQLFreeStmt(....)
```
same database within a server. Currently, the only way to communicate between LPARs on the same server is using a network interconnect because the LPARs cannot currently communicate using any optimized methods, such as shared memory. Therefore, when creating multiple database partitions on the same IBM pSeries P690 server, it is normally best not to use LPARs so that the inter-partition communication can benefit from the use of shared memory.
An additional benefit of this approach is that when some database partitions are not as busy as others, they can take advantage of all of the resources available on the server and not be constrained to the resources assigned to the LPAR.

**Types of Database Objects**

A DB2 database is composed of data, indexes, and LOB and Long Varchar objects. When a table is created, the data object for the table is automatically created by DB2. If there are any indexes, primary keys, or unique constraints defined on the table, an index object will also be created. If the table contains one or more LOB columns, a LOB object will be created, and if the table contains one or more Long Varchar columns, a Long Varchar object will be created.

**Tables**

A table consists of data logically arranged in columns and rows. Users access the data by referring to its content instead of its location or organization in storage.

As we discussed early in this chapter, database partition groups are created within a database. Table spaces are created within database partition groups, and when tables are created, the name of the table space where the table objects will be created can be specified. If the database is partitioned, the rows of the table are distributed across the database partitions, based on the partition group’s partitioning map, via a hashing algorithm.

**Partition Maps**

As shown in Figure 2.11, DB2 uses a hashing algorithm to assign a given row of a table to a corresponding database partition. When a table is created, a partitioning key is defined for that table. This partitioning key consists of one or many columns. (See “Partitioning Keys,” later in this chapter for more details.) DB2 uses the value of the partitioning key for a given row as the input for the hashing function, and the output of the hashing function is a displacement into the partitioning map. The partitioning map is an array of 4,096 entries, each entry containing a database partition ID. The value at the specified position (displacement) in the partition map is the database partition ID where the row will be stored. Figure 2.11 shows an example of this process.

Based on the example above, a row is inserted into the table with a partitioning key value of 10004. This value is then used as input for the hashing algorithm. When the value of 10004 is hashed, the output of the algorithm is 514. DB2 then selects the value at offset 514 in the partitioning map as the database partition where the row will be stored. In this case, the row will be stored in database partition number 3.
A partitioning map is an array (or vector) of 4,096 database partition numbers. Each partitioning map corresponds to only one database partition group and is created when the database partition group is created (or when data is redistributed).

Figure 2.12 shows three additional examples of partitioning maps. A partitioning map normally consists of an array of 4,096 elements (starting at element number 0) and is formed of two rows: The first row represents the element of the array—which ranges from 0 to 4,095—and the second row is the value at that element in the array, which is a database partition number.

In the first example, partition group PG123 is created on database partitions 1, 2, and 3. Therefore, the default partitioning map is created containing only these database partitions.

In the second example, a single-partition partition group is created. The partition map, therefore, has only one entry because all of the data rows will be stored in the same database partition. In this example, this is database partition 3.

**Note**: All tables in a database partition group will use the same partitioning map.
In the third example, a partition group is created, spanning all partitions in the database (i.e., database partitions 1, 2, 3, and 4). The partitioning map associated with this partition group will then contain all of these database partition numbers.

The partitioning map is created when a partition group is created, and it is stored in the SYSCAT.PARTITIONMAPS catalog view.

The default partitioning map contains the database partition numbers of the partition group assigned in a round-robin fashion, as shown in Figure 2.12. It is also possible to create and use partitioning maps that do not use this round-robin system. These maps are known as *customized partitioning maps* and may be required to achieve an even distribution of the table’s data across the database partitions used by the table.

The utility db2gmap can be used to extract and display the partitioning map of a partition group from the SYSCAT.PARTITIONMAPS catalog view. To extract the partitioning map for the partition group PG123 depicted in Figure 2.12 and write it to the file PG123map.txt, use the following command:

```
db2gmap -d <database_name> -g PG123 -m PG123map.txt
```
The first eight lines of the output file (PG123map.txt) generated by the db2gpmap utility would look like the following:

```
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

The upper left corner corresponds to the partition ID at element 0 of the partitioning map. The map is read by moving to the right and down, line by line.

**Tables**

In a partition database, a table will span all of the database partitions in its partition group. A table cannot span more than one partition group, but multiple tables can exist in the same partition group. Figure 2.13 shows the relationship between partition groups, table spaces, and tables.

The figure shows that the tables T2 and T3 are in the table space TS2, and the table space TS2 is in the partition group PG1234. The table spaces TS1 and TS2 both span all four database partitions, because the partition group PG1234 is created on all four database partitions.
Partitioning Keys

The element that determines how the rows in a table are allocated among the database partitions is the table’s partitioning key. The partitioning key for a table is a subset of the columns of the table and is defined when a table is created within a partitioned database.

For tables created in database partition groups containing more than one database partition, the partition key cannot be changed. If no partitioning key is explicitly specified, the first column that is not a Long Varchar or LOB will be used as the partitioning key.

Choosing a Partitioning Key

The following considerations should be taken into account when choosing a partitioning key for a table:

• Include frequently used join columns.
• Use the smallest number of columns possible.
• The column(s) should have a high proportion of different values.
• Integer columns are more efficient than character columns, which are more efficient than decimal columns.
• Long Varchar and LOB columns fields cannot be part of the partitioning key.
• The primary key or any unique index must include the partitioning key.

Specifying the Partitioning Key

If the partitioning key is not explicitly specified in the create table statement and the table is being created in a multiple-partition database partition group, the partitioning key will be defined as follows:

• If a primary key is specified, the first column of the primary key is the partitioning key.
• If the table is a typed table, the object identifier column will be the primary key.
• Otherwise, the first column whose data type is not a LOB, LONG VARCHAR, LONG VARGRAPHIC, DATALINK column, distinct type based on one of these types, or structured type column is the partitioning key.

If none of the columns satisfy the requirements described above, the table will be created without a partitioning key, and it can be created only in a table space defined on single-partition database partition groups.
For tables in table spaces defined on single-partition database partition groups, any collection of non-long (i.e., not Long Varchar, not LOB) columns can be used to define the partitioning key. If the partitioning key is not explicitly specified, no partitioning key will be created.

Partitioning Key Examples

To determine the partitioning key used for an existing table, use the following statement:

```sql
select colname, partkeyseq
from syscat.columns
where tabname='<table_name>'
order by partkeyseq
```

The following examples will show how the partitioning key for a table can be specified, and if not specified, how DB2 will choose the partitioning key. The examples will show the create table statement, followed by the partitioning key.

```sql
create table <table_name>
(c1 varchar(30), c2 int, c3 char(10), c4 int) partitioning key (c2)
partitioning key c2

create table <table_name>
(name char(30), ID int, email varchar(50)) partitioning key (ID)
partitioning key ID

create table <table_name>
(name char(30), ID int, email varchar(50))
partitioning key name

create table <table_name>
(description longvarchar(3000), ID int, email varchar(50))
partitioning key ID

create table <table_name>
(name char(30) not null, ID int not null, email varchar(50),
 primary key (ID))
partitioning key ID

create table <table_name>
(name char(30) not null, ID int not null, email varchar(50),
 primary key (ID, name))
partitioning key ID
```
For tables T1 and T2 with the following columns:

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th></th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>int</td>
<td>C1</td>
<td>char(10)</td>
</tr>
<tr>
<td>C2</td>
<td>char(10)</td>
<td>C2</td>
<td>int</td>
</tr>
<tr>
<td>C3</td>
<td>real</td>
<td>C3</td>
<td>real</td>
</tr>
</tbody>
</table>

To support the following SQL statement without needing to send data between partitions:

```
select t1.c, t1.c3 from t1, t2
where t1.c2=t2.c1
```

The partitioning key for table T1 should be c2, and the partitioning key for table T2 should be c1. To create the tables with these partitioning keys, the following statements can be used:

```
create table T1  (c1 int, c2 char(10), c3 real) partitioning key (c2)
create table T2  ( c1 char(10), c2 int, c3 real) partitioning key (c1)
```

Because the column c1 would be the default partitioning key for table T2, the partitioning key does not need to be explicitly specified. However, it is always good practice to explicitly specify the partitioning key when a table is created.

**Changing the Partitioning Key**

Once a partitioning key has been specified for a partitioned table, it cannot be changed. To change it, the table’s data would need to be exported, the table dropped and recreated, and the data reloaded. For a table in a single-partition database partition group, the partitioning key can be changed by dropping the existing partitioning key and creating a new partitioning key, as follows:

```
alter table <table_name> drop partitioning key
alter table <table_name> add partitioning key(<column,column,...>)
```

For example, for the following table in a nonpartitioned database partition group:

```
create table T2  ( c1 char(10), c2 int, c3 real) partitioning key (c1)
```

To change the partitioning key to be column c2, use the following statements:

```
apt alter table T2 drop partitioning key
alter table T2 add partitioning key(c2)
```

If an attempt is made to drop the partitioning key of a table created in a multi-partition database partition group, the following error will be returned:

```
SQL0264N Partitioning key cannot be added or dropped because table resides in a table space defined on the multi-node nodegroup "<name>".
```
Indexes and Partitioning Keys

The indexes defined on a table are partitioned based on the partitioning key of the underlying table. When an index is created, it will be created in all of the database partitions where the table was created. In each database partition, the entries in the index will have entries only for the rows of the table that are located in the same database partition.

Non-unique indexes can be created on any columns of a table, regardless of whether it is partitioned. However, any unique index or primary key created on a partitioned table must include the columns in the partitioning key.

Collocated Tables

Often in a database, there will be tables that are frequently joined by applications. If the database is partitioned, it is good practice to try to avoid sending data between the partitions to satisfy these joins. This can be achieved in DB2 by exploiting table collocation. Collocation between two joined tables occurs when the matching rows of the two tables are stored in the same database partitions, so that no data needs to be sent between partitions. Figure 2.14 shows an example where tables T1 and T2 are collocated. A join made between these two tables is satisfied without the need for rows to be sent from one database partition to another.

Two tables are collocated when they satisfy all of the following conditions:

- The partition groups of both tables must have the same partitioning map.
- The partitioning key from both tables must have the same number of columns.
• Corresponding partitioning key columns must be partition-compatible (the same or similar base data type).
• For tables in multi-partition partition groups:
  ◦ The two tables must be in the same partition group.
• For tables in single-partition partition groups:
  ◦ The tables can be in different partition groups, but these partition groups must include the same (single) database partition.

A collocated join will occur if two collocated tables are joined using all of the columns in the partitioning key.

If two tables can be joined using a collocated join, the query can be performed much more efficiently, compared with a join that sends data between database partitions.

**SUMMARY**

DB2 UDB Version 8 must read and update data rows in the database buffer pools. To be able to do this as efficiently and effectively as possible, it is important that the buffer pools be sized correctly and assigned to the correct table spaces. DB2 UDB V8 can manipulate the data in parallel, using intra-partition parallelism, inter-partition parallelism, or a combination of both.

It is important to remember that parallelism will not help all database workloads and that it should be enabled only where appropriate.