Best Practices
Tuning and Monitoring Database System Performance

Steve Rees
Senior Manager
DB2 Performance Quality Assurance

Thomas Rech
Senior Consultant
DB2 SAP Center of Excellence

Gang Shen
Technical Sales Specialist
IBM Data Servers

Roman B. Melnyk
DB2 Information Development

Last updated: 2008-10-23
Executive summary ................................................................. 4
Introduction ......................................................................................... 5
The first step: configuring for good performance ......................... 6
  Hardware configuration ................................................................. 6
  AIX configuration ........................................................................... 7
  Solaris and HP-UX configuration .................................................. 8
  Linux configuration ......................................................................... 9
  DB2 Data Partitioning Feature ....................................................... 9
  Choice of code page and collation .................................................. 10
Physical database design ................................................................. 11
Initial DB2 configuration settings .................................................... 11
DB2 autonomics and automatic parameters ................................. 12
Explicit configuration settings ........................................................ 13
Statistics collection .......................................................................... 14
Considerations for SAP and other ISV environments .................. 14
The next step: monitoring system performance ......................... 15
  A good ‘starter set’ of DB2 snapshot metrics ................................... 17
  Other important data to collect ..................................................... 21
  Cross-partition monitoring in partitioned database environments .... 22
Performance Tuning and Troubleshooting ....................................... 23
  Types of problems that you might see ............................................ 24
  Disk bottlenecks ........................................................................... 25
    Disk bottlenecks: The overall picture .......................................... 32
  CPU bottlenecks ........................................................................... 33
    System CPU bottlenecks: The overall picture ............................ 40
  Memory bottlenecks ....................................................................... 41
  ‘Lazy System’ bottlenecks ............................................................ 43
    System bottlenecks – the Overall Picture .................................... 50
Executive summary

Most DB2 systems go through something of a “performance evolution”. The system must first be configured, both from hardware and software perspectives. In many ways, this sets the stage for how the system behaves when it is in operation. Then, after the system is deployed, a diligent DBA monitors system performance, in order to detect any problems that might develop. If such problems develop, we come to the next phase – troubleshooting. Each phase depends on the previous ones, in that without proper preparation in the previous phase, we are much more likely to have difficult problems to solve in the current phase.

This paper presents DB2 system performance best practices following this same progression. We begin by touching on a number of important principles of hardware and software configuration that can help ensure good system performance. Then we discuss various monitoring techniques that help you understand system performance under both operational and troubleshooting conditions. Lastly, because performance problems can occur despite our best preparations, we talk about how to deal with them in a step-wise, methodical fashion.
Introduction

System performance issues, in almost any form, can significantly degrade the value of a system to your organization. Reduced operational capacity, service interruptions, and increased administrative overhead all contribute to higher total cost of ownership (TCO). A lack of understanding of the basic principles of system configuration, monitoring, and performance troubleshooting can result in prolonged periods of mildly-to-seriously poor performance and reduced value to the organization.

By spending some time early on to consider basic configuration guidelines and to establish sound system monitoring practices, you will be better prepared to handle many typical performance problems that might arise. The result is a data server that can perform at a higher level and may provide an improved return on investment (ROI).
The first step: configuring for good performance

Some types of DB2 deployment, such as the InfoSphere Balanced Warehouse (BW), or those within SAP systems, have configurations that are highly specified. In the BW case, hardware factors, such as the number of CPUs, the ratio of memory to CPU, the number and configuration of disks, and versions are pre-specified, based on thorough testing to determine the optimal configuration. In the SAP case, hardware configuration is not as precisely specified; however, there are a great many sample configurations available. In addition, SAP best practice provides recommended DB2 configuration settings. If you are using a DB2 deployment for a system that provides well-tested configuration guidelines, you should generally take advantage of the guidelines in place of more general rules-of-thumb.

Consider a proposed system for which you do not already have a detailed hardware configuration. An in-depth study of system configuration is beyond the scope of this paper. However, there are a number of basic guidelines that are well worth the time to understand and apply. Your goal is to identify a few key configuration decisions that get the system well on its way to good performance. This step typically occurs before the system is up and running, so you might have limited knowledge of how it will actually behave. In a way, you have to make a “best guess,” based on your knowledge of what the system will be doing. Fine tuning and troubleshooting based on actual monitoring data collected from the system, are dealt with later in this paper.

Hardware configuration

CPU capacity is one of the main independent variables in configuring a system for performance. Because all other hardware configuration typically flows from it, it is not easy to predict how much CPU capacity is required for a given workload. In business intelligence (BI) environments, 200-300 GB of active raw data per processor core is a reasonable estimate. For other environments, a sound approach is to gauge the amount of CPU required, based on one or more existing DB2 systems. For example, if the new system needs to handle 50% more users, each running SQL that is at least as complex as that on an existing system, it would be reasonable to assume that 50% more CPU capacity is required. Likewise, other factors that predict a change in CPU usage, such as different throughput requirements or changes in the use of triggers or referential integrity, should be taken into account as well.

After you have the best idea of CPU requirements (derived from available information), other aspects of hardware configuration start to fall into place. Although you must consider the required system disk capacity in gigabytes or terabytes, the most important factors regarding performance are the capacity in I/Os per second (IOPS), or in megabytes per second of data transfer. In practical terms, this is determined by the number of individual disks involved.
Why is that the case? The evolution of CPUs over the past decade has seen incredible increases in speed, whereas the evolution of disks has been more in terms of their capacity and cost. There have been improvements in disk seek time and transfer rate, but they haven’t kept pace with CPU speeds. So to achieve the aggregate performance needed with modern systems, using multiple disks is more important than ever, especially for systems that will drive a significant amount of random disk I/O. Often, the temptation is to use close to the minimum number of disks that can contain the total amount of data in the system, but this generally leads to very poor performance.

In the case of RAID storage, or for individually addressable drives, a rule-of-thumb is to configure at least ten to twenty disks per processor core. For storage servers, a similar number is recommended; however, in this case, a bit of extra caution is warranted. Allocation of space on storage servers is often done more with an eye to capacity rather than throughput. It is a very good idea to understand the physical layout of database storage, to ensure that the inadvertent overlap of logically separate storage does not occur. For example, a reasonable allocation for a 4-way system might be eight arrays of eight drives each. However, if all eight arrays share the same eight underlying physical drives, the throughput of the configuration would be drastically reduced, compared to eight arrays spread over 64 physical drives. See Database Storage and Physical Database Design for more information on storage configuration best practices.

It is good practice to set aside some dedicated (unshared) disk for the DB2 transaction logs. This is because the I/O characteristics of the logs are very different from DB2 containers, for example, and the competition between log I/O and other types of I/O can result in a logging bottleneck, especially in systems with a high degree of write activity.

In general, a RAID-1 pair of disks can provide enough logging throughput for up to 400 reasonably write-intensive DB2 transactions per second. Greater throughput rates, or high-volume logging (for example, during bulk inserts), requires greater log throughput, which can be provided by additional disks in a RAID-10 configuration, connected to the system through a write-caching disk controller. The troubleshooting section below describes how to tell if the log is a bottleneck.

Because CPUs and disks effectively operate on different time scales – nanoseconds versus microseconds – you need to decouple them to enable reasonable processing performance. This is where memory comes into play. In a database system, the main purpose of memory is to avoid I/O, and so up to a point, the more memory a system has, the better it can perform. Fortunately, memory costs have dropped significantly over the last several years, and systems with tens to hundreds of gigabytes (GB) of RAM are not uncommon. In general, four to eight gigabytes per processor core should be adequate for most applications.

**AIX configuration**

There are relatively few AIX parameters that need to be changed to achieve good performance. For the purpose of these recommendations, we assume an AIX level of 5.3 or later. Again, if there are specific settings already in place for your system (for example,
a BW or SAP configuration), those should take precedence over the following general guidelines.

- The VMO parameter `LRU_FILE_REPAGE` should be set to 0. This parameter controls whether AIX victimizes computational pages or file system cache pages. In addition, `minperm` should be set to 3. These are both default values in AIX 6.1.

- The AIO parameter `maxservers` can be initially left at the default value of ten per CPU. After the system is active, `maxservers` is tuned as follows:

  1. Collect the output of the `ps -elfk | grep aio` command and determine if all asynchronous I/O (AIO) kernel processes (aioservers) are consuming the same amount of CPU time.
  2. If they are, `maxservers` might be set too low. Increase `maxservers` by 10%, and repeat step 1.
  3. If some aioservers are using less CPU time than others, the system has at least as many of them as it needs. If more than 10% of aioservers are using less CPU, reduce `maxservers` by 10% and repeat step 1.

- The AIO parameter `maxreqs` should be set to `MAX( NUM_IOCLEANERS x 256, 4096 )`. This parameter controls the maximum number of outstanding AIO requests.

- The hdisk parameter `queue_depth` should be based on the number of physical disks in the array. For example, for IBM disks, the default value for `queue_depth` is 3, and the recommended value would be `3 x number-of-devices`. This parameter controls the number of queuable disk requests.

- The disk adapter parameter `num_cmd_elems` should be set to the sum of `queue_depth` for all devices connected to the adapter. This parameter controls the number of requests that can be queued to the adapter.

**Solaris and HP-UX configuration**

For DB2 running on Solaris or HP-UX, the `db2osconf` utility is available to check and recommend kernel parameters based on the system size. The `db2osconf` utility allows you to specify the kernel parameters based on memory and CPU, or with a general scaling factor that compares the current system configuration to an expected future configuration. A good approach is to use a scaling factor of 2 or higher if running large systems, such as SAP applications. In general, `db2osconf` gives you a good initial starting point to configure Solaris and HP-UX, but it does not deliver the optimal value, because it cannot consider current and future workloads.
**Linux configuration**

When a Linux system is used as a DB2 server, some of the Linux kernel parameters might have to be changed. Because Linux distributions change, and because this environment is highly flexible, we only discuss some of the most important settings that need to be validated on the basis of the Linux implementation.

SHMMAX (maximum size of a shared memory segment) on a 64-bit system must be set to a minimum of 1 GB – 1073741824 bytes – whereas the parameter SHMALL should be set to 90% of the available memory on the database server. SHMALL is 8 GB by default. Other important Linux kernel configuration parameters and their recommended values for DB2 are:

- `kernel.sem` (specifying four kernel semaphore settings – SEMMSL, SEMMNS, SEMOPM, and SEMMNI): 250 256000 32 1024
- `kernel.msgmni` (number of message queue identifiers): 1024
- `kernel.msgmax` (maximum size of a message, in bytes): 65536
- `kernel.msgmnb` (default size of a message queue, in bytes): 65536

**DB2 Data Partitioning Feature**

The decision to use the DB2 Data Partitioning Feature (DPF) is not generally made based purely on data volume, but more on the basis of the workload. As a general guideline, most DPF deployments are in the area of data warehousing and business intelligence. The DPF is highly recommended for large complex query environments, because its shared-nothing architecture allows for outstanding scalability. For smaller data marts (up to about 300 GB), which are unlikely to grow rapidly, a DB2 Enterprise Server Edition (ESE) configuration is often a good choice. However, large or fast-growing BI environments benefit greatly from the DPF.

Although a thorough treatment of partitioned database system design is beyond the scope of this paper, a basic description of CPU-to-partition allocation is fairly straightforward.

A typical partitioned database system usually has one processor core per data partition. For example, a system with N processor cores would likely have the catalog on partition 0, and have N additional data partitions. If the catalog partition will be heavily used (for example, to hold single partition dimension tables), it might be allocated a processor core as well. If the system will support very many concurrent active users, two cores per partition might be required.

In terms of a general guide, you should plan on about 250 GB of active raw data per partition.
The InfoSphere Balanced Warehouse documentation contains in-depth information regarding partitioned database configuration best practices. This documentation contains useful information for non-Balanced Warehouse deployments as well.

**Choice of code page and collation**

As well as affecting database behavior, choice of code page or code set and collating sequence can have a strong impact on performance. The use of Unicode has become very widespread because it allows you to represent a greater variety of character strings in your database than has been the case with traditional single-byte code pages. Unicode is the default for new databases in DB2 Version 9.5. However, because Unicode code sets use multiple bytes to represent some individual characters, there can be increased disk and memory requirements. For example, the UTF-8 code set, which is one of the most common Unicode code sets, uses from one to four bytes per character. An average string expansion factor due to migration from a single-byte code set to UTF-8 is very difficult to estimate because it depends on how frequently multi-byte characters are used. For typical North American content, there is usually no expansion. For most western European languages, the use of accented characters typically introduces an expansion of around 10%.

On top of this, the use of Unicode can cause extra CPU consumption relative to single-byte code pages. First, if expansion occurs, the longer strings require more work to manipulate. Second, and more significantly, the algorithms used by the more sophisticated Unicode collating sequences, such as UCA500R1_NO, can be much more expensive than the typical SYSTEM collation used with single-byte code pages. This increased expense is due to the complexity of sorting Unicode strings in a culturally-correct way. Operations that are impacted include sorting, string comparisons, LIKE processing, and index creation.

If Unicode is required to properly represent your data, choose the collating sequence with care.

- If the database will contain data in multiple languages, and correct sort order of that data is of paramount importance, use one of the culturally correct collations (for example, UCA500R1_xxx). Depending on the data and the application, this could have a performance overhead of 1.5 to 3 times more, relative to the IDENTITY sequence.

- There are both normalized and non-normalized varieties of culturally-correct collation. Normalized collations (for example, UCA500R1_NO) have additional checks to handle malformed characters, whereas non-normalized collations (for example, UCA500r1_NX) do not. Unless the handling of malformed characters is an issue, we recommend using the non-normalized version, because there is a performance benefit in avoiding the normalization code. That said, even non-normalized culturally correct collations are very expensive.

- If a database is being moved from a single-byte environment to a Unicode environment, but does not have rigorous requirements about hosting a variety of
languages (most deployments will be in this category), ‘language aware’
collation might be appropriate. Language aware collations (for example,
SYSTEM_819_BE) take advantage of the fact that many Unicode databases
contain data in only one language. They use the same lookup table-based
collation algorithm as single-byte collations such as SYSTEM_819, and so are
very efficient. As a general rule, if the collation behavior in the original single-
byte database was acceptable, then as long as the language content does not
change significantly following the move to Unicode, culturally aware collation
should be considered. This can provide very large performance benefits relative
to culturally correct collation.

Physical database design
The details of physical database design are well covered in Physical Database Design, but
for our purposes, we address a couple of the top-level best practices here.

- In general, file-based database managed storage (DMS) regular table spaces give
  better performance than system managed storage (SMS) regular table spaces.
  SMS is often used for temporary table spaces, especially when the temporary
  tables are very small; however, the performance advantage of SMS in this case is
  shrinking over time.

- In the past, DMS raw device table spaces had a fairly substantial performance
  advantage over DMS file table spaces; however, with the introduction of direct
  I/O (now defaulted through the NO FILE SYSTEM CACHING clause in the
  CREATE TABLESPACE and the ALTER TABLESPACE statements), DMS file
  table spaces provide virtually the same performance as DMS raw device table
  spaces.

Initial DB2 configuration settings
The DB2 configuration advisor, also known as the AUTOCONFIGURE command, takes
basic system guidelines that you provide, and determines a good starting set of DB2
configuration values. The AUTOCONFIGURE command can provide real
improvements over the default configuration settings, and is recommended as a way to
obtain initial configuration values. Some additional fine-tuning of the recommendations
generated by the AUTOCONFIGURE command is often required, based on the
characteristics of the system.

Here are some suggestions for using the AUTOCONFIGURE command:

- Even though, starting in DB2 Version 9.1, the AUTOCONFIGURE command is
  run automatically at database creation time, it is still a good idea to run the
  AUTOCONFIGURE command explicitly. This is because you then have the
  ability to specify keyword/value pairs that help customize the results for your
  system.
- Run (or re-run) the `AUTOCONFIGURE` command after the database is populated. This provides the tool with more information about the nature of the database. Ideally, ‘populated’ means with the amount of active data that you use (which affects buffer pool size calculations, for example). Too much or too little data makes these calculations less accurate.

- Try different values for important `AUTOCONFIGURE` command keywords, such as `mem_percent`, `tpm`, and `num_stmts` to get an idea of which, and to what degree, configuration values are affected by these changes.

- If you are experimenting with different keywords and values, use the `apply none` option. This gives you a chance to compare the recommendations with the current settings.

- Specify values for all keywords, because the defaults might not suit your system. For example, `mem_percent` defaults to 25%, which is too low for a dedicated DB2 server; 85% is the recommended value in this case.

**DB2 autonomies and automatic parameters**

Recent releases of DB2 database products have significantly increased the number of parameters that are either automatically set at instance or database start-up time, or that are dynamically tuned during operation. For most systems, automatic settings provide better performance than all but the very carefully hand-tuned systems. This is particularly due to the DB2 self-tuning memory manager (STMM), which dynamically tunes total database memory allocation as well as four of the main memory consumers in a DB2 system: the buffer pools, the lock list, the package cache, and the sort heap.

Because these parameters apply on a partition-by-partition basis, using the STMM in a partitioned database environment should be done with some caution. On partitioned database systems, the STMM continuously measures memory requirements on a single partition (automatically chosen by the DB2 system, but that choice can be overridden), and ‘pushes out’ heap size updates to all partitions on which the STMM is enabled. Because the same values are used on all partitions, the STMM works best in partitioned database environments where the amounts of data, the memory requirements, and the general levels of activity are very uniform across partitions. If a small number of partitions have skewed data volumes or different memory requirements, the STMM should be disabled on those partitions, and allowed to tune the more uniform ones. For example, the STMM should generally be disabled on the catalog partition.

For partitioned database environments with skewed data distribution, where continuous cross-cluster memory tuning is not advised, the STMM can be used selectively and temporarily during a ‘tuning phase’ to help determine good manual heap settings:

- Enable the STMM on one ‘typical’ partition. Other partitions continue to have the STMM disabled.
• After memory settings have stabilized, disable the STMM and manually ‘harden’ the affected parameters at their tuned values.

• Deploy the tuned values on other database partitions with similar data volumes and memory requirements (for example, partitions in the same partition group).

• Repeat the process if there are multiple disjointed sets of database partitions containing similar volumes and types of data and performing similar roles in the system.

The configuration advisor generally chooses to enable autonomic settings where applicable. This includes automatic statistics updates from the RUNSTATS command (very useful), but excludes automatic reorganization and automatic backup. These can be very useful as well, but need to be configured according to your environment and schedule for best results. Automatic statistics profiling should remain disabled by default. It has quite high overhead and is intended to be used temporarily under controlled conditions and with complex statements.

Explicit configuration settings

Some parameters do not have automatic settings, and are not set by the configuration advisor. These need to be dealt with explicitly. We only consider parameters that have performance implications.

• logpath or newlogpath determine the location of the transaction log. Even the configuration advisor cannot decide for you where the logs should go. As mentioned above, the most important point is that they should not share disk devices with other DB2 objects, such as table spaces, or be allowed to remain in the default location, which is under the database path. Ideally, transaction logs should be placed on dedicated storage with sufficient throughput capacity to ensure that a bottleneck won’t be created.

• logbufsz determines the size of the transaction logger internal buffer, in 4KB pages. The default value of only eight pages is far too small for good performance in a production environment. The configuration advisor always increases it, but possibly not enough, depending on the input parameters. A value of 256-1000 pages is a good general range, and represents only a very small total amount of memory in the overall scheme of a database server.

• mincommit controls ‘group commit’, which causes a DB2 system to try to batch together $N$ committing transactions. With the current transaction logger design, this is rarely the desired behavior. Leave mincommit at the default value of 1.

• buffpage determines the number of pages allocated to each buffer pool that is defined with a size of -1. The best practice is to ignore buffpage, and either explicitly set the size of buffer pools that have an entry in SYSCAT.BUFFERPOOLS, or let the STMM tune buffer pool sizes automatically.
• **diagpath** determines the location of various useful DB2 diagnostic files. It generally has little impact on performance, except possibly in a partitioned database environment. The default location of **diagpath** on all partitions is typically on a shared, NFS-mounted path. The best practice is to override **diagpath** to a local, non-NFS directory for each partition. This prevents all partitions from trying to update the same file with diagnostic messages. Instead, these are kept local to each partition, and contention is greatly reduced.

• **DB2_PARALLEL_IO** is not a configuration parameter, but a DB2 registry variable. It is very common for DB2 systems to use storage consisting of arrays of disks, which are presented to the operating system as a single device, or to use file systems that span multiple devices. The consequence is that by default, a DB2 database system makes only one prefetch request at a time to a table space container. This is done with the understanding that multiple requests to a single device are serialized anyway. But if a container resides on an array of disks, there is an opportunity to dispatch multiple prefetch requests to it simultaneously, without serialization. This is where **DB2_PARALLEL_IO** comes in. It tells the DB2 system that prefetch requests can be issued to a single container in parallel. The simplest setting is **DB2_PARALLEL_IO=\*** (meaning that all containers reside on multiple – assumed in this case to be seven – disks), but other settings also control the degree of parallelism and which table spaces are affected. For example, if you know that your containers reside on a RAID-5 array of four disks, you might set **DB2_PARALLEL_IO** to “*:3”. Whether or not particular values benefit performance also depends on the extent size, the RAID segment size, and how many containers use the same set of disks. See Database Storage for more information on storage configuration and **DB2_PARALLEL_IO**.

**Statistics collection**

It’s no exaggeration to say that having the right statistics is often critical to achieving the best SQL performance, especially in complex query environments. For a complete discussion of this topic, see Writing and Tuning Queries for Optimal Performance.

**Considerations for SAP and other ISV environments**

If you are running a DB2 database server for an ISV application such as SAP, some best practice guidelines that take into account the specific application might be available. The most straightforward mechanism is the DB2 registry variable **DB2_WORKLOAD**, which can be set to the value **SAP** to enable aggregated registry variables that are optimized for SAP workloads.

Other recommendations and best practices might apply, such as the choice of a code page or code set and collating sequence, because they must be set to a predetermined value. Refer to the application vendor’s documentation for details.

For many ISV applications, such as SAP Business One, the **AUTOCONFIGURE** command can be successfully used to define the initial configuration. However, it should not be used in SAP NetWeaver installations, because an initial set of DB2 configuration
parameters is applied during SAP installation. In addition, SAP has a powerful alternative best practices approach with SAP Notes that describes the preferred DB2 parameter setting; for example, SAP Note 1086130 - DB6: DB2 9.5 Standard Parameter Settings.

Pay special attention to SAP applications when using the DB2 DPF feature. SAP uses DPF mainly in their SAP NetWeaver Business Intelligence (Business Warehouse) product. The recommended layout has the DB2 system catalog, the dimension and master tables, plus the SAP base tables on Partition 0. This leads to a different workload on this partition compared to other DB2 DPF installations. Because the SAP application server runs on this partition, up to eight processors might be assigned to just this partition. As the SAP BW workload becomes more highly parallelized, with many short queries running concurrently, the number of partitions for SAP BI is typically smaller than for other applications. In other words, more than one CPU per data partition is required.

To find more details about the initial setup for DB2 and SAP, please check the SAP Service Marketplace (service.sap.com) or the SAP Developer Network.

The next step: monitoring system performance

After devising an initial system configuration, it is important to put a monitoring strategy in place to keep track of many important system performance metrics over time. This not only gives you critical data to refine that initial configuration to be more tailored to your requirements, but it also prepares you to address new problems that might appear on their own or following software upgrades, increases in data or user volumes, or new application deployments.

There are hundreds of metrics to choose from, but collecting all of them can be counter-productive due to the sheer volume of data produced. You want metrics that are:

- Easy to collect – You don’t want to have to use complex or expensive tools for everyday monitoring, and you don’t want the act of monitoring to significantly burden the system.

- Easy to understand – You don’t want to have to look up the meaning of the metric each time you see it.

- Relevant to your system – Not all metrics provide meaningful information in all environments.

- Sensitive, but not too sensitive – A change in the metric should indicate a real change in the system; the metric should not fluctuate on its own.

The DB2 database product has many monitoring elements, and we discuss those that meet these requirements.

We draw a distinction between operational monitoring (which is something done on a day-to-day basis) and exception monitoring (collecting extra data to help diagnose a
problem. The primary difference is that operational monitoring needs to be very light weight (not consuming much of the system it is measuring) and generic (keeping a broad ‘eye’ out for potential problems that could appear anywhere in the system). In this section, we focus primarily on operational monitoring.

A DB2 database system provides some excellent sources of monitoring data. The primary ones are snapshot monitors and, in DB2 Version 9.5 and later, workload management (WLM) aggregate functions. Both of these focus on summary data, where tools like counters, timers, and histograms maintain running totals of activity in the system. By sampling these monitor elements over time, you can derive the average activity that has taken place between the start and end times, which can be very informative.

Using a photographic analogy, snapshots and aggregate statistics give us a single picture of system activity. In some cases, it is instantaneous, like flash photography, but more often it is a ‘time exposure’, showing what’s happened over a considerable period of time. DB2 also provides ‘motion picture’ monitoring, which records the stream of execution of a series of individual activities. This is achieved with trace-like mechanisms, such as event monitors (especially statement event monitors) and WLM activity monitors, which are closely related. These tools provide a much more comprehensive, detailed recording of system activity than the summary you get from snapshots. However, traces produce a huge amount of data and impose a much greater overhead on the system. Consequently, they are more suitable for exception monitoring than for operational monitoring.

There is no reason to limit yourself to just metrics that DB2 provides. In fact, non-DB2 data is more than just a nice-to-have. Contextual information is key for performance problem determination. The users, the application, the operating system, the storage subsystem, and the network – all of these can provide valuable information about system performance. Including metrics from outside of the DB2 database software is an important part of producing a complete overall picture of system performance.

Because you plan regular collection of operational metrics throughout the life of the system, it is important to have a way to manage all that data. For many of the possible uses you have for your data, such as long-term trending of performance, you want to be able to do comparisons between arbitrary collections of data that are potentially many months apart. The DB2 product itself facilitates this kind of data management very well. Analysis and comparison of monitoring data becomes very straightforward, and you already have a robust infrastructure in place for long-term data storage and organization.

The trend in recent releases of the DB2 database product has been to make more and more monitoring data available through SQL interfaces. This makes management of monitoring data with DB2 very straightforward, because you can easily redirect the data from the administration views, for example, right back into DB2 tables. For deeper dives, event and activity monitor data can also be written to DB2 tables, providing similar benefits. With the vast majority of our monitoring data so easy to store in DB2, a small investment to store system metrics (such as CPU utilization from `vmstat`) in DB2 is manageable as well.
A good ‘starter set’ of DB2 snapshot metrics

These come from the database snapshot (GET SNAPSHOT FOR DATABASE command). As mentioned earlier, the best practice is to access these elements through the appropriate administrative view (SYSIBMADM.SNAPDB), which makes analysis and long-term management simpler. The instance-level monitor switches must be turned on if snapshot table functions and administrative views are to access the data. For example, `DFT_MON_BUFPOOL` enables collection of buffer pool monitoring data. Fortunately, these switches are dynamic and do not require an instance restart to turn them on or off. In the examples below, we highlight the administrative view columns that are used to determine each metric.

A note about DPF: if you are running in a multi-partition environment, you should include `DBPARTITIONNUM` in your monitoring `SELECT` statements, to distinguish the rows that you get back for each partition.

1. The number of transactions, SELECT statements, and INSERT, UPDATE, or DELETE statements executed:

| COMMIT_SQL_STMTS, SELECT_SQL_STMTS and UID_SQL_STMTS |

These provide an excellent base level measurement of system activity.

2. Buffer pool hit ratios, measured separately for data, index, and temporary data:

| 100 * (POOL_DATA_L_READS - POOL_DATA_P_READS) / POOL_DATA_L_READS |
| 100 * (POOL_INDEX_L_READS - POOL_INDEX_P_READS) / POOL_INDEX_L_READS |
| 100 * (POOL_TEMP_DATA_L_READS - POOL_TEMP_DATA_P_READS) / POOL_TEMP_DATA_L_READS |
| 100 * (POOL_TEMP_INDEX_L_READS - POOL_TEMP_INDEX_P_READS) / POOL_TEMP_INDEX_L_READS |

Buffer pool hit ratios are one of the most fundamental metrics we have, and give an important overall measure of how effectively the system is exploiting memory to avoid disk I/O. Hit ratios of 80-85% or better for data and 90-95% or better for indexes are generally considered good for an OLTP environment, and of course these ratios can be calculated for individual buffer pools using data from the buffer pool snapshot.

Although these metrics are generally useful, for systems such as data warehouses that frequently perform large table scans, data hit ratios are often irretrievably low, because data is read into the buffer pool and then not used again before being evicted to make room for other data.

3. Buffer pool physical reads and writes per transaction:
These metrics are closely related to buffer pool hit ratios, but have a slightly different purpose. Although we can talk about target values for hit ratios, there are no possible targets for reads and writes per transaction. Why do we bother with these calculations? Because disk I/O is such a major factor in database performance, it is useful to have multiple ways of looking at it. As well, these calculations include writes, whereas hit ratios only deal with reads. Lastly, in isolation, it’s difficult to know, for example, whether a 94% index hit ratio is worth trying to improve. If we do only 100 logical index reads per hour, and 94 of them are in the buffer pool, working to keep those last 6 from turning into physical reads is not a good use of time. However, if our 94% index hit ratio were accompanied by a statistic that each transaction did twenty physical reads (which could be further broken down by data and index, regular and temporary), the buffer pool hit ratios might well deserve some investigation.

The metrics aren’t just ‘physical reads and writes’, but are normalized per transaction. We follow this trend through many of the metrics. The purpose is to decouple metrics from the length of time data was collected, and from whether the system was very busy or less busy at that time. In general, this helps ensure that we get similar values for our metrics, regardless of whether we are very precise about how and when monitoring data is collected. Some amount of consistency in the timing and duration of data collection is a good thing; however, normalization reduces it from being ‘critical’ to being ‘a good idea’.

4. The ratio of database rows read to rows selected:

\[
\frac{\text{ROWS\_READ}}{\text{ROWS\_SELECTED}}
\]

This calculation gives us an indication of the average number of rows that are read from database tables in order to find the rows that qualify. Low numbers are an indication of efficiency in locating data, and generally show that indexes are being used effectively. For example, this number can be very high in the case where the system does many table scans, and millions of rows need to be inspected to determine if they qualify for the result set. On the other hand, this statistic can be very low in the case of access to a table through a fully-qualified unique index. Index-only access plans (where no rows need to be read from the table) do not cause ROWS\_READ to increase.
In an OLTP environment, this metric is generally no higher than 2 or 3, indicating that most access is through indexes instead of table scans. This metric is a simple way to monitor plan stability over time – an unexpected increase is often an indication that an index is no longer being used and should be investigated.

5. The amount of time spent sorting per transaction:

\[
\text{TOTAL\_SORT\_TIME} \div \text{COMMIT\_SQL\_STMTS}
\]

This is an efficient way to handle sort statistics, because any extra overhead due to spilled sorts automatically gets included here. That said, you might also want to collect TOTAL\_SORTS and SORT\_OVERFLOWS for ease of analysis, especially if your system has a history of sorting issues.

6. The amount of lock wait time accumulated per thousand transactions:

\[
1000 \times \text{LOCK\_WAIT\_TIME} \div \text{COMMIT\_SQL\_STMTS}
\]

Excessive lock wait time often translates into poor response time, so it is important to monitor. We normalize to one thousand transactions because lock wait time on a single transaction is typically quite low. Scaling up to one thousand transactions simply gives us measurements that are easier to handle.

7. The number of deadlocks and lock timeouts per thousand transactions:

\[
1000 \times (\text{DEADLOCKS} + \text{LOCK\_TIMEOUTS}) \div \text{COMMIT\_SQL\_STMTS}
\]

Although deadlocks are comparatively rare in most production systems, lock timeouts can be more common. The application usually has to handle them in a similar way: re-executing the transaction from the beginning. Monitoring the rate at which this happens helps avoid the case where many deadlocks or lock timeouts drive significant extra load on the system without the DBA being aware.

8. The number of dirty steal triggers per thousand transactions:

\[
1000 \times \text{POOL\_DRTY\_PG\_STEAL\_CLNS} \div \text{COMMIT\_SQL\_STMTS}
\]

A ‘dirty steal’ is the least preferred way to trigger buffer pool cleaning. Essentially, the processing of an SQL statement that is in need of a new buffer pool page is interrupted while updates on the victim page are written to disk. If dirty steals are allowed to happen frequently, they can have a significant impact on throughput and response time.

9. The number of package cache inserts per thousand transactions:
Package cache insertions are part of normal execution of the system; however, in large numbers, they can represent a significant consumer of CPU time. In many well-designed systems, after the system is running at steady-state, very few package cache inserts occur, because the system is using or re-using static SQL or previously prepared dynamic SQL statements. In systems with a high traffic of ad hoc dynamic SQL statements, SQL compilation and package cache inserts are unavoidable. However, this metric is intended to watch for a third type of situation, one in which applications unintentionally cause package cache churn by not reusing prepared statements, or by not using parameter markers in their frequently executed SQL.

10. The amount of log activity per transaction:

\[
\frac{1000 \times \text{PKG_CACHE_INSERTS}}{\text{COMMIT_SQL_STMTS}}
\]

The transaction log has significant potential to be a system bottleneck, whether due to high levels of activity, or to improper configuration, or other causes. By monitoring log activity – both in number of writes and in write time – we can detect problems both from the DB2 side (meaning an increase in number of log requests driven by the application) and from the system side (often due to a decrease in log subsystem performance caused by hardware or configuration problems).

11. In partitioned database environments, the number of fast communication manager (FCM) buffers sent and received between partitions:

\[
\frac{\text{SYSIBMADM.SNAPFCM_PART.TOTAL_BUFFERS_SENT} \times \text{SYSIBMADM.SNAPFCM_PART.TOTAL_BUFFERS_RCVD}}{\text{COMMIT_SQL_STMTS}}
\]

These give the rate of flow of data between different partitions in the cluster, and in particular, whether the flow is balanced. Significant differences in the numbers of buffers received from different partitions might indicate a skew in the amount of data that has been hashed to each partition.

Querying SYSIBMADM.SNAPFCM_PART gives information about flow between all partition pairs in the system – not just sends and receives involving the current database partition, which is what `GET SNAPSHOT FOR FCM` does. To return only data involving the current database partition, add a predicate on `DBPARTITIONNUM`, as follows:
Other important data to collect

Although DB2 monitoring elements provide much key operational data, as mentioned above, it is important to augment this with other types of data:

- **DB2 configuration information**
  Taking regular copies of database and database manager configuration, DB2 registry variables, and the schema definition helps provide a history of any changes that have been made, and can help to explain changes that arise in monitoring data.

- **Overall system load**
  If CPU or I/O utilization is allowed to approach saturation, this can create a system bottleneck that might be difficult to detect using just DB2 snapshots. As a result, the best practice is to regularly monitor system load with `vmstat` and `iostat` (and possibly `netstat` for network issues) on UNIX-based systems, and `perfmon` on Windows. You can also use the administrative views, such as `ENV_SYS_RESOURCES`, to retrieve operating system, CPU, memory, and other information related to the system. Typically you look for changes in what is normal for your system, rather than for specific one-size-fits-all values.

- **Throughput and response time measured at the business logic level**
  An application view of performance, measured above DB2, at the business logic level, has the advantage of being most relevant to the end user, plus it typically includes everything that could create a bottleneck, such as presentation logic, application servers, web servers, multiple network layers, and so on. This data can be vital to the process of setting or verifying a service level agreement (SLA).

The above metrics represent a good core set of data to collect on an ongoing basis. The snapshot monitoring elements and system load data are compact enough that even if they are collected every five to fifteen minutes, the total data volume over time is irrelevant in most systems. Likewise, the overhead of collecting this data is typically in the one to three percent range of additional CPU consumption, which is a small price to pay for a continuous history of important system metrics. Configuration information typically changes relatively rarely, so collecting this once a day is usually frequent enough to be useful without creating an excessive amount of data.

Depending on your environment and the circumstances of your system, you might find it useful to collect additional data beyond these core metrics:
• Per-buffer pool data, which gives you the opportunity to break down hit ratios and physical I/O information for each buffer pool (obtained from the SYSIBMADM.SNAPBP administrative view)

• Dynamic SQL data, which gives you information about each such statement executed in the system, including a breakdown of buffer pool activity, elapsed time, and CPU consumption (obtained from the SYSIBMADM.SNAPDYN_SQL administrative view)

• Application status data (obtained from the SYSIBMADM.SNAPAPPL and SYSIBMADM.SNAPAPPL_INFO administrative views), which gives you data similar to what you get from the core database administrative view used above (SYSIBMADM.SNAPDB), but broken down by application. This provides very useful additional information when drill-down is required, helping us to understand which application might be causing a performance problem.

If you are going to drill down and pick up extra monitoring data, it is simplest to just collect all elements provided by the administrative views in which you are interested, rather than try to anticipate in advance which fields might be needed.

**Cross-partition monitoring in partitioned database environments**

Almost all of the individual monitoring element values mentioned above are reported on a per-partition basis. The same applies for much of the non-DB2 performance data you can collect, such as `vmstat` and `iostat`, which report statistics for a single OS instance (albeit possibly spanning multiple logical DB2 partitions). The monitor granularity is helpful in the sense that it matches that of the DB2 configuration parameters that you use to control the system. However, in partitioned database environments, you also have to be able to monitor relative activity between partitions, for example, to detect imbalances in per-partition data volumes.

Fortunately, the DB2 monitoring data you collect from the administrative views contains a DBPARTITIONNUM column value, where appropriate, which helps you query monitoring data by partition number, as well as compare across partitions.

In general, you expect most monitoring statistics to be fairly uniform across all partitions in the same DB2 partition group. Significant differences might indicate data skew. Sample cross-partition comparisons to track include:

• Logical and physical buffer pool reads for data, indexes, and temporary tables

• Rows read, at the partition level and for large tables

• Sort time and sort overflows

• FCM buffer sends and receives
• CPU and I/O utilization

If, through these metrics, any of the data partitions appear to be significantly more active (for example, 10-20% busier) than the least active data partitions in the same partition group (particularly if any of the busier partitions are CPU or I/O saturated) then it is likely that data repartitioning is required. Finding the number of rows from each large partitioned table that hashed to each partition will confirm whether significant skew exists:

```
SELECT COUNT(*), DBPARTITIONNUM(P_PARTKEY)
FROM TPCD.PART
GROUP BY DBPARTITIONNUM(P_PARTKEY)
```

Physical Database Design discusses how to avoid data skew and minimize expensive non-collocated joins by using the DB2 Design Advisor to choose the right partition keys for your system.

**Performance Tuning and Troubleshooting**

Even the most carefully configured system eventually finds itself in need of some performance tuning, and this is where the operational monitoring data that we collected comes in very handy.

It is important that we maintain a methodical approach to tuning and troubleshooting. When a problem occurs, it can be very tempting to apply changes almost at random, in the hope of fixing the problem. However, when you do this, the probability of actually addressing the root cause is relatively low, and you can even make the problem worse. Here are a few basic rules for performance tuning:

1. **Be prepared.** Try to understand how the system performs when all is well. Collect operational monitoring data to track changes in behavior over time.

2. **Understand the whole picture.** Do not limit yourself to looking only into the DB2 database – collect and analyze data that is coming from the operating system, storage, the application, and also from users. Understanding the nature of the system helps you to interpret monitoring data.

3. **Only tune things that can explain the symptoms you are seeing.** Don’t change the tire if the engine won’t start. Don’t try to fix a disk bottleneck by tuning to reduce CPU consumption.

4. **Change one thing at a time.** Observe the effects before changing anything else.
**Types of problems that you might see**

Performance problems tend to fall into two broad categories: those that affect the entire system, and those that only affect a part of it, such as a particular application or SQL statement. During the course of investigation, a problem of one type might turn into the other, and vice versa. For example, the root cause of an overall system slow-down might be a single statement, or a system-wide problem might first be seen only in a particular area. We start with system-wide problems.

Our overall approach to finding the cause of a slowdown is to start at a high level and then gradually refine our diagnosis. This “decision tree” strategy helps us rule out, as early as possible, causes that don’t explain the symptoms we see, and is applicable to both system-wide and more localized problems. This saves effort that would otherwise be spent making changes that have little or no impact.

Before starting an investigation within a DB2 database, it is often helpful to consider some preliminary questions, such as the following:

- If there seems to be a performance slowdown, what is it in relation to? What is our ‘baseline’?

- Is degradation seen on one system over time? Or is it a degradation as compared to a different system, or even a different application? This question might reveal a variety of possible root causes for the slowdown. Did data volume increase? Are all hardware upgrades running properly?

- When does the slowdown occur? Slowdowns might show up periodically – before, during, or after another task is run. Even if the task is not directly related to the database, it might influence performance by consuming network or CPU resources.

- Has something changed in the context of the slowdown? Sometimes, new hardware has been added, or the application has been changed, mass data was uploaded or more users are accessing the system.

These questions are usually an important part of a consolidated analysis approach, where database specialists work together with application and infrastructure experts. The DB2 server is almost always just one part of a complex environment of hardware, other middleware, and applications, and so skills from multiple domains might be required to solve the problem.

There are four common types of bottlenecks, each of which is discussed in detail below:

1. Disk
2. CPU
3. Memory
Disk bottlenecks

The basic symptoms of a disk bottleneck include:

- High I/O wait time, as reported in `vmstat` or `iostat`. This is an indication of the fraction of time that the system is waiting for disk I/O requests to complete. Up to 20% or 25% is not uncommon, but values above 30% indicate a bottleneck. High I/O wait time is a particularly good indicator of a bottleneck if idle CPU time is very low.

- Disks showing up as more than 80% busy in `iostat` or `perfmon`.

- Low-mid CPU utilization (25-50%), as seen in `vmstat`.

We might need to add more disks eventually, but for now we start by checking if we can tune the DB2 system to remove the bottleneck.

If a disk bottleneck exists, the system administrator can help map the name of a busy device to the affected file system path or paths. From there, you can determine how DB2 uses the affected path or paths:

- As a table space container? This is determined by querying TBSP_NAME, TBSP_ID and CONTAINER_NAME from SYSIBMADM.SNAPCONTAINER, and looking for the bottlenecked path in CONTAINER_NAME.

- As the transaction log path? This is determined by examining the database configuration and looking for the bottlenecked path in ‘Path to log files’.

- As the diagnostic path? This is determined by examining DIAG_PATH in the database manager configuration, and looking for the bottlenecked path in DIAG_PATH.

We consider these cases separately.

To determine what is causing the container or containers to be a bottleneck, we need to determine which tables are stored in that table space, and which ones are most active.

1. Which tables are in this table space? Query SYSCAT.TABLES, matching TBSPACEID with SNAPCONTAINER.TBSP_ID from above.

2. Which tables are most active? Query SYSIBMADM.SNAPTAB, selecting ROWS_READ and ROWS_WRITTEN for the table or tables from our hot container. Look for levels of activity that are much higher than other tables. Note that this requires the instance-level table monitor switch DFT_MON_TABLE to be ON.
Drilling down further, we need to find out what is causing the high level of activity on this table. Are dynamic SQL statements causing high activity? Query the dynamic SQL snapshot through SYSIBMADM.SNAPDYN_SQL.TBSP_ID, using the LIKE predicate on STMT_TEXT to identify statements that touch the table(s) in which we are interested:

```
SELECT ... FROM SYSIBMADM.SNAPDYN_SQL
WHERE TRANSLATE(CAST(SUBSTR(STMT_TEXT, 1, 32672) AS VARCHAR(32672))) LIKE '%<tbname>%' 
ORDER BY ...
```

Columns returned could include rows read and written, buffer pool activity, execution time, CPU time, and so on. We can use the ORDER BY clause on columns like ROWS_READ, ROWS_WRITTEN, and NUM_EXECUTIONS to concentrate on those statements that are having the greatest impact on the table. We are assuming here that the table name falls in the first 32672 characters of the SQL statement. Not a perfect assumption, but true in most cases, and required by the LIKE predicate.

Are static SQL statements causing high activity? Here we need to use the system catalog and `db2pd` to find out which statements are most active. Query SYSCAT.STATEMENTS, selecting PKGSHEMA, PKGNAME and SECTNO for the statements that reference the table in which we are interested:

```
SELECT PKGSHEMA, PKGNAME, SECTNO, SUBSTR(TEXT, 1, 80)
FROM SYSCAT.STATEMENTS
WHERE TRANSLATE(CAST(SUBSTR(TEXT, 1, 32672) AS VARCHAR(32672)))
LIKE '%<tbname>%' 
```

After we have the package names and section numbers of the static SQL statements that touch the table of interest, we use `db2pd --static` to find out which, if any, of these are highly active. The `db2pd --static` output has a row for each static SQL statement that the system has executed since `db2start` time. The NumRef counter indicates how many times that statement has been run, and the RefCount counter indicates how many DB2 agents are currently running that statement. Monitor the output of several invocations of `db2pd --static`. Rapidly climbing values of NumRef, or frequent values of RefCount over 2 or 3, tend to indicate a highly active statement.

If we get to this point and identify one or more SQL statements that are causing our I/O bottleneck, we must next determine whether the statement or statements can be
optimized to reduce I/O. Is the statement driving an unwanted table scan? This can be verified by examining the access plan with `db2exfmt`, and by comparing `ROWS_READ` and `ROWS_SELECTED` for the statement in question. Table scans are often unavoidable for ad hoc queries, but a repeated query that creates a bottleneck due to too much I/O should be addressed. Out-of-date statistics or an indexing problem might be behind the use of a table scan. On the other hand, if the affected table is small enough, increasing buffer pool size might be sufficient to reduce I/O and eliminate the bottleneck. For more information, see *Writing and Tuning Queries for Optimal Performance* and *Physical Database Design*.

Finally, consider two unusual cases of data container disk bottlenecks:

1. We would expect a table scan to drive large disk reads through the prefetchers. If there is a problem with prefetching (see 'Lazy System' bottlenecks), reads into the buffer pool can be done by the agent itself, one page at a time. Depending on the circumstance, this can result in a mostly-idle ‘lazy system’, or (as we are considering here) a disk bottleneck. So if the bottlenecked container is being driven by a table scan, but the read sizes in `iostat` appear much smaller than the prefetchsize for that table space, insufficient prefetching might be the problem.

2. Ordinarily, page cleaning drives a steady stream of page writes out to the table space, in order to ensure a good supply of available buffer pool pages for use by subsequent table space reads. However, if there are problems with the tuning of page cleaning (see 'Lazy System' bottlenecks), the agent itself can end up doing much of the cleaning. This often results in ‘bursty’ cleaning – sporadic periods of intense write activity (possibly creating a disk bottleneck) alternating with periods of better performance.

More information on diagnosing and solving these two problems is contained in 'Lazy System' bottlenecks.

---

**System Bottleneck > Container Disk Bottleneck > Hot Index Container > Hot Index?**

A bottleneck in a container is more likely to be due to table activity than index activity, but after we rule out a table as a likely cause, we should investigate the possibility that index activity is causing the problem. Because we don’t have index snapshots to work with, we have to come at the problem somewhat indirectly.

1. Is there a high level of index read or write activity in this table space? Query `SYSIBMADM.SNAPTBS` for the table space with `TBSP_ID` matching `TBSP_ID` of `SNAPCONTAINER` from above.

   ```sql
   SELECT DBPARTITIONNUM, TBSP_ID, TBSP_NAME, 
       POOL_INDEX_P_READS, POOL_INDEX_WRITES
   FROM SYSIBMADM.SNAPTBS T
   WHERE T.TBSP_ID = <table space id of hot container>
   ```
A large and increasing value for POOL_INDEX_P_READS or POOL_INDEX_Writes indicates one or more ‘busy indexes’ in this table space.

2. Which indexes are in this table space? Query SYSCAT.TABLES, matching INDEX_TBSPACE with SNAPCONTAINER.TBSP_NAME, as above.

```
SELECT T.TABNAME, I.INDNAME
FROM SYSCAT.TABLES T, SYSCAT.INDEXES I
WHERE T.TABNAME = I.TABNAME AND
COALESCE(T_INDEX_TBSPACE, T.TBSPACE) =

<name of table space with hot container>
```

3. Which of these indexes are highly active? If there is more than one index in the table space that we are investigating, we need to look for high activity at the index level. Collect data from multiple iterations of `db2pd -tcbstats index -db <dbname>`. The ‘TCB Index Stats’ section lists all currently active indexes, and statistics for each. The ‘Scans’ column indicates how many index scans have been executed for each index. Using the list of indexes from the hot table space, look for an index whose counts for Scans, KeyUpdates, or ‘InclUpdats’ (updates of include column values) is steadily, even rapidly, increasing.

```
System Bottleneck > Container Disk Bottleneck > Hot Index Container > Hot Index > Buffer pool too small?

Index access is generally desirable, so at this point it is reasonable to investigate whether more of the hot index can be kept in the buffer pool, rather than having to be read from disk. Increasing the buffer pool size, or relocating the index to a dedicated buffer pool, might reduce I/O enough to eliminate the bottleneck. In data warehousing environments, where indexes are often very large, it might be impossible to make enough buffer pool space available to reduce I/O sufficiently. In that case, reducing the bottleneck by improving disk I/O bandwidth through adding additional containers might be more effective.
```

```
System Bottleneck > Container Disk Bottleneck > Hot Index Container > Hot Index > Hot SQL Statement?

If we cannot eliminate the index I/O bottleneck through tuning, such as what occurred when we identified a ‘hot table’, we might have to drill down further to find the SQL statements that are driving the index I/O. Unfortunately, we can’t just mine the SQL statement text as we did before, at least not directly.

Using the index names that are showing high activity, we determine the tables corresponding to those indexes, and then use the table names and the methods described above for hot tables to find the dynamic and static SQL statements that might be using those indexes. (There is no guarantee that a reference to the table means that the index used.) We then need to use `db2exfmt` to determine whether the statements used those indexes.
```
If the hot container belongs to a temporary table space, we need to consider a couple of possible causes:

1. Is the high level of temporary table space I/O due to spilled sorts? This is the case in which sorting activity overflows the designated in-memory buffers and must use a temporary table space instead. If the sort time and spilled sorts snapshot monitor elements are high and increasing, this might be the cause.

   ```sql
   SELECT DBPARTITIONNUM, TOTAL_SORTS, TOTAL_SORT_TIME, SORT_OVERFLOWS
   FROM SYSIBMADM.SNAPDB
   ```

   The STMM tries to avoid this kind of situation; however if you are not using the STMM to control `sheapthresh_shr` and `sortheap`, you might want to manually increase these values.

2. Is the I/O due to large intermediate results? This is revealed through high numbers of temporary data physical reads or writes. We would initially check for these in the table space-level snapshot data, and if there was evidence of high temporary data I/O, we would then drill down into the dynamic SQL snapshot or static SQL event monitor data, looking for individual statements that caused high levels of temporary buffer pool activity.

   ```sql
   SELECT DBPARTITIONNUM, TBSP_NAME, POOL_TEMP_DATA_P_READS, POOL_DATA_WRITES
   FROM SYSIBMADM.SNAPTBSP
   WHERE TBSP_ID = <table space id of hot container>
   ```

Suppose that we have identified one of the above types of containers as the bottleneck, but – as is sometimes the case – beyond that we can see no obvious single cause. No hot table, no hot index, no hot SQL statement. There are a few possible causes to investigate:

1. Are there too many ‘fairly active’ tables or indexes in the table space? Even if none of them is individually active enough to cause the bottleneck on its own, it is possible that the aggregate activity might be too much for the underlying disks. One answer would be to distribute the tables and indexes over multiple table spaces. Another possibility would be to add more containers to the table space (provided they were on different disks than the existing containers, so that I/O capacity was increased).
2. Are there too many table spaces sharing the same disks? This can even happen inadvertently, when table spaces occupy seemingly separate logical volumes that nevertheless use the same physical disks underneath. As above, total activity – this time across table spaces instead of tables – might be to blame. The logical response here would be to move one or more of the table spaces to other disks.

If we get to this point without finding a specific cause of our container disk bottleneck, we have effectively ruled out the vast majority of ‘tunable problems’, and should consider adding additional disk throughput capacity to the problem table space to improve performance.

System Bottleneck > Log Disk Bottleneck?

Although container disk bottlenecks are more common, a log disk bottleneck can have a greater impact on system performance. This is because a slow log can interfere with all INSERT, UPDATE, or DELETE statements on the system, not just those affecting a particular table or index. As with other types of disk bottlenecks, the main symptom is very high utilization, as reported in `iostat` or `perfmon` (90% or higher). A log bottleneck also causes long commit times in the statement event monitor, and a higher-than-normal proportion of agents in ‘commit active’ state in the application snapshot.

As mentioned in the section on log configuration, the log not share a disk with anything ‘active’ (such as a container, for example) during database operation. This is one of the first things to verify in the case of a log bottleneck.

If the log has its own disks, we need to understand the nature of the bottleneck.

1. If `iostat` shows that the log device is performing more than 80-100 operations per second, and if the average I/O size is about 4 KB, this indicates that the log is more saturated by I/O operations than by sheer data volume.

   There are a couple of ways to influence this:

   - Some applications in the system might be committing very frequently – possibly more frequently than necessary. Applications with high commit rates can be identified through the application snapshot, by comparing the ratio of commits to SELECT, INSERT, UPDATE, or DELETE statements, and also by looking at the number of commits per minute. In the extreme case (with autocommit enabled, and with short SQL statements, for example), there is the potential to saturate the log device. Reducing commit frequency in the application has a direct benefit in reducing the log bottleneck.

   - Another possible cause of frequent log writes is the log buffer being too small. When the log buffer fills up, the DB2 system must flush it to disk, regardless of whether there was a commit. A rapidly increasing number of log buffer full conditions (as reported by the NUM_LOG_BUFFER_FULL element in SYSIBMADM.SNAPDB) indicates that this is the likely cause of the problem.
2. A log bottleneck can also be caused by an excessive volume of data being written. If, along with high device utilization, `iostat` also shows that writes to the log device are much larger than 4 KB, this indicates that data volume is more of a factor than high transaction rate.

It might be possible to reduce the volume of data being logged:

a. When a DB2 system updates a table row, it logs all column data from the first modified column to the last modified column – including any columns in between that aren’t being modified. Placing columns that are frequently modified next to one another in the table definition can reduce the volume of data that is logged during updates.

b. Large object (CLOB, BLOB, DBCLOB) columns are logged by default, but if the data they contain is recoverable from outside of the database, it might be appropriate to mark these columns as NOT LOGGED, to reduce the volume of data being logged during insert, update, or delete operations.

c. If the excessive log volume is correlated with bulk SQL operations (such as `INSERT` with `subselect`, as is sometimes used for maintenance and data load procedures), the target table can be set to NOT LOGGED INITIALLY (NLI). This suspends logging during the current unit of work. The recovery procedure of the NLI table needs to be taken into account; however, if it is appropriate in your environment, NLI can provide a significant reduction in log data volume and a corresponding performance increase. Of course, if the bulk operation is a straightforward insert that can be replaced with a call to the load utility, that would eliminate logging as well.

In either case – whether the log bottleneck is due to a very high rate of log writes or a very high volume of data being written – it is often not possible or practical to eliminate the cause of the problem. After you verify that the log configuration follows best practices as described above, you might need to increase the capacity of the log subsystem, either by adding additional disks into the log RAID array, or by providing a dedicated or upgraded caching disk controller.

Heavy disk writes on the DB2 diagnostic path – where the `db2diag.log` is located – can cause an overall system slowdown that is difficult to isolate, because normal DB2 monitoring facilities do not track this. In a partitioned database environment, all partitions write to the same diagnostic path, which is typically shared over the network.
through NFS. Concurrent writes to db2diag.log from a large number of partitions can cause a high network and I/O load, as well as synchronization between partitions, thereby degrading system performance. As mentioned in the configuration section above, a straightforward solution to this is to have a dedicated diagnostic path (and hence a dedicated db2diag.log file) for each partition.

Setting the `diaglevel` database manager configuration parameter to 4 increases the volume of diagnostic messages by several factors, which can have a significant performance impact – in particular in large partitioned database environments. A dramatic slowdown of performance might even be followed by an eventual system stall due to a file system full condition on DIAGPATH. To avoid this, you should verify that there is sufficient free space in the diagnostic file system, by archiving DB2 diagnostic information or assigning a dedicated file system for the DB2 diagnostic information.

Disk bottlenecks: The overall picture

- **Inadequate disk configuration or subsystem?**
  - **Bad plans giving excessive index scanning?**
    - Need more or different indexes?
    - Buffer pool too small?
  - Insufficient sort heap? Missing indexes?

- **Data table space**
- **Index table space**
- **Temp table space**
- **Log Devices**

- **System bottleneck type?**

**CPU bottlenecks**

A CPU bottleneck manifests itself in two main ways:

1. **Overall CPU saturation.** All processors on the system are busy. This is generally measured as the sum of user CPU time and system CPU time, reported using the `vmstat` or `perfmon` commands. CPU utilization over 95% is considered saturated.

2. **Individual CPU saturation.** The load on the system is such that one processor is fully saturated, but other processors are partially or completely idle. This generally arises when there is only one heavy application or statement running on the system. Even though there is available CPU capacity, the system cannot consume it, and the speed of the application or statement is therefore limited by the performance of the one busy processor core.

Also consider the difference between user mode and system mode CPU consumption. User CPU time is accumulated while the processor is running software outside the operating system kernel, such as applications or middleware like DB2 for Linux, UNIX, and Windows. System CPU time is accumulated while running in the operating system kernel. These amounts are reported separately, and can help us identify the source of a CPU bottleneck, depending on the distribution between the two. A ratio of between 3:1 and 4:1 in user to system CPU time is typical. If the balance of user to system CPU time in the bottlenecked system is higher than this, first investigate possible causes of increased user CPU time.

---

**System Bottleneck > CPU Bottleneck > User CPU Bottleneck**

Many causes of a user CPU bottleneck on a DB2 server can be diagnosed through the snapshot and statement event monitors. Drill down to find out which users are consuming the most CPU time by using the application snapshot, or querying the `SYSIBMADM.SNAPAPPL` administrative view:

```sql
SELECT APPL.DBPARTITIONNUM, APPL_NAME,
       AGENT_USR_CPU_TIME_S + AGENT_USR_CPU_TIME_MS / 1000000.0 AS USER_CPU
FROM SYSIBMADM.SNAPAPPL APPL, SYSIBMADM.SNAPAPPL_INFO APPL_INFO
WHERE APPL.AGENT_ID = APPL_INFO.AGENT_ID AND
      APPL.DBPARTITIONNUM = APPL_INFO.DBPARTITIONNUM
ORDER BY USER_CPU DESC
```
Similarly, you can also determine which SQL statements are using the most CPU time from the dynamic SQL snapshot, or by querying the SYSIBMADM.SNAPDYN_SQL administrative view:

```sql
SELECT SUBSTR(STMT_TEXT, 1, 200),
       TOTAL_USR_CPU_TIME + TOTAL_USR_CPU_TIME_MS / 1000000.0
       AS USER_CPU
FROM SYSIBMADM.SNAPDYN_SQL
ORDER BY USER_CPU DESC
```

In general, look for one or more statements that are consuming ‘more than their fair share’ of CPU. This translates to high and increasing values of `TOTAL_USR_CPU_TIME` and `TOTAL_USR_CPU_TIME_MS`.

At the same time, consider static SQL statements. As mentioned above, they are not covered by the snapshot monitors, so use either a statement event monitor or a WLM activity event monitor to collect CPU usage information for them. The following steps help minimize statement event monitor overhead:

1. The default BUFFERSIZE for event monitors is four pages, which should be increased to 512 pages for better performance.

2. Create the statement event monitor tables in a separate table space. This avoids I/O conflicts between the statement event monitor data and other tables. It also gives you the opportunity to choose a larger page size, which helps minimize truncation of SQL statements (see below).

3. Use the TRUNC option on the CREATE EVENT MONITOR statement. This forces the SQL statement text to be saved in the same row as other statement metrics, such as elapsed time, CPU time, and so on, rather than being stored in a separate LOB column. This can result in the SQL statement being truncated in the event monitor output. The portion of the SQL statement that can be stored this way depends on the page size used for the statement event monitor tables (for example, approximately 3500 bytes in a 4-KB page).

4. Use the WHERE clause to focus monitoring on a subset of connections or applications. Although the monitored connections experience some additional overhead, using WHERE reduces the total system overhead due to event monitoring.

Putting these all together, and obtaining the application ID of the connection you’re interested in from `LIST APPLICATIONS`, we would have something like the following:
CREATE EVENT MONITOR STMT_EVT FOR STATEMENTS
WHERE APPL_ID = '*LOCAL.DB2.075D83033106'
WRITE TO TABLE
    CONNHEADER(TABLE STMT_EVT_CH, IN TBS_EVMON),
    STMT(TABLE STMT_EVT_STMT, IN TBS_EVMON, TRUNC),
    CONTROL(TABLE STMT_EVT_CTRL, IN TBS_EVMON)
BUFFERSIZE 512

For WLM activity monitors, the principles are similar. A large BUFFERSIZE, a separate table space, and the TRUNC option are all good ideas for reducing overhead. Activity monitors do not support the WHERE clause, but instead have an implicit focus on the activities within the workload or service class where the activity monitor is defined. Similar to the WHERE clause, this focus is a very effective way to reduce the performance overhead and the volume of data collected. Using the WITHOUT DETAILS clause on the activity monitor provides the basic CPU consumption information that we need. Increasing to WITH DETAILS or WITH DETAILS AND VALUES provides extra information that might prove useful, but if monitoring overhead is an issue in your environment, it is usually best to start with WITHOUT DETAILS, and use DETAILS or VALUES if additional information is required.

System Bottleneck > CPU Bottleneck > User CPU Bottleneck > High CPU SQL

We cannot always reduce the amount of CPU a given SQL statement consumes, but there are some cases where we can have an impact.

1. A frequently-executed in-buffer pool table scan can consume a surprising amount of CPU time when a small, hot table is queried or participates in a join, but has no suitable index. Symptoms include:

   • A relatively short statement execution time
   • User CPU consumption approximately equal to the execution time
   • A relation scan in the explain plan
   • A rising number of scans in `db2pd -tcbstats`
   • A low or very low number of buffer pool physical data reads for the statement

Even though this type of statement isn’t usually considered a bottleneck, the frequent execution and high CPU consumption can make it a problem. You can respond by creating an index that gives the optimizer an alternative to the table scan. The right index definition might be obvious from the query, but if not, the Design Advisor can likely assist here.
2. If the application executing an SQL statement consumes only a fraction of the rows the statement produces, using the OPTIMIZE FOR \( n \) ROWS (OFnR) or FETCH FIRST \( n \) ROWS ONLY (FFnRO) clauses can help reduce resource consumption of all types, including CPU. In particular, OFnR and FFnRO can help optimize the SQL access plan to return the initial rows of the result set most efficiently, rather than optimizing for the return of all rows in the result set to the calling application. If only OFnR is used, \( n \) can be exceeded at run time; however, FFnRO prevents more than \( n \) rows from being returned, even if the application attempts to do so.

3. As mentioned in the configuration section above, the use of a culturally correct collating sequence with a Unicode code page can introduce a significant amount of overhead, particularly in CPU consumption. Because the amount of overhead is directly related to the number of string comparisons that SQL statements make (for example, in predicates or in sorting due to an ORDER BY clause), if we reduce the number of comparisons a statement makes, we reduce its CPU consumption. A reduction in the number of comparisons can often be achieved by encouraging the use of indexes for both predicate evaluation and result set ordering. The Design Advisor can be very helpful in designing the appropriate indexes to minimize table scans and sorts (see Physical Database Design).

4. Locking issues are often thought of only in terms of conflicts and wait time; however, even when there are few or no conflicts, the process of acquiring and releasing locks can consume a significant amount of CPU time. Consider an application or statement that examines many rows in the table, but produces few lock conflicts because it runs on its own, because it has exclusive access to the tables it references, or because all concurrent applications only use the table in read-only mode. In a case like this, it might be possible to use table-level locking to achieve the required level of isolation while reducing CPU.

If no individual SQL statements appear to be consuming the bulk of the CPU cycles, there are broader potential issues that can cause an overall increase in CPU usage.

<table>
<thead>
<tr>
<th>System Bottleneck</th>
<th>CPU Bottleneck</th>
<th>User CPU Bottleneck</th>
<th>Dynamic SQL without Parameter Markers</th>
</tr>
</thead>
</table>

1. Many applications build their SQL statements ‘on the fly’ by concatenating statement fragments and literal values, rather than by using parameter markers. (For complex SQL statements querying tables with distribution statistics, embedded literals can help the SQL optimizer choose a better access plan. Instead, we are focusing on lightweight statements where embedded literals have no benefit.)
String procNameVariable = "foo";
String query =
    "SELECT language FROM "
+ "syscat.procedures "
+ "WHERE procname = "
+ " \"
+ procNameVariable // inject literal value
+ " \";

Even though the statement strings generated in this way differ from each other only in the literal value they contain, when the application prepares the queries, the DB2 system has to compile them each time, rather than find them in the dynamic SQL cache. Even for very simple statements, whose compile cost is very low, the aggregate cost of a high statement volume can be significant.

Two signs that this problem might be occurring are:

a. A large number of similar but not identical statements in the dynamic SQL snapshot, which turn out to differ only by the literal values that have been embedded in the statements.

b. A steadily increasing value of package cache inserts (SYSIBMADM.SNAPDB.PKG_CACHE_INSERTS), even after the system reaches steady state.

The best practice for avoiding this overhead is to ensure that applications use parameter markers for simple dynamic SQL statements.

**System Bottleneck > CPU Bottleneck > User CPU Bottleneck > Utilities Running?**

2. DB2 utilities are designed to scale well and exploit system resources to get the job done as quickly as possible. That can mean that while a utility is running, there might be a significant increase in CPU consumption. Load and runstats are good examples of utilities that often drive high CPU consumption, but under the right circumstances, other utilities can do so as well. To see which utilities are currently running, use the LIST UTILITIES SHOW DETAIL command.

If a utility is executing, we can drill down to determine its CPU consumption as follows. On UNIX and Linux systems, prior to the introduction of the threaded engine in DB2 Version 9.5, the various worker processes that utilities used were visible (along with their CPU consumption, for example) through a simple ps command. As of DB2 Version 9.5, the db2pd –edu command shows all of the threads inside the DB2 engine (see the [DB2 Process Model](#)), including their user...
and system CPU usage. This is very useful in determining if any of the worker threads are behind the CPU bottleneck.

Setting UTIL_IMPACT_PRIORITY can be helpful in restricting the amount of impact the backup and the runstats utilities have on the system. In addition, the overhead and run time of runstats can be reduced by gathering statistics on only the columns involved in SQL predicates (and which therefore need statistics), if this information is known, and making use of sampling runstats. Good recommendations for the latter are included in Writing and Tuning Queries for Optimal Performance.

By default, the LOAD command creates a formatter thread (db2lfrm) for each CPU, but by using the CPU PARALLELISM N option, we can reduce the number of formatters to N, leaving more CPU capacity for the rest of the system. In general, throttling a utility with UTIL_IMPACT_PRIORITY or CPU PARALLELISM extends the run time of the utility proportionately.

---

3. When a system temporary table is no longer needed and is dropped, a DB2 system must remove its unneeded pages from the buffer pool. If this happens frequently, and if temporary tables share a buffer pool with regular user data, the result can be extra CPU cycles consumed in resolving conflicts and processing the pages. This problem is more common on systems that perform transaction processing rather than complex queries. If table snapshots show that there are a significant number of temporary tables being created and destroyed, best practice is to place temporary table spaces in their own buffer pools. This eliminates the extra conflict and processing overhead, and can contribute to reduced CPU consumption.

---

Although user CPU tends to be the dominant factor in most CPU-bound environments, system CPU time can sometimes be a dominant factor, but the number of problems that we can diagnose and solve is quite a bit smaller.

One cause of high system CPU time that is relevant to DB2 systems is a high context switch rate in the operating system (OS). A context switch is used by the OS to alternate between the different tasks it needs to handle. Context switches are triggered by a number of different rules in the OS, and generally provide a smooth progression of all work that the system must handle. However, when context switches are triggered too frequently, they themselves can end up consuming a significant amount of CPU time. On UNIX systems, context switches are reported using the vmstat command, under the ‘CS’ column. A rate of more than 75,000 to 100,000 context switches per second would be considered quite high.

---

A common cause of a high context switch rate in a DB2 system is the presence of a very large number of database connections. Each connection has one or more database agents working on its behalf, so if the connections are active – particularly with short
transactions – a high context switch rate and high system CPU consumption can result. One way to avoid this is to enable the DB2 connection concentrator. It allows multiple connections to share a single agent, thereby reducing the number of agents (saving memory footprint), and reducing the context switch rate.

Device interrupts can also be a cause of high system CPU time. An interrupt occurs when a device, such as a network adaptor, needs ‘attention’ from the OS. The cost of an individual interrupt is not high; however, if the interrupt rate climbs too high, the aggregate load on the system can be significant. Fortunately, modern network and disk adapters have a high degree of independence from the OS, and cause far fewer interrupts than their predecessors did just a few years ago. Significant overhead from disk interrupts is quite rare; however, in a network-intensive client/server environment (such as many SAP R/3 installations), the load imposed by network interrupts can be quite high. Although there are steps that can be taken to reduce network-driven overhead on the server, this type of tuning is beyond the scope of this paper. In cases such as this, it is best to involve your network administrator to confirm and solve the problem.

If application logic (especially for longer transactions) can be encapsulated in an SQL stored procedure, this can help reduce context switches and network traffic. Not only does this get the application logic onto the server, in the context of context switches, it also pushes the logic right into the DB2 agent. This eliminates the back-and-forth flow of SQL invocations and results – and context switches – between the agent and the client application.

System Bottleneck > CPU Bottleneck > High System CPU > High Device Interrupts

In a DB2 system, strive to exploit system memory by leaving as little of it unused as possible. Unfortunately, if you over-allocate memory – that is, mistakenly configure DB2 or other software to use more than the amount of physical memory on the system – the result is system CPU overhead (and possibly disk overhead) due to paging. This situation is identified on UNIX-based systems by low free memory and high page in or page out activity reported in `vmstat` (the free, pi and po columns, respectively). The solution is to reduce memory allocation below the point where paging starts.

One thing that can make this slightly challenging is the memory consumed due to file system caching. The OS generally uses ‘free’ memory to buffer data from disk, thereby avoiding I/O. Although memory used by the file system cache is available to the DB2 database if needed, you generally want to avoid the case where the DB2 database and the file system get into a ‘tug of war’ over memory. Although file system cache processing itself takes place in user mode (that is, it is not a consumer of system CPU), the virtual memory management involved can drive up system time as well. The DB2 configuration section earlier in this paper makes recommendations on how to avoid file system caching impact in DB2. On AIX, use of the vmo parameter `LRU_FILE_REPAGE=0` (also discussed above) can help keep file system cache overhead under control, even outside of DB2.

System Bottleneck > CPU Bottleneck > High System CPU > Over-allocation of Memory

Servers with very large amounts of physical memory – 100 or more GB – can be subject to extra CPU overhead if the system is not configured to use large memory pages. The OS manages memory at a page-level granularity. OS memory pages are different than DB2
pages. A typical page size is 4 KB – meaning that the OS must look after 250 million page table entries in a machine with 100 GB of RAM. Most operating systems support larger page sizes, which helps to reduce the overhead of virtual memory management. The AIX operating system, for example, supports large pages up to 16 GB in size, although these would be very rarely used in practice. DB2 databases automatically use 64-KB pages if they are enabled on the system, and other sizes can be manually selected. (For more information, see Enabling large page support in a 64-bit environment.) The best practice on most large memory systems (on AIX systems) is to ensure that 64-KB pages are enabled so that the DB2 system can use them. This is the best compromise between good performance and the potential side-effects of using even larger page sizes. On Linux systems, large page support for DB2 databases must be manually enabled with DB2_LARGE_PAGE_MEM.

On AIX systems, vmstat -P ALL shows what page sizes are available and in use on the system. If 64-KB pages are enabled on the system and the DB2 database is running, you should see large allocations of 64-KB pages in vmstat -P ALL, where the DB2 database manager has allocated memory. If not, and the system has a large amount of RAM, this might be the cause of higher-than-normal system CPU consumption.
Memory bottlenecks

Having sufficient and properly configured memory is critical for good system performance. Without adequate memory, access to data that would otherwise be buffered turns into disk I/O – often creating a disk bottleneck in the process. As well, smaller but just as important amounts of memory are used to store metadata and calculated results, such as SQL access plans and locks. With insufficient memory for these, the system must discard or collapse important information and either recalculate it or otherwise compensate with additional processing, increasing CPU overhead. Thus, a memory bottleneck can actually disguise itself as a disk or CPU problem.

The following table summarizes disk and CPU bottlenecks that have memory as a potential underlying cause.

<table>
<thead>
<tr>
<th>Bottleneck Type</th>
<th>Primary symptom</th>
<th>Memory issue potentially causing or worsening bottleneck</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk</td>
<td>Data or index table space bottleneck</td>
<td>• Buffer pool too small</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Total system memory too small</td>
</tr>
<tr>
<td>Disk</td>
<td>Temporary table space bottleneck</td>
<td>• Values for sortheap or sheapths_shr too small</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Total system memory too small</td>
</tr>
<tr>
<td>Disk</td>
<td>Log disk bottleneck</td>
<td>• Log buffer size too small</td>
</tr>
<tr>
<td>CPU</td>
<td>CPU bottleneck due to repeated package cache inserts</td>
<td>• Package cache too small</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Total system memory too small</td>
</tr>
<tr>
<td>CPU</td>
<td>Excess system CPU time spent in VMM</td>
<td>• Total system memory over-allocated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Very large system memory managed with small OS memory pages</td>
</tr>
</tbody>
</table>

Most memory bottlenecks manifest themselves with the symptoms of either a CPU or a disk bottleneck, and memory issues are key possibilities in the investigation of such problems. However, it can also happen that a shortage of available memory (as reported from running the `vmstat` command), along with the poor performance that led us down this path in the first place, are the dominant symptoms. If further examination of `vmstat` data indicates sustained paging activity (with or without elevated system CPU usage), there is excessive memory pressure on the system.

Memory allocation in DB2 systems falls into two broad categories:

1. **Database and instance-level allocations**, such as shared database memory, from which things like the buffer pools, the sort heap, the lock list, and the package cache are allocated. In identifying the source of memory over-allocation, shared
memory is fairly straightforward to deal with, because the combined allocation is independent of the number of database connections. These allocations are capped by the DATABASE_MEMORY configuration parameter.

2. **Connection-level or application allocations**, such as the application heap and the statement heap. The total amount of memory consumed by connection-level allocations depends on the number of connections, which can make it somewhat more difficult to predict at the time of initial system configuration. In DB2 Version 9.5, per-application memory comes under the INSTANCE_MEMORY umbrella, so ‘runaway’ allocations due to sudden spikes in connections are less likely.

Whether explicit heap sizes have been specified, or the STMM is used instead, the best way to determine actual DB2 memory consumption is with the memory usage sections of the database, database manager, and application snapshots (or SYSIBMADM.SNAPDB_MEMORY_POOL, SNAPDBM_MEMORY_POOL, and SNAPAGENT_MEMORY_POOL administrative views), which provide configured and current heap sizes at the database, instance, and application levels, respectively. These enable you to examine configured and current heap and application allocations, and to determine the current total allocation.

The top-level limit of DB2 Version 9.5 memory usage is enforced with the INSTANCE_MEMORY database manager configuration parameter, and the DATABASE_MEMORY database configuration parameter. On most platforms, these default to AUTOMATIC, meaning memory usage by instance and by database can fluctuate, and that (where supported) the DB2 database product acquires memory from and returns memory to the OS, depending on the needs of the DB2 database and system conditions. (Current numeric values of these parameters are obtained using the SHOW DETAIL option on the GET DATABASE MANAGER CONFIGURATION and GET DATABASE CONFIGURATION commands.) Thus, the DB2 memory management system is designed to avoid the kind of memory bottleneck considered here.

Memory bottlenecks can still occur under the following conditions:

1. If INSTANCE_MEMORY has been explicitly set to a numeric (non-AUTOMATIC) value, and the STMM is enabled, the STMM tunes DB2 memory consumption right up to the value of INSTANCE_MEMORY. In this case, the STMM does not react to memory pressure on the system by freeing memory (again, because INSTANCE_MEMORY was not set to AUTOMATIC). Thus, the combination of the DB2 system and other big memory consumers, such as application servers, might push the total system-wide memory usage too high. The ps command on UNIX systems or the task manager on Windows shows memory usage by process, and is an invaluable tool to track down memory hogs outside of the DB2 database.

As mentioned previously, although file system cache memory is technically available to consumers such as the DB2 database, it can also be the case that large amounts of file system cache (particularly of modified data that must be flushed
out to disk before the memory can be released to other memory users) can cause paging to occur if additional demands on memory build up.

If significant memory consumption from things other than DB2 databases on the server is unavoidable, the INSTANCE_MEMORY parameter should either be set to AUTOMATIC, or reduced to reflect the real portion of system memory to which DB2 databases are entitled.

2. An explicit value of INSTANCEMEMORY that is too high for the system might not be an issue until additional databases are brought online, pushing the total DB2 database allocation higher than the system can accommodate, but still within INSTANCE_MEMORY.

Because the STMM is designed to cope with memory pressure by freeing memory back to the OS if required, this scenario would be most likely to occur when the STMM is not enabled, or when the platform involved (for example, the Solaris operating system) does not support the release of memory back to the OS, or when the memory demands of the database are extremely dynamic (for example, rapid creation or destruction of database connections, or very short-term activation of the database).

3. If a very large number of database connections is required, this can result in a large portion of the instance memory being consumed by agents. If this amount is excessive – leaving too little memory for database global memory allocations either with or without the STMM – it can be reduced through the use of the connection concentrator.

‘Lazy System’ bottlenecks

The fourth and most interesting category of bottlenecks that we examine is ‘lazy system’ bottlenecks. These represent cases in which none of the previous bottleneck areas appear to be at fault. There is no apparent bottleneck caused by factors related to CPU, disk, memory, or network, yet the system cannot be pushed any further.

A very common culprit in a ‘lazy system’ is lock contention. Fortunately, lock contention is easy to detect in snapshot data. The snapshot elements LOCK_WAIT_TIME and LOCKS_WAITING, available through SYSIBMADM.SNAPDB, indicate total lock wait time and number of agents currently waiting on locks, respectively. A high percentage of active agents waiting on locks (for example, 20% or higher) or an increasing value of lock wait time indicates that locking might be a significant bottleneck.

There are no per-statement lock wait time diagnostics available; however, extended execution time relative to CPU consumption and I/O activity in dynamic SQL snapshot and statement event monitors are generally reasonable indications of a statement that is
affected by lock wait time. The application snapshot also reports lock wait time by application, which can be very useful in narrowing down where a system-side lock wait problem originates. We can get further lock wait information from the SYSIBMADM.SNAPLOCKWAIT view. It shows:

- Lock type – shared or exclusive
- Object type – row, table, and so on
- Agent ID of holder and requestor

Unlike many other types of DB2 snapshot monitor data, locking information is very transient. Apart from LOCK_WAIT_TIME, which is a running total, most other lock information goes away when the locks themselves are released. Thus, lock and lock wait snapshots are most valuable if collected periodically over a period of time, so that the evolving picture can be better understood. As suggested early on, the best practice for analyzing large volumes of monitor data is collecting it through the administrative views and storing it in DB2. This is especially true for data from SNAPLOCKWAIT, which is composed of data from application, database, and lock snapshots, and is not available in this form from the GET SNAPSHOT command.

Also, unlike other types of snapshots, the main overhead in lock snapshots is in taking the snapshot, not in having the snapshot enabled through the lock monitor switch. So, although it is useful to have regular lock snapshots collected and stored for analysis, overly frequent snapshots can cause bottlenecks on their own.

There are a number of guidelines that help to reduce lock contention and lock wait time.

1. If possible, avoid very long transactions and WITH HOLD cursors. The longer locks are held, the more chance that they cause contention with other applications.

2. Avoid fetching result sets that are larger than necessary, especially under the repeatable read (RR) isolation level. The more that rows are touched, the more that locks are held, and the greater the opportunity to run into a lock that is held by someone else. In practical terms, this often means pushing down row selection criteria into a WHERE clause of the SELECT statement, rather than bringing back more rows and filtering them at the application. For example:

```sql
exec sql declare curs for
    select c1,c2 from t
    where c1 not null;
exec sql open curs;
do {
    exec sql fetch curs
        into :c1, :c2;
} while( P(c1) != someVar );
```

```sql
exec sql declare curs for
    select c1,c2 from t
    where c1 not null
        and myUdfP(c1) = :someVar;
exec sql open curs;
exec sql fetch curs
    into :c1, :c2;
```
3. Avoid using higher isolation levels than necessary. Repeatable read might be necessary to preserve result set integrity in your application; however, it does incur extra cost in terms of locks held and potential lock conflicts.

4. If appropriate for the business logic in the application, consider modifying locking behavior through `DB2_EVALUNCOMMITTED`, `DB2_SKIPDELETED`, and `DB2_SKIPINSERTED`. These registry variables enable DB2 to delay or avoid taking locks in some circumstances, thereby reducing contention and potentially improving throughput.

Lock escalation can also be a major source of contention. Whereas individual row locks taken by a well-designed application might not conflict, block- or table-level locks resulting from escalation are much more likely to cause serialization and severe performance problems. The database snapshot reports a global count of lock escalations (SYSIBMADM.SNAPDB.LOCK_ESCALS). This is most easily broken down to escalations in individual tables by examining the messages written to db2diag.log (at DIAGLEVEL 3) when each escalation occurs.

Lock escalation is triggered when an application consumes its allowed portion of the lock list (determined by the database configuration parameter `MAXLOCKS`, which is expressed as a percentage of the lock list size). Thus, increasing `MAXLOCKS` or `LOCKLIST` can reduce the likelihood or frequency of escalations. As well, as mentioned above, reducing the number of locks taken by applications (through increased commit frequency, reducing the isolation level, and so on) tends to reduce escalations.

**System Bottleneck > Lazy System > Deadlocks and Lock Timeout**

Although lock wait time can be quite a subtle bottleneck, deadlocks and lock timeouts are harder to ignore, because they both return negative SQL codes to the application. Even so, many applications retry the failed transaction and eventually succeed without reporting the deadlock. In this case, the most straightforward indication of a potential deadlock issue is the DEADLOCK element in the SYSIBMADM.SNAPDB administrative view. As mentioned above, we recommend collecting this as part of regular operational monitoring.

The cost of a deadlock varies, and is directly proportional to the length of the rolled-back transaction. All the same, more than one deadlock per 1000 transactions generally indicates a problem.

Deadlock frequency can sometimes be reduced simply by ensuring that all applications access their common data in the same order – meaning, for example, that they access (and therefore lock) rows in Table A, followed by Table B, followed by Table C, and so on. If two applications take incompatible locks on the same objects in different order, they run a much larger risk of deadlocking.

The default deadlock event monitor `DB2DETAILDEADLOCK` records information about all deadlock participants under `dftdbpath` (the database manager default database path), for example in `<dftdbpath>/NODE0000/SQL00001/db2event/db2detaildeadlock`. Like all
event monitors, it imposes a small amount of additional overhead; however, the benefit from being able to track deadlocks usually outweighs the small performance penalty.

Lock timeouts in excessive numbers can be as disruptive to a system as deadlocks. Because the regular deadlock event monitor does not track lock timeouts, we need a different mechanism. DB2 Version 9.5 supports a technique to generate a text-based report of lock timeouts, which builds on the deadlock event monitor, plus other infrastructure inside the engine. This greatly simplifies lock timeout diagnosis over previous releases. The process for producing the lock timeout report is described on developerWorks: “New options for analyzing lock timeouts in DB2 9.5”.

System Bottleneck > Lazy System > Insufficient Prefetching?

Queries that require large amounts of data to be sequentially read from disk are far more efficiently executed when DB2 prefetchers read the data, than when the agent itself reads the data. There are several good reasons for this:

- The prefetchers bring in multiple pages with each read, the size of which is controlled by database or table space prefetch size, whereas agents read a single page at a time.
- The agent can be executing part of the query while the prefetchers do their work, reducing serialization of computation and I/O.
- Multiple prefetchers can each be assigned a range of pages to read, achieving I/O parallelism.

When the agent needs data from a range of pages, it queues a prefetch request. When the time comes for the agent to use a page, but the prefetcher has not yet started the I/O for that page (that is, if the page has not been requested from the prefetcher at all, or if the request is still in the prefetch queue), the agent itself reads that single page. This reduces the frequency with which the agent has to wait for the prefetcher (it only waits if the I/O is actually in progress). However, all the benefits of prefetching disappear if we have to fall back to agent I/O. Symptoms of this problem include:

- A ‘prefetch ratio’ of less than 100% for statements with large scans. (At the database or buffer pool level, the target value drops, depending on what fraction of total activity is not scan-based.) We define this metric similarly to the buffer pool hit ratio, but here calculating the ratio of number of physical reads done by the prefetcher, compared to the total number of physical reads:

\[
100\% \times \frac{\text{pool_data_p Reads} - \text{async_data_reads}}{\text{pool_data_p Reads}}
\]

This can be calculated at the database level with SYSIBMADM.SNAPDB, or at the buffer pool level with SYSIBMADM.SNAPBP, or at the dynamic SQL statement level with SYSIBMADM.SNAPDYNSQL.
• High and climbing ‘time spent waiting for prefetch’, reported in
PRECETCH_WAIT_TIME in SYSIBMADM.SNAPDB and
SYSIBMADM.SNAPAPPL. As mentioned above, the agent only waits for
prefetch I/O that is actually ‘in-flight’.

• As with other ‘lazy system’ problems, you also generally see a large amount
of idle time in `vmstat` and `perfmon`. However, there can also be increased I/O wait
time, because agents reading single pages are far less efficient than prefetchers
doing big-block reads. But even so, it is unlikely that the I/O wait climbs high
enough to appear to be the bottleneck.

The most likely cause of this problem is that the number of prefetchers (database
configuration parameter NUM_IOSERVERS) is too low. The AUTOMATIC setting
available in DB2 Version 9.1 and later uses factors such as table space parallelism, for
example, to calculate the number of prefetchers, and generally does not require tuning.
However, if tuning appears to be needed based on a low prefetch ratio, the process is as
follows:

1. Determine whether all prefetchers are consuming roughly equal amounts of CPU
time. This can be done with the `ps` command on DB2 Version 9.1 and earlier on
UNIX systems, or with `db2pd -edu` on DB2 Version 9.5 and later. If some
prefetchers are consuming significantly less CPU than others, there are already
enough (and possibly too many) prefetchers. (If there are more than a couple of
‘idle’ prefetchers, you can reduce NUM_IOSERVERS slightly, but having extra
prefetchers is generally not a problem.)

2. Increase NUM_IOSERVERS by 10%. Allow the system to run normally with the
larger number of prefetchers. If there is no improvement in prefetch ratio or in
performance of heavy scan queries, the problem is not being looked at correctly,
and NUM_IOSERVERS should be returned to its previous setting.

3. Repeat this process until you have found the optimum level of prefetchers for the
system

If prefetching still appears to be operating below par, it is also worth verifying that
PREETCHSIZE is set correctly. The process for this is discussed thoroughly in the DB2
Information Center so we won’t repeat it here.

System Bottleneck > Lazy System > Insufficient Page Cleaning?

Similar to prefetching, a problem with buffer pool page cleaning forces agents executing
SQL statements to interrupt their normal processing to do I/O that should normally be
taken care of by one of the DB2 ‘worker threads’. However, in this case, the agent has to
write (a modified page) instead of read. This is generally referred to as a ‘dirty steal’.

The symptoms for a buffer pool page cleaning problem aren’t quite as cut-and-dry as
those for the ‘lack of prefetching’ problem described above. Poor page cleaning tends to
be more of a problem in an online transaction processing (OLTP) environment, where
there are many DB2 agents operating concurrently. If they cannot find clean buffer pool
pages, and are having to do dirty steals, there could be potentially very many extra single-page writes going to the containers. This means that in this case, instead of the typically idle ‘lazy system’, we might see an I/O bottleneck instead. The degree to which this happens depends, for example, on the number of connections and page cleaning performance.

A related symptom is ‘bursty’ system activity, as seen in `vmstat`. The system might run well for a short time, with all agents working normally, followed by a period in which the majority of agents is blocked, flushing a dirty page to disk. This appears as high I/O wait and a short-to-empty run queue in `vmstat`. When the agents have finished the dirty steal, performance spikes back up again – and the cycle repeats.

Within DB2’s monitoring data, the count of dirty page steals (POOL_DRTY_PG_STEAL_CLNS in SYSIBMADM.SNAPDB) is the best indicator of this problem. We would normally expect only a very few of these in a smoothly-running system, so any non-trivial rising number is cause for some concern.

If page cleaning is falling behind and dirty steals are occurring, the first thing to check is the number of page cleaners (database configuration parameter NUM_IOCLEANERS). DB2 Version 9.1 and later versions support the AUTOMATIC setting, which follows the best practice of one cleaner per processor in the current partition. In DB2 Version 9.5, extra cleaners beyond the recommended number can actually hurt performance.

The DB2 database product supports two types of page cleaning: ‘classic’ reactive page cleaning (the default) and proactive page cleaning, introduced in DB2 Universal database Version 8.2.

- **Classic page cleaning** is controlled by two database configuration parameters:
  - `chnpgps_thresh` – determines the percentage of modified buffer pool pages at which to activate page cleaning
  - `softmax` – limits the age of the oldest modified page in the buffer pool (LSN gap), thereby controlling recovery time

  Reducing either of these parameters generally makes cleaning more aggressive; however, `chnpgps_thresh` is the preferred way to affect the number of clean pages in the buffer pool. Decreasing `chnpgps_thresh` can help to reduce the number of dirty page steals, and stabilize uneven cleaning. Setting this parameter too low can result in excessive disk writes, so it should be set just low enough to avoid dirty steals.

- **Proactive page cleaning** (also known as alternate page cleaning, or APC) is enabled using the registry variable `DB2_USE_ALTERNATE_PAGE_CLEANING`. It differs from classic page cleaning in that it adjusts its cleaning rate to maintain the desired LSN gap. Rather than cleaning being ‘on’ or ‘off’, triggered or not, APC can throttle its activity to avoid the ‘bursty’ behavior that is sometimes seen with classic page cleaning. Similar to classic page cleaning, reducing `softmax` effectively increases the rate of cleaning and should reduce dirty steals. APC is
controlled only by softmax, not by chngpgs_thresh, so that DBAs enabling APC for the first time might have to tune softmax if their system was previously cleaned based on hitting chngpgs_thresh (that is, dirty page threshold triggers).

System Bottleneck > Lazy System > Application side problem?

The back-and-forth synchronous flow of requests and responses between a client application and the DB2 server means that both play a role in the performance of the overall system. An increase in the run time of a batch application, for example, could be due to a slow-down at the server, but it could also be caused by a decrease in the rate at which the application makes requests to the DB2 database. The symptoms of this type of problem at the server tend to fit the ‘lazy system’ mold quite well.

Symptoms of a reduction in the rate at which requests arrive at the DB2 database include:

- An increased number of agents in ‘UOW wait’ status in the application (APPL_STATUS in SYSIBMADM.SNAPAPPL_INFO), meaning that they are waiting longer for more work

- An increased time between arrivals of requests to agents (STATUS_CHANGE_TIME in SYSIBMADM.SNAPAPPL_INFO)

- An increased time between requests made at the client side, as seen by the statement event monitor, or a CLI or JDBC trace. CLI and JDBC traces capture API calls at the client side, and record timestamps when the calls were made. Although the overhead for client-side traces is high, they have the advantage that their timings include network response time and other factors outside of the DB2 engine.

- If available, application-side metrics, such as business-level transaction throughput or response time, might show a degradation.

If an application-side slowdown appears to be the problem, possible causes include:

- Deployment of a new version of the application

- A network bottleneck between client and server

- Excessive load on the client system; for example, too many users or too many copies of the application running
System bottlenecks – the Overall Picture

- System bottleneck type?
  - Disk bottleneck
  - Index table space
  - Temp table space
  - Data table space
  - Log devices
  - Memory bottleneck
  - Instance_memory too high?
  - Database_memory too high?
  - Too much application memory?

- Lock escalation? Lock contention? Deadlocks or lock timeout?
  - Too few prefetchers?
  - Too few cleaners?
  - Application issue?

- Inadequate disk configuration or subsystem?
  - SQL without parameter markers?
    - Too small dynamic SQL cache?
    - Applications connecting or disconnecting?
    - Non-parallelized application?

- Bad plans giving table scan?
  - Old statistics?
  - Need more indexes?
  - Buffer pool too small?

- Insufficient sort heap?
  - Missing indexes?
  - Large intermediate results?

- Anything sharing the disks?
  - High transaction rate:
    - Too-frequent commits?
    - Log buffer filling?
    - High data volume
    - Logging too much data?

- Bad plans giving excessive index scanning?
  - Need more or different indexes?
  - Buffer pool too small?

- Old device drivers
  - Creating or destroying agents
  - Too many connections

- High user time
  - High system time

- CPU bottleneck

- System bottleneck type?
Localized and system-wide troubleshooting

Up to this point, we have dealt with performance issues that are seen in the system as a whole: top-level disk, CPU, memory, and lazy system problems. But performance problems don’t always come in this form. Often, the system as a whole is running well, but there is one user, or one application, or one stored procedure, or one SQL statement, that is experiencing problems. What is different about dealing with localized rather than system-wide performance issues?

Fortunately, the methodical approach to performance troubleshooting that we have built in this paper is equally applicable whether the problem is pervasive or more selective. What we need to be able to do is extract the relevant parts from the large amount of monitoring data that the system can provide.

Assume that an application is performing below expected levels. Before you can launch into a diagnosis, you need to be able to identify the activity on the system that arises from this application.

1. Knowing the application name and the authorization ID under which the application is running, we can use the `LIST APPLICATIONS` command to retrieve the ‘appl ID’ (for example, LOCAL.srees.0804250311139), which is the key to identifying monitor data that is specific to this application.

2. The application snapshot is an excellent source of application-specific monitoring data, and by specifying the appl ID, we can focus on the connection of interest.

```
GET SNAPSHOT FOR APPLICATION APPLID
'LOCAL.srees.0804250311139'
```

From here (or from SYSIBMADM.SNAPAPPL and SYSIBMADM.SNAPAPPLINFO), you can determine many important things about the application, such as the statement being executed when the snapshot was taken, the buffer pool hit ratios, the amount of sort time, the ratio of rows read to rows selected, the CPU time, and elapsed time. In short, you get much of the same information that you used to debug system-level problems, but in this case, it is focused on the application in which you’re interested.

3. For an even deeper dive, use the application ID as an argument to the WHERE clause in the statement event monitor, to focus the rather heavier event monitor data collection on just the application of interest. This gives you statement-by-statement timing, buffer pool information, and CPU consumption information for just your application.

Although you’re not really focused on changing the global CPU consumption or disk activity (remember that things are working well overall), it is still important to
understand the situation in which your application is running. If the system is CPU bound, and our application is CPU hungry, its performance is constrained; likewise for disk activity.

After collecting several application snapshots or event monitor traces, you have almost the same type of monitoring data available that you had for the system-wide troubleshooting. The basic goal is to determine where in the application the bulk of the time is being spent – where exactly is your bottleneck? Which of the SQL statements in the application is taking the longest to run? Which statement consumes the most CPU, or drives the most physical disk I/O? Answering these questions mimics the initial step of the system-wide decision tree.

After you have identified one or more culprit SQL statements, and you have an idea of the kind of bottleneck they are facing, you can apply many of the same approaches discussed in previous sections. This especially includes techniques involving drill-downs to ‘hot SQL statements’, hot tables, and so on – elements that are involved in the localized problem.
Best Practices

Configuration:

- Ensure adequate disk capacity in terms of number of spindles.
- Locate transaction logs on dedicated disks.
- Use the DB2 Data Partitioning Feature for data warehousing deployments larger than 300 GB.
- Consider language-aware collation for best performance with Unicode.
- For ISV applications like SAP, follow the vendor’s configuration recommendations.
- Use the AUTOCONFIGURE command to obtain good initial configuration settings.
- The STMM and other autonomies provide stability and strong performance.
- In partitioned database environments, use a local rather than an NFS-mounted file system for DIAGPATH.

Monitoring:

- Collect basic operational monitoring data regularly, so that background information is available in case of a problem.
- Use the administration views to access and manipulate monitoring data with SQL.
- Monitor non-DB2 metrics, such as CPU utilization and application-level response time.
- Keep track of changes in configuration and environment settings.

Troubleshooting:

- Be methodical—change only one thing at a time, and observe the result carefully.

- Start with the highest-level symptoms—such as a CPU, disk, or memory bottleneck—to rule out unlikely or impossible causes early.

- Drill down into possible causes, refining with each step; for example, I/O bottleneck might lead to container C, which might lead to table T, which might lead to inefficient statement S.

- Don’t make changes to the system on just a ‘hunch’—make sure to understand how the problem you’re trying to fix could cause the symptoms you see.

- Use the same top-down methodical approach for both system-wide problems, and for more localized ones.
Conclusion

This paper considered three key areas that are important to understand when trying to avoid degradations in the performance of your system: configuration, monitoring, and performance troubleshooting.

We made recommendations concerning hardware and software configuration that can help you to ensure good system performance. We discussed several monitoring techniques that help you to understand system performance under both operational and troubleshooting conditions. We also presented a number of DB2 performance troubleshooting best practices for dealing with problems in a step-wise, methodical fashion.

If your system is configured appropriately and monitored well, you can more effectively resolve performance problems that might arise, this can help reduce the total cost of ownership and potentially increase the return on investment for your business.
Further reading

  
  - Physical Database Design
  - Writing and Tuning Queries for Optimal Performance
  - Virtualization: Improving Data Server Utilization and Simplifying Management
  - Database Storage


Contributors

Sonia Ang
Senior Consulting IT Specialist

Michael Cornish
Senior Software Developer
DB2 Level 3 Support

Doug Doole
Senior Software Developer
Information Management Software

Nela Krawez
DB2 Data Warehousing Solutions
Information Management Software

Michael Kwok
DB2 Performance Analyst

Jacques Milman
Server Specialist
Business Intelligence & Data Warehousing

Mika Nikolopoulou
Senior Certified IT Specialist
Information Management Software

Adam Storm
Software Developer
DB2 Kernel Development
Notices

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

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

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

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

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

Without limiting the above disclaimers, IBM provides no representations or warranties regarding the accuracy, reliability or serviceability of any information or recommendations provided in this publication, or with respect to any results that may be obtained by the use of the information or observance of any recommendations provided herein. The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any recommendations or techniques herein is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Anyone attempting to adapt these techniques to their own environment do so at their own risk.

This document and the information contained herein may be used solely in connection with the IBM products discussed in this document.

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

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

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

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

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

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

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

COPYRIGHT LICENSE: © Copyright IBM Corporation 2008. All Rights Reserved.

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided “AS IS”, without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

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