IBM® DB2® for Linux®, UNIX®, and Windows®

Best practices

Tuning and monitoring database system performance

Steve Rees
Senior Technical Staff Member
DB2 Performance

Thomas Rech
Senior Consultant
DB2 SAP Center of Excellence

Olaf Depper
Principal
Information Management SAP Ecosystem

Naveen K Singh
Senior S/W Engineer
DB2 Monitor Infrastructure

Gang Shen
Executive I/T Specialist
IBM Information Management

Roman B. Melnyk
Senior Writer
DB2 Information Development

Last updated: July 2013

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 IBM Smart Analytics System, or IBM PureData System for Operational Analytics, or those within SAP systems, have configurations that are tightly specified. In the IBM PureData case, hardware factors, such as the number of CPUs, the ratio of memory to CPU, the number and configuration of disks, as well as software 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 generic 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, 1.5 terabyte (TB) 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 your best estimate 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 and type of disks involved: 'spinning' or hard-disk drives (HDDs), solid-state disks (SSDs), flash drives, etc…

Why is that the case? The evolution of CPUs over the past decade has seen incredible increases in speed, whereas the evolution of disk drives (apart from the most modern flash drives or SSDs) has been more in terms of increased capacity and reduced cost. There have been improvements in seek time and transfer rate for spinning disks, 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.

For high performance applications, SSDs or flash drives can be excellent options. Because they have no moving parts, they are able to process both read and write operations extraordinarily quickly, when compared to HDDs. Random reads and writes, in particular, can be up to 200x faster, and even sequential scans can proceed two to three times more quickly. Because of higher cost, SSDs are often reserved for smaller and more performance-sensitive areas of the database, such as temporary table space storage in a data warehouse. However, the price of SSDs and flash drives is dropping quickly, making this an increasingly realistic option for many customers. You should definitely consider them for performance-sensitive IO-bound workloads.

In the case of RAID arrays of HDDs, a rule-of-thumb is to configure at least ten to twenty disks per processor core. For SAN 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 the best practices "Database Storage" (https://ibm.biz/Bdx2My) and "Physical Database Design" (https://ibm.biz/Bdx2nr) for more information on storage configuration best practices.

It is good practice to set aside some dedicated (unshared) disk for the DB2 transaction logs, especially in transactional systems. This is because the I/O characteristics of the logs are very different from other consumers such as DB2 containers, 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 (HDDs) can provide enough logging throughput for up to 500 reasonably write-intensive DB2 transactions per second. Greater throughput rates, or high-volume logging (for example, during bulk inserts or ETL processing), 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 HDDs 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 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 6.1 or later. Again, if there are specific settings already in place for your system (for example, an IBM PureData system 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 and later.

- The AIO parameter `maxservers` can be initially left at the default value of ten per CPU. This parameter controls the number of asynchronous IO kprocs or threads that AIX creates per processor. 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 experiments have resulted in a recommended value that 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.

- Use scalable volume groups to avoid offset data blocks and allow for large PV sizes and numbers. Do not mirror or stripe using the AIX LVM as this will potentially conflict with storage subsystem striping and can result in the storage subsystem I/O becoming more random, in which case it could confuse the caching algorithms in the storage subsystem and result in reduced performance.

Solaris and HP-UX configuration

Setting sufficient kernel settings on HP-UX and Solaris is important for both the stability and performance of DB2 on these operating systems. For DB2 running on Solaris or HP-UX, the `db2osconf` utility is available to check and recommend kernel parameters based on the system size. You can use `db2osconf` 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.

In HP-UX environments, in addition to the results provided by running `db2osconf`, you might also want to consider tuning the following:

- Changing the internal attributes of the `db2sysc` executable using the `chatr` command: `chatr +pd 256M +pi 256M db2sysc`. This will increase the virtual memory page sizes from the default 1 MB to 256 MB.

- A larger-than-default base page size allows the operating system to manage memory more efficiently, especially on systems with large RAM. This is achieved by executing `kctune base_pagesize = 16.`

- HP-UX supports a modified scheduling policy, known as SCHED_NOAGE. This will prevent the increase or decrease of a process priority. Changing the default scheduling policy is especially useful in OLTP environments. It is achieved by starting DB2 with `/usr/bin/rtsched –s SCHED_NOAGE –p 178 db2start`

Linux configuration

When a Linux system is used as a DB2 server, the DB2 database manager will automatically enforce certain minimum kernel settings for memory management (like SHMMAX or SHMALL) and for IPC communication (like the number and size of semaphores).

However, in larger and more complex environments some additional kernel parameters might have to be changed for optimal performance. 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. For a complete list of validated DB2 Linux environments refer to this link http://www.ibm.com/software/data/db2/linux/validate/.

Some important Linux tunables are:

- To influence the Linux kernel in swap decisions in a way that is appropriate for a DB2 server, set `vm.swappiness` to 0 (default: 60) Note that in environments where an application server like SAP shares the host with DB2, setting `vm.swappiness:0` might cause performance issues with the application. In such cases, a value of 10 is often a reasonable compromise.

- To influence the management of page cache on the Linux server, tuning the parameters `vm.dirty_ratio` and `vm.dirty_background_ratio` is important. A good starting point is usually to set `vm.dirty_ratio:10` and `vm.dirty_background_ratio:5.`

- On SUSE Linux Enterprise Server, it is recommend to disable barrier at mount time for ext3 and ext4 file systems by using the mount option `barrier=0.` This is the default setting on other Linux distributions.

The Linux operating system supports three I/O schedulers (NOOP, Deadline and CFQ). While the default CFQ scheduler is usually a good choice, sometimes a performance benefit can be seen by using the Deadline scheduler for file systems containing the DB2 table space containers. This is due to the fact that the Deadline scheduler favors reads versus write operations – and typically databases see more read requests than write requests. In general, it requires some experimentation to confirm the benefit of this change.

Tuning the Linux network kernel parameters is especially important in multi-tier environments where many application servers connect to the database. In these configurations, you will typically want to tune the send buffer and receive buffer sizes as well as the minimum, initial and maximum size, in /etc/sysctl.conf:

- net.core.wmem_max = 268435456.

- net.core.rmem_max = 268435456

- net.ipv4.tcp_wmem = 4096 1048576 268435456

- net.ipv4.tcp_rmem = 4096 1048576 268435456

Partitioned database environments

The decision to use a hash-partitioned database environment is not generally made based purely on data volume, but more on the nature of the workload. As a general guideline, most partitioned database deployments are in the area of data warehousing and business intelligence. Partitioned databases are highly recommended for large complex query environments, because the shared-nothing architecture allows for outstanding scalability. For smaller data marts (up to about 1 TB), which are unlikely to grow rapidly, a DB2 Enterprise Server Edition (ESE) configuration is often a good choice, potentially with intra-query parallelism (INTRA_PARALLEL) enabled. However, large or fast-growing BI environments benefit greatly from partitioned databases.

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 1.5TB of active raw data per partition.

The IBM PureData for Operational Analytics documentation (http://www.ibm.com/software/data/puredata/operationalanalytics/) contains in-depth information regarding partitioned database configuration best practices. This documentation contains useful information for custom deployments as well.

Choice of code page and collation

As well as affecting database behavior, the 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 DB2 databases. 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%.

In addition to 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 CLDR181_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, CLDR181_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, CLDR181_NO) have additional checks to handle malformed characters, whereas non-normalized collations (for example, CLDR181_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, language-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 the best practices papers Physical database design for OLTP environments (https://ibm.biz/Bdx2nr) and Physical database design for data warehouse environments (https://ibm.biz/Bdx2np) but for our purposes, we address a couple of the top-level best practices here.

- In general, automatic storage (AS) and 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 noticeable performance advantage over AS and 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), AS and DMS file table spaces provide virtually the same performance as DMS raw device table spaces.

- If the size of a row happens to be just over half the page size of the table space it uses, this will leave almost half the page empty. While this can be intentional sometimes (avoiding page-level contention for a particularly hot table), generally that much waste is not desirable, and it would more efficient to store the table in a larger page size table space.

Initial DB2 configuration settings

The DB2 configuration advisor (http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/index.jsp?topic=%2Fcom.ibm.db2.luw.admin.cmd.doc%2Fdoc%2Fr0008960.html), 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 the **AUTOCONFIGURE** command is run automatically at database creation time since DB2 v9.1, 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). Significantly 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 autonomics 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 most 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 shared 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, from a performance perspective, 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. Where possible, transaction logs should ideally 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 256 pages is too small for good performance in many production environments. The configuration advisor always increases it, but possibly not enough, depending on the input parameters. A value of around 1024 pages is a good general range, and represents only a very small total amount of memory in the overall scheme of a database server.

- **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.

- **logfilsiz** determines the size of each transaction log file on disk. The log files should be sized so that log switches do not occur more frequently than every few minutes at their fastest. The rate of change of active log numbers can be obtained by querying MON_GET_TRANSACTION_LOG. If a larger logfilsiz causes an unacceptably long time between log switches during quieter times, you may choose to schedule some manual switches to trigger archiving.

- **diagpath** determines the location of various useful DB2 diagnostic files. It generally has little impact on performance, except possibly in partitioned or clustered database environments. The default location of **diagpath** on all partitions is typically on a shared, NFS or GPFS-mounted path. The best practice is to override **diagpath** to a local, non-shared 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 typical for DB2 systems to use storage consisting of arrays of disks (which are presented to the operating system by the storage controller as a single device) or to use file systems that span multiple devices. The consequence is that by default, a non-pureScale 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 six – 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. Note that on DB2 pureScale systems, DB2_PARALLEL_IO defaults to '*', whereas on non-pureScale configurations it defaults to off. See "Database Storage" (https://ibm.biz/Bdx2My) 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" (https://ibm.biz/Bdx2ng).

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 has to be set to the value `SAP` in SAP environments. This will enable aggregated registry variables that are optimized for SAP workloads.

In SAP environments, the database configuration has to follow SAP standards, for example in regards of database code page, table space page and extent size as well as naming conventions. During the installation of a SAP NetWeaver system an initial set of DB2 configuration parameters is applied. In addition, SAP Notes describe the preferred DB2 parameter settings for each supported DB2 version. For example, recommended minimum parameters settings for SAP NetWeaver systems based on DB2 10.1 are described in, SAP Note 1692571 ("DB6: DB2 10.1 Standard Parameter Settings"). In addition to configuration recommendations, you will also find best practices regarding the administration of DB2 databases in various SAP notes.

SAP environments offer a powerful monitoring and administration platform in transaction "DBACockpit". In newer SAP releases, the DBACockpit is based on DB2 in-memory monitoring functions described in this paper.

Pay special attention to SAP applications when using partitioned databases. SAP uses partitioned databases mainly in their SAP NetWeaver Business Warehouse (BW) product. The recommended layout has the DB2 system catalog, the dimension and master tables, plus the SAP base tables on Partition 0. All SAP application servers connect to partition 0. This leads to a different workload on this partition compared to the other partitions in the partitioned cluster. Because of the additional workload, up to eight processors might be assigned to just partition 0. As the SAP BW workload becomes more highly parallelized, with many short queries running concurrently, the number of partitions for SAP BW 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 Community Network (http://scn.sap.com/community/db2-for-linux-unix-windows).

Other recommendations and best practices might apply for other non-SAP ISV environments, 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.

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 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 mechanisms as of DB2 Version 9.7 are monitor table functions. These represent a significant improvement over the snapshot infrastructure of previous versions, providing not only additional monitoring data, but doing so with reduced overhead. The table functions focus on summary data, where 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, most monitor table functions 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. The DB2 system 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 activity event monitors.) These tools provide a much more comprehensive, detailed recording of system activity than the summary you get from monitor table functions. However, traces produce a large amount of data and impose a greater overhead on the system. Consequently, they are more suitable for exception monitoring than for operational monitoring. That said, as with monitor table functions, the latest generation of activity event monitors impose a much lower overhead on the system compared to statement event monitors of previous versions.

There is no reason to limit yourself to just metrics regarding the DB2 engine itself. 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. Fortunately, DB2 now provides key operating system information via table functions such as ENV_GET_SYSTEM_RESOURCES. 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 database engine 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.

As mentioned above, recent releases of the DB2 database product have shifted the emphasis in monitoring from textual snapshot output to SQL interfaces such as the monitoring table functions. As well as introducing new metrics and reducing overhead, this also makes management of monitoring data with DB2 very straightforward, because you can easily redirect the data from the table functions and administration views right back into DB2 tables. For deeper dives, event and activity monitor data can also be written to DB2 tables, providing similar benefits.

Delta values and DB2 monitor table functions

If you are used to DB2 snapshots from previous releases, you are probably familiar with the concept of the `RESET MONITOR` command. This command allowed you to reset the internal monitor elements to zero, so that the next time you captured a snapshot on that connection, you would have values for *just* the interval since the reset. This is important, because the monitored interval might have very different activity from the total period since database activation. In particular, the longer a database is active, the more the monitor element values trend toward the true average behavior, and 'damp out' the peaks and valleys we might be interested in.

In order to maximize efficiency, the monitor table function values cannot be reset. Instead, we will use SQL to calculate 'delta values' for the period we're interested in. This means that we will first capture a sample of monitor data at the beginning of the interval, wait some time and capture again at the end, and then just find the difference between these two collections to determine the activity during the interval. Fortunately, this is a pretty straightforward process in SQL

In the following example, we will find the delta values for data logical and physical bufferpool reads.

1. First, we create a place to keep our baseline data (collected at the beginning of the interval we're interested in.) We will use a declared global temporary table (DGTT), to give somewhere temporary (but persistent for the database connection) where we can keep our baseline, and that gives us isolation from other users. Of course, if we wanted to share our baseline across multiple connections, we could just use a regular table. Note that a user temporary table space needs to exist before the baseline DGTTs can be created. Fortunately, it can be quite small, since the baseline DGTTs will only contain a single sample from the table functions.

```
declare global temporary table
  mgb_baseline
as
  (select *
```

```
    from table(mon_get_bufferpool(null,-2)))
  with no data
  on commit preserve rows;
```

2. Once we have the temporary table, we can collect our baseline data. We want to make sure there's just one collection in the table at a time, so we delete any previous data first.

```
  delete from session.mgb_baseline;

  insert
    into session.mgb_baseline
    select *
    from table(mon_get_bufferpool(null,-2));
```

3. Now we have a baseline for our data from `mon_get_bufferpool`. We wait a bit of time, to allow activity in our interval to occur and be recorded by internal DB2 monitor counters.

4. Once we reach the end of the desired interval, we query the desired elements from `mon_get_bufferpool`, *and subtract off the corresponding baseline values*. This gives us the activity that's happened only during the interval we are interested in.

```
-- Set up a common table expression ('with clause')
-- to calculate deltas for each table function we reference

with mgb_delta
  ( MEMBER,
    BP_NAME,
    POOL_DATA_L_READS,
    POOL_DATA_P_READS )
as (
  select
    mgb.MEMBER,
    substr(mgb.BP_NAME,1,20) as BP_NAME,
    mgb.POOL_DATA_L_READS - mgb_baseline.POOL_DATA_L_READS,
    mgb.POOL_DATA_P_READS - mgb_baseline.POOL_DATA_P_READS
  from
    table(MON_GET_BUFFERPOOL(null,-2)) as mgb,
    session.mgb_baseline as mgb_baseline
  where
    mgb.MEMBER  = mgb_baseline.MEMBER and
    mgb.BP_NAME = mgb_baseline.BP_NAME
)

-- Then pick out the values we need from our
-- 'mgb_delta' common table expression

select
  MEMBER,
  BP_NAME,
  POOL_DATA_L_READS,
```

```
  POOL_DATA_P_READS
from mgb_delta;
```

A few notes about the above example, since we're going to follow this pattern below:

- These queries may look a little 'bulky', but the process is exactly the same in all of them: define one or more common table expressions to find the delta values for the table functions we need (just straightforward subtraction), and then follow this with the actual query to return the desired values and/or calculations. They are very well suited to 'cut and paste'.

- The common table expression is tailored to just find the delta values we're interested in here, but it could be done once to find a superset of columns that any one query needs (maybe all the columns in the table function), and then reused. In that case, it would probably be set up as a view instead.

- The BP_NAME column is important here, since it's the 'primary key' of the output of mon_get_bufferpool. We need to make sure we have a 1:1 mapping from the rows in the baseline to the rows returned from the closing or 'final' call to mon_get_bufferpool. Different table functions have different key columns.

- Each time you execute this query, it calls mon_get_bufferpool again, and each time it does that, it might get a slightly different set of data for the final sample. If there were multiple queries to be done and you wanted them all to see exactly the same data, then the data for the closing call to mon_get_bufferpool could be collected once and stored in a table, and used multiple times in place of the call to mon_get_bufferpool in the query.

Easy & powerful monitoring of DB2 performance with Optim Performance Manager (OPM)

DB2 provides a comprehensive and powerful set of performance monitoring metrics, accessible through snapshots, table functions, administrative views and event monitors. While these interfaces together provide a complete view of DB2 performance, it can sometimes be challenging to pull all the relevant data together on an ongoing basis, particularly if many DB2 databases or instances need to be monitored.

This is where InfoSphere® Optim™ Performance Manager (OPM) can have a significant impact. OPM exploits DB2's native monitoring interfaces to bring together the most important performance indicators in an organized, powerful and easy-to-use way. This greatly speeds the process of establishing a monitoring regimen and getting down to the core performance issues in the system. OPM not only builds on DB2's monitoring interfaces to collect & visualize real-time performance data, it also provides an historical

performance repository to store and analyze metrics over time. This means that OPM can help you "go back in time" to study the system's current behavior in comparison with how it was at prior times of good, poor or average performance. This is an extremely powerful feature. When combined with OPM's ability to alert you when monitored metrics reach warning or critical levels, its powerful investigation workflows help you to efficiently get to root cause. With the ability of the Extended Insight feature to measure true end-to-end performance, the overall OPM package provides significant additional value over monitoring with native DB2 interfaces.

In the following section, sample queries for collecting key performance metrics directly from DB2 are accompanied by guidance on finding the same information in OPM.

A good 'starter set' of DB2 performance queries

Most of the metrics that these queries collect can come from the database snapshot (**GET SNAPSHOT FOR DATABASE** command). However, as mentioned earlier, the best practice is to access these elements through the appropriate SQL interfaces (MON_GET_WORKLOAD, MON_GET_BUFFERPOOL, etc.), which provide access to the most current set of monitoring metrics, and make analysis and long-term management simpler.

The database configuration parameters affecting monitoring (MON_REQ_METRICS, MON_ACT_METRICS and MON_OBJ_METRICS) must be at BASE (the default) or EXTENDED if the monitor table functions are to provide the data we need. Fortunately, these switches are dynamic and do not require a database reactivation to turn them on or off. In the examples below, we highlight the table function columns that are used to determine each metric.

A note about partitioned database environments and DB2 pureScale environments: if you are running in a multi-partition environment, you should include MEMBER 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:

```
-- DB2 10.5 can use mon_get_workload

with
mgw_delta (
  MEMBER,
  SELECT_SQL_STMTS,
  UID_SQL_STMTS,
  TOTAL_APP_COMMITS )
as (
  select
    mgw.MEMBER,
    sum( mgw.SELECT_SQL_STMTS
       - mgw_baseline.SELECT_SQL_STMTS),
    sum( mgw.UID_SQL_STMTS
```

```
       - mgw_baseline.UID_SQL_STMTS),
    sum( mgw.TOTAL_APP_COMMITS
       - mgw_baseline.TOTAL_APP_COMMITS)
  from
    table(MON_GET_WORKLOAD(null,-2)) as mgw,
    session.mgw_baseline as mgw_baseline
  where
    mgw.MEMBER      = mgw_baseline.MEMBER and
    mgw.WORKLOAD_ID = mgw_baseline.WORKLOAD_ID
  group by mgw.MEMBER )

select
  MEMBER,
  SELECT_SQL_STMTS,
  UID_SQL_STMTS,
  TOTAL_APP_COMMITS
from mgw_delta

-- On earlier versions of DB2, we use SNAPDB
with
sdb_delta (
  SELECT_SQL_STMTS,
  UID_SQL_STMTS,
  COMMIT_SQL_STMTS )
as (
  select
    sum( sdb.SELECT_SQL_STMTS
       - sdb_baseline.SELECT_SQL_STMTS),
    sum( sdb.UID_SQL_STMTS
       - sdb_baseline.UID_SQL_STMTS),
    sum( sdb.COMMIT_SQL_STMTS
       - sdb_baseline.COMMIT_SQL_STMTS)
  from
    sysibmadm.snapdb as sdb,
    session.sdb_baseline as sdb_baseline )

select
  SELECT_SQL_STMTS,
  UID_SQL_STMTS,
  COMMIT_SQL_STMTS
from sdb_delta;
```

OPM:
Workload Dashboard -> Throughput pane
Overview Dashboard -> Data Server Throughput pane

These provide an excellent base level measurement of system activity.

Note that `sum()` is used around columns from MON_GET_WORKLOAD because this table function returns one row for each workload defined in the system. Even if Workload Management (WLM) isn't in use, at least two rows will be returned. Of course, the `sum()` can be excluded, if WLM is in use, and workload-level data is desired. In such a case, though, an extra predicate would

need to be added to the join, linking WORKLOAD_NAME on mgw with mgw_baseline.

2. Buffer pool hit ratios, measured separately for data and index activity. This is obtained from MON_GET_BUFERPOOL

```
with mgb_delta
( MEMBER,
  BP_NAME,
  POOL_DATA_L_READS,
  POOL_DATA_P_READS,
  POOL_TEMP_DATA_L_READS,
  POOL_TEMP_DATA_P_READS,
  POOL_ASYNC_DATA_READS,
  POOL_INDEX_L_READS,
  POOL_INDEX_P_READS,
  POOL_TEMP_INDEX_L_READS,
  POOL_TEMP_INDEX_P_READS,
  POOL_ASYNC_INDEX_READS )
as ( select
  mgb.MEMBER,
  mgb.BP_NAME,
  mgb.POOL_DATA_L_READS
            - mgb_base.POOL_DATA_L_READS,
  mgb.POOL_DATA_P_READS
            - mgb_base.POOL_DATA_P_READS,
  mgb.POOL_TEMP_DATA_L_READS
            - mgb_base.POOL_TEMP_DATA_L_READS,
  mgb.POOL_TEMP_DATA_P_READS
            - mgb_base.POOL_TEMP_DATA_P_READS,
  mgb.POOL_ASYNC_DATA_READS
            - mgb_base.POOL_ASYNC_DATA_READS,
  mgb.POOL_INDEX_L_READS
            - mgb_base.POOL_INDEX_L_READS,
  mgb.POOL_INDEX_P_READS
            - mgb_base.POOL_INDEX_P_READS,
  mgb.POOL_TEMP_INDEX_L_READS
            - mgb_base.POOL_TEMP_INDEX_L_READS,
  mgb.POOL_TEMP_INDEX_P_READS
            - mgb_base.POOL_TEMP_INDEX_P_READS,
  mgb.POOL_ASYNC_INDEX_READS
            - mgb_base.POOL_ASYNC_INDEX_READS
  from
    table(MON_GET_BUFFERPOOL(null,-2)) as mgb,
    session.mgb_baseline as mgb_base
  where
    mgb.member  = mgb_base.member and
    mgb.bp_name = mgb_base.bp_name )

select MEMBER,
 case
   when sum(b.POOL_DATA_L_READS + b.POOL_TEMP_DATA_L_READS)
     < 1000 then null else
       100 * sum(b.POOL_DATA_L_READS
               + b.POOL_TEMP_DATA_L_READS
```

```
              - (b.POOL_DATA_P_READS
               + b.POOL_TEMP_DATA_P_READS
               - b.POOL_ASYNC_DATA_READS))
        / decimal(sum(b.POOL_DATA_L_READS
                    + b.POOL_TEMP_DATA_L_READS))
   end as DATA_BP_HR,
 case
   when sum(b.POOL_INDEX_L_READS +
            b.POOL_TEMP_INDEX_L_READS)
     < 1000 then null else
       100 * sum(b.POOL_INDEX_L_READS
               + b.POOL_TEMP_INDEX_L_READS
             - (b.POOL_INDEX_P_READS
              + b.POOL_TEMP_INDEX_P_READS
              - b.POOL_ASYNC_INDEX_READS))
         / decimal(sum(b.POOL_INDEX_L_READS
                     + b.POOL_TEMP_INDEX_L_READS))
   end as INDEX_BP_HR
from mgb_delta as b

group by MEMBER
```

OPM:
Buffer Pool and I/O Dashboard

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 MON_GET_BUFFERPOOL table function.

Why the CASE clause? The first reason is that this avoids a pesky divide-by-zero in the case when there has been no activity on a particular buffer pool. The second reason – really, the reason why we use "< 1000" and not "= 0" – is that for buffer pools with very low levels of activity, we don't really care what the hit ratio is. For example, one logical read and one physical read is NOT grounds to panic over a 0% hit ratio!

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. In these cases, it's best to focus on temporary data and temporary index hit ratios, as they offer the best opportunity for performance improvements in data warehouse applications.

3. Buffer pool physical reads and writes per transaction (here we get the metrics we need from two different table functions):

```
with mgb_delta
( MEMBER,
  POOL_DATA_P_READS,
```

```
  POOL_TEMP_DATA_P_READS,
  POOL_INDEX_P_READS,
  POOL_TEMP_INDEX_P_READS,
  POOL_DATA_WRITES,
  POOL_INDEX_WRITES )
as (
  select
    mgb.MEMBER,
    sum( mgb.POOL_DATA_P_READS
       - mgb_baseline.POOL_DATA_P_READS ),
    sum( mgb.POOL_TEMP_DATA_P_READS
       - mgb_baseline.POOL_TEMP_DATA_P_READS ),
    sum( mgb.POOL_INDEX_P_READS
       - mgb_baseline.POOL_INDEX_P_READS ),
    sum( mgb.POOL_TEMP_INDEX_P_READS
       - mgb_baseline.POOL_TEMP_INDEX_P_READS ),
    sum( mgb.POOL_DATA_WRITES
       - mgb_baseline.POOL_DATA_WRITES ),
    sum( mgb.POOL_INDEX_WRITES
       - mgb_baseline.POOL_INDEX_WRITES )
  from
    table(MON_GET_BUFFERPOOL(null,-2)) as mgb,
    session.mgb_baseline as mgb_baseline
  where
    mgb.MEMBER  = mgb_baseline.MEMBER and
    mgb.BP_NAME = mgb_baseline.BP_NAME
  group by mgb.MEMBER ),

mgw_delta (
  MEMBER,
  TOTAL_APP_COMMITS )
as (
  select
    mgw.MEMBER,
    sum( mgw.TOTAL_APP_COMMITS
       - mgw_baseline.TOTAL_APP_COMMITS )
  from
    table(MON_GET_WORKLOAD(null,-2)) as mgw,
    session.mgw_baseline as mgw_baseline
  where
    mgw.MEMBER      = mgw_baseline.MEMBER and
    mgw.WORKLOAD_ID = mgw_baseline.WORKLOAD_ID
  group by mgw.MEMBER)

select b.MEMBER,
  case when w.TOTAL_APP_COMMITS < 1000 then null
  else ( b.POOL_DATA_P_READS
       + b.POOL_INDEX_P_READS
       + b.POOL_TEMP_DATA_P_READS
       + b.POOL_TEMP_INDEX_P_READS )
    / decimal(w.TOTAL_APP_COMMITS) end
    as BP_PHYS_RD_PER_TX,

  case when w.TOTAL_APP_COMMITS < 1000 then NULL
  else (b.POOL_DATA_WRITES
      + b.POOL_INDEX_WRITES)
```

```
    / decimal(w.TOTAL_APP_COMMITS) end
    as BP_PHYS_WR_PER_TX

from mgb_delta as b,
     mgw_delta as w
where b.MEMBER = w.MEMBER
```

OPM:
Overview Dashboard -> I/O and Disk Space pane

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 not really any sensible 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, or per minute in OPM. 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 returned:

```
with
mgw_delta (
  MEMBER,
  ROWS_READ,
  ROWS_RETURNED )
as (
  select
    mgw.MEMBER,
    sum( mgw.ROWS_READ
       - mgw_baseline.ROWS_READ ),
    sum( mgw.ROWS_RETURNED
       - mgw_baseline.ROWS_RETURNED )
  from
    table(MON_GET_WORKLOAD(null,-2)) as mgw,
    session.mgw_baseline as mgw_baseline
```

```
  where
    mgw.MEMBER      = mgw_baseline.MEMBER and
    mgw.WORKLOAD_ID = mgw_baseline.WORKLOAD_ID
  group by mgw.MEMBER )

select w.MEMBER, case
  when ROWS_RETURNED < 1000 then null
  else ROWS_READ / decimal(ROWS_RETURNED) end
  as ROWS_READ_PER_ROWS_RET
from mgw_delta as w
```

OPM:
Overview Dashboard -> Workload pane

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 (possibly due to indexes not being available, or old statistics), 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 often 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:

```
with
mgw_delta (
  MEMBER,
  TOTAL_APP_COMMITS,
  TOTAL_SECTION_SORT_TIME )
as (
  select
    mgw.MEMBER,
    sum( mgw.TOTAL_APP_COMMITS
       - mgw_baseline.TOTAL_APP_COMMITS ),
    sum( mgw.TOTAL_SECTION_SORT_TIME
       - mgw_baseline.TOTAL_SECTION_SORT_TIME )
  from
    table(MON_GET_WORKLOAD(null,-2)) as mgw,
    session.mgw_baseline as mgw_baseline
  where
    mgw.MEMBER      = mgw_baseline.MEMBER and
    mgw.WORKLOAD_ID = mgw_baseline.WORKLOAD_ID
  group by mgw.MEMBER )

select w.MEMBER, case
```

```
  when w.TOTAL_APP_COMMITS < 1000 then null else
  w.TOTAL_SECTION_SORT_TIME /
      decimal(w.TOTAL_APP_COMMITS) end
  as SORT_TIME_PER_TX
  from mgw_delta as w

OPM:
 Workload Dashboard -> Sorting pane
```

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:

```
with
mgw_delta (
  MEMBER,
  TOTAL_APP_COMMITS,
  LOCK_WAIT_TIME,
  LOCK_ESCALS )
as (
  select
    mgw.MEMBER,
    sum( mgw.TOTAL_APP_COMMITS
       - mgw_baseline.TOTAL_APP_COMMITS ),
    sum( mgw.LOCK_WAIT_TIME
       - mgw_baseline.LOCK_WAIT_TIME ),
    sum( mgw.LOCK_ESCALS
       - mgw_baseline.LOCK_ESCALS )
  from
    table(MON_GET_WORKLOAD(null,-2)) as mgw,
    session.mgw_baseline as mgw_baseline
  where
    mgw.MEMBER      = mgw_baseline.MEMBER and
    mgw.WORKLOAD_ID = mgw_baseline.WORKLOAD_ID
  group by mgw.MEMBER )

select  w.MEMBER,
  case when w.TOTAL_APP_COMMITS < 1000 then null else
  1000 * w.LOCK_WAIT_TIME
    / decimal(w.TOTAL_APP_COMMITS)  end
    as LOCK_WAIT_TIME_PER_1000_TX,
  case when w.TOTAL_APP_COMMITS < 1000 then null else
  1000 * w.LOCK_ESCALS
    / decimal(w.TOTAL_APP_COMMITS)end
    as LOCK_ESCALS_PER_1000_TX
from mgw_delta as w

OPM:
 Overview Dashboard -> Locking pane
```

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:

```
with
mgw_delta (
  MEMBER,
  DEADLOCKS,
  LOCK_TIMEOUTS,
  TOTAL_APP_COMMITS )
as (
  select
    mgw.MEMBER,
    sum( mgw.DEADLOCKS
       - mgw_baseline.DEADLOCKS ),
    sum( mgw.LOCK_TIMEOUTS
       - mgw_baseline.LOCK_TIMEOUTS ),
    sum( mgw.TOTAL_APP_COMMITS
       - mgw_baseline.TOTAL_APP_COMMITS )
  from
    table(MON_GET_WORKLOAD(null,-2)) as mgw,
    session.mgw_baseline as mgw_baseline
  where
    mgw.MEMBER      = mgw_baseline.MEMBER and
    mgw.WORKLOAD_ID = mgw_baseline.WORKLOAD_ID
  group by mgw.MEMBER )

select w.MEMBER, case
  when w.TOTAL_APP_COMMITS < 1000 then null else
  1000 * (w.DEADLOCKS + w.LOCK_TIMEOUTS)
         / decimal(w.TOTAL_APP_COMMITS)end
  as DL_AND_LOCK_TMO_PER_1000_TX
from mgw_delta as w
```

OPM:
Overview Dashboard -> Locking pane

Although deadlocks are comparatively rare in most production systems, lock timeouts can be more common. (Note that the default lock timeout value is -1 – that is, infinite – so lock timeouts will only occur if this has been explicitly changed.) 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:

```
with
mgw_delta (
  MEMBER,
```

```
  TOTAL_APP_COMMITS )
as (
  select
    mgw.MEMBER,
    mgw.TOTAL_APP_COMMITS - mgw_baseline.TOTAL_APP_COMMITS
  from
    table(MON_GET_WORKLOAD(null,-2)) as mgw,
    session.mgw_baseline as mgw_baseline
  where
    mgw.MEMBER      = mgw_baseline.MEMBER and
    mgw.WORKLOAD_ID = mgw_baseline.WORKLOAD_ID ),

mgb_delta (
  MEMBER,
  POOL_DRTY_PG_STEAL_CLNS )
as (
  select
    mgb.MEMBER,
    mgb.POOL_DRTY_PG_STEAL_CLNS
               - mgb_baseline.POOL_DRTY_PG_STEAL_CLNS
  from
    table(MON_GET_BUFFERPOOL(null,-2)) as mgb,
    session.mgb_baseline as mgb_baseline
  where
    mgb.MEMBER  = mgb_baseline.MEMBER and
    mgb.BP_NAME = mgb_baseline.BP_NAME )

select w.MEMBER,
  case when sum(w.TOTAL_APP_COMMITS) < 1000 then null else
  1000 * sum(b.POOL_DRTY_PG_STEAL_CLNS)
         / decimal(sum(w.TOTAL_APP_COMMITS)) end as
  DRTY_STEAL_PER_1000_TX
from mgw_delta as w,
     mgb_delta as b
where w.MEMBER = b.MEMBER
group by w.MEMBER
```

OPM:
Buffer Pool and I/O Dashboard -> Prefetch and Page Cleaning tab

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. Note that dirty steals are currently only reported if 'classic' page cleaning is used. If alternate page cleaning (see below) is in use, dirty steal counts are always zero.

9. The number of package cache inserts per thousand transactions, and the percent of time spent compiling SQL:

```
with
mgw_delta (
  MEMBER,
```

```
  PKG_CACHE_INSERTS,
  TOTAL_COMPILE_TIME,
  TOTAL_RQST_TIME,
  TOTAL_APP_COMMITS )
as (
  select
    mgw.MEMBER,
    sum( mgw.PKG_CACHE_INSERTS
           - mgw_baseline.PKG_CACHE_INSERTS ),
    sum( mgw.TOTAL_COMPILE_TIME
           - mgw_baseline.TOTAL_COMPILE_TIME ),
    sum( mgw.TOTAL_RQST_TIME
           - mgw_baseline.TOTAL_RQST_TIME ),
    sum( mgw.TOTAL_APP_COMMITS
           - mgw_baseline.TOTAL_APP_COMMITS )
  from
    table(MON_GET_WORKLOAD(null,-2)) as mgw,
    session.mgw_baseline as mgw_baseline
  where
    mgw.MEMBER      = mgw_baseline.MEMBER and
    mgw.WORKLOAD_ID = mgw_baseline.WORKLOAD_ID
  group by mgw.MEMBER )

select w.MEMBER,
  case when w.TOTAL_APP_COMMITS < 1000 then null else
  1000 * w.PKG_CACHE_INSERTS
    / decimal(w.TOTAL_APP_COMMITS) end
    as PKG_CACHE_INS_PER_1000_TX,
  case when w.TOTAL_RQST_TIME < 1000 then null else
  100.0 * (w.TOTAL_COMPILE_TIME
    / decimal(w.TOTAL_RQST_TIME)) end
    as PCT_COMPILE_TIME
from mgw_delta as w
```

OPM:
KPI Overview Dashboard -> Caching pane
Memory Dashboard -> Health Overview pane

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 and the percent of time spent in compiling SQL is very low, 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, these metrics are intended to watch for a third type of situation, one in which applications unintentionally cause package cache churn and high CPU consumption by not reusing prepared statements, or by not using parameter markers in their frequently executed SQL.

10. The amount of log activity per transaction, and the time per log write

```
with
mgw_delta (
  MEMBER,
  TOTAL_APP_COMMITS )
as (
  select
    mgw.MEMBER,
    sum( mgw.TOTAL_APP_COMMITS
       - mgw_baseline.TOTAL_APP_COMMITS )
  from
    table(MON_GET_WORKLOAD(null,-2)) as mgw,
    session.mgw_baseline as mgw_baseline
  where
    mgw.MEMBER      = mgw_baseline.MEMBER and
    mgw.WORKLOAD_ID = mgw_baseline.WORKLOAD_ID
  group by mgw.MEMBER ),

mgtl_delta (
  MEMBER,
  LOG_WRITES,
  LOG_WRITE_TIME )
as (
  select
    mgtl.MEMBER,
    mgtl.LOG_WRITES     - mgtl_baseline.LOG_WRITES,
    mgtl.LOG_WRITE_TIME - mgtl_baseline.LOG_WRITE_TIME
  from
    table(MON_GET_TRANSACTION_LOG(-2)) as mgtl,
    session.mgtl_baseline as mgtl_baseline
  where
    mgtl.MEMBER = mgtl_baseline.MEMBER )

select
  w.MEMBER,
  case when w.TOTAL_APP_COMMITS < 1000 then null else
    1000 * tl.LOG_WRITES
                / decimal(w.TOTAL_APP_COMMITS) end
    as LOG_WR_PER_1000_TX,
  case when w.TOTAL_APP_COMMITS < 1000 then null else
    1000 * tl.LOG_WRITE_TIME
                / decimal(w.TOTAL_APP_COMMITS) end
    as LOG_WR_TIME_PER_1000_TX,
  case when tl.LOG_WRITES < 1000 then null else
    tl.LOG_WRITE_TIME
                / decimal(tl.LOG_WRITES) end
    as TIME_PER_LOG_WR
from mgw_delta as w,
     mgtl_delta as tl
where w.MEMBER = tl.MEMBER
```

OPM:

Logging Dashboard

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).

The above query makes use of the MON_GET_TRANSACTION_LOG table function, which is available as of DB2 10. For earlier releases, you can use the following, which uses one of the older-style administrative views:

```
with
sdb_delta (
  COMMIT_SQL_STMTS,
  LOG_WRITE_TIME_S,
  LOG_WRITE_TIME_NS,
  LOG_WRITES )
as (
  select
    sdb.COMMIT_SQL_STMTS  - sdb_baseline.COMMIT_SQL_STMTS,
    sdb.LOG_WRITE_TIME_S  - sdb_baseline.LOG_WRITE_TIME_S,
    sdb.LOG_WRITE_TIME_NS - sdb_baseline.LOG_WRITE_TIME_NS,
    sdb.LOG_WRITES        - sdb_baseline.LOG_WRITES
  from
    SYSIBMADM.SNAPDB as sdb,
    session.sdb_baseline as sdb_baseline )

select
  case when sdb.COMMIT_SQL_STMTS < 1000 then null else
    1000 *
      sdb.LOG_WRITES / decimal(sdb.COMMIT_SQL_STMTS) end
    as LOG_WR_PER_1000_TX,
  case when sdb.COMMIT_SQL_STMTS < 1000 then null else
    1000 *
      (1000 * sdb.LOG_WRITE_TIME_S +
       sdb.LOG_WRITE_TIME_NS / 1000000.0)
      / decimal(sdb.COMMIT_SQL_STMTS) end
    as LOG_WR_TIME_PER_1000_TX,
  case when sdb.LOG_WRITES < 1000 then null else
       (1000 * sdb.LOG_WRITE_TIME_S +
        sdb.LOG_WRITE_TIME_NS / 1000000.0)
       / decimal(sdb.LOG_WRITES) end
    as TIME_PER_LOG_WR
  from sdb_delta as sdb
```

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

```
with
mgw_delta (
  MEMBER,
  FCM_SENDS_TOTAL,
  FCM_RECVS_TOTAL)
as (
  select
```

```
    mgw.MEMBER,
    sum(mgw.FCM_SENDS_TOTAL
                - mgw_baseline.FCM_SENDS_TOTAL),
    sum(mgw.FCM_RECVS_TOTAL
                - mgw_baseline.FCM_RECVS_TOTAL)
  from
    table(MON_GET_WORKLOAD(null,-2)) as mgw,
    session.mgw_baseline as mgw_baseline
  where
    mgw.MEMBER      = mgw_baseline.MEMBER and
    mgw.WORKLOAD_ID = mgw_baseline.WORKLOAD_ID
  group by mgw.MEMBER )

select
  MEMBER,
  FCM_SENDS_TOTAL,
  FCM_RECVS_TOTAL
from mgw_delta as w
```

OPM:
Connection Dashboard -> Locking and Communication tab

These numbers give the volume of flow of FCM buffers between different partitions in the partitioned instance, 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.

Because this query is specifically targeted for partitioned databases, we pass -2 into MON_GET_WORKLOAD, so that we get data for the entire instance. Then, we use SUM and GROUP BY to provide information on a per-partition basis.

A helpful short-cut: the MONREPORT module

The monitor table functions provide a wealth of information about system performance, and queries like those above are a very powerful way to keep track of what's going on. However, it's unlikely that many of them would be entered on-the-fly from the command line. They're definitely more suited to putting together into a script for ongoing monitoring. To get quick, 'human-readable' performance monitoring output, you may want to try MONREPORT.DBSUMMARY. It is a SQL stored procedure included with recent versions of DB2 that produces a text-based report with key metrics like bufferpool activity, and various wait and processing times.

```
call monreport.dbsummary(30)
```

MONREPORT.DBSUMMARY takes one parameter, which is the number of seconds over which to collect monitor data. In this way, it reports the all-important delta values, rather than totals since the database was activated.

The MONREPORT module contains other procedures, such as MONREPORT.PKGCACHE, MONREPORT.LOCKWAIT and

MONREPORT.CURRENTSQL, which provide basic drill-down performance information, and can be helpful if MONREPORT.DBSUMMARY indicates a problem.

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.

  As an alternative to storing copies of the database and database manager configurations, DB2 10 also offers the `CREATE EVENT MONITOR (change history)` statement. This creates an event monitor that tracks changes made in these areas.

- 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 monitoring data. As a result, the best practice is to regularly monitor system load with **vmstat** / **iostat** / **netstat (or an all-in-one tool like sar)** on UNIX-based systems[1], and **perfmon** on Windows. You can also use DB2 table functions, such as ENV_GET_SYSTEM_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 application 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). For example, in SAP environments, you would use transaction ST03N for this monitoring.

The above metrics represent a good core set of data to collect on an ongoing basis. The DB2 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

[1] For users on AIX and Linux, the nmon tool is a very convenient & integrated way to collect performance metrics.

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 MON_GET_BUFFERPOOL table function as above, with a simple additional GROUP BY clause.)

- SQL statement 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 MON_GET_PKG_CACHE_STMT table function.) This can be useful in identifying statements that are consuming the most resources.

- Connection and unit-of-work data (obtained from the MON_GET_CONNECTION and MON_GET_UNIT_OF_WORK table functions), which gives you data similar to what you get from the core database table functions used above (MON_GET_WORKLOAD, etc.), but broken down by connection. This provides very useful additional information when drill-down is required, helping us to understand which application might be causing a performance problem, as well as a useful breakdown of overall statistics over different applications.

If you are going to drill down and pick up extra monitoring data, it is simplest to just collect all elements provided by the table functions 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 or per-member basis, as shown in the samples above. The same applies for much of the non-DB2 performance data you can collect that report statistics for a single OS instance (albeit possibly spanning multiple logical DB2 partitions), such as `vmstat` and `iostat`. 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 table functions contains a MEMBER column, where appropriate, that helps you query monitoring data by, or compare data between, partitions or members.

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

- Logical and physical buffer pool reads for data and indexes
- 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(*) as ROWS,
  DBPARTITIONNUM(<partition key>) as PARTITION
from <partitioned table>
group by DBPARTITIONNUM( <partition key> )
```

The best practices paper Physical Database Design (https://ibm.biz/Bdx2nr) 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 occasionally 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 can be 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, network, 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 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, CPU or disk 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
4. 'Lazy system'

Disk bottlenecks

System Bottleneck > Disk Bottleneck?

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% tend to 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`.
- High read & write times reported by DB2. In fact, this is usually the most important symptom – even more so than how busy the disk is. After all, if the disk is busy, but DB2 is still seeing good I/O times, then we wouldn't be likely to see a big benefit from reducing the disk activity.

```
with mgtbsp_delta (
  MEMBER,
  TBSP_NAME,
  POOL_DATA_P_READS,  POOL_TEMP_DATA_P_READS,
  POOL_XDA_P_READS,   POOL_TEMP_XDA_P_READS,
  POOL_INDEX_P_READS, POOL_TEMP_INDEX_P_READS,
  POOL_READ_TIME,     POOL_DATA_WRITES,   POOL_XDA_WRITES,
  POOL_INDEX_WRITES,  POOL_WRITE_TIME,
  DIRECT_READ_REQS,   DIRECT_READ_TIME,
  DIRECT_WRITE_REQS,  DIRECT_WRITE_TIME )
as (
  select
    mgtbsp.MEMBER,
    mgtbsp.TBSP_NAME,
    sum(mgtbsp.POOL_DATA_P_READS
      - mgtbsp_baseline.POOL_DATA_P_READS),
    <insert sum & difference for remaining items>
    sum(mgtbsp.DIRECT_WRITE_TIME
      - mgtbsp_baseline.DIRECT_WRITE_TIME)
  from
    table(MON_GET_TABLESPACE(null,-2)) as mgtbsp,
    session.mgtbsp_baseline as mgtbsp_baseline
```

```
  where
    mgtbsp.MEMBER    = mgtbsp_baseline.MEMBER and
    mgtbsp.TBSP_NAME = mgtbsp_baseline.TBSP_NAME
  group by mgtbsp.MEMBER, mgtbsp.TBSP_NAME )

select
  m.MEMBER, m.TBSP_NAME,
  case when
     m.POOL_DATA_P_READS  + m.POOL_TEMP_DATA_P_READS
   + m.POOL_XDA_P_READS   + m.POOL_TEMP_XDA_P_READS
   + m.POOL_INDEX_P_READS + m.POOL_TEMP_INDEX_P_READS
    > 100
  then decimal(m.POOL_READ_TIME)
    / ( m.POOL_DATA_P_READS  + m.POOL_TEMP_DATA_P_READS
      + m.POOL_XDA_P_READS   + m.POOL_TEMP_XDA_P_READS
      + m.POOL_INDEX_P_READS + m.POOL_TEMP_INDEX_P_READS)
  else null end as MS_PER_POOL_READ,
  case when
    m.POOL_DATA_WRITES + m.POOL_XDA_WRITES
    + m.POOL_INDEX_WRITES > 100
  then decimal(m.POOL_WRITE_TIME)
    /(m.POOL_DATA_WRITES + m.POOL_XDA_WRITES
      + m.POOL_INDEX_WRITES)
  else null end as MS_PER_POOL_WRITE,
  case when
    m.DIRECT_READ_REQS > 100
  then decimal(m.DIRECT_READ_TIME) / m.DIRECT_READ_REQS
  else null end as MS_PER_DIRECT_READ_REQ,
  case when
    m.DIRECT_WRITE_REQS > 100
  then decimal(m.DIRECT_WRITE_TIME) / m.DIRECT_WRITE_REQS
  else null end as MS_PER_DIRECT_WRITE_REQ
from mgtbsp_delta as m
```

In the monitoring query above, we find the average time in milliseconds for buffer pool reads and writes, and for direct read and write requests. As a rule-of-thumb, we would look for buffer pool read times in the 5-10 ms range, and writes generally a bit faster, in the 2-5 ms range. Times significantly beyond these (or which are worse than typical for your system) would suggest a disk bottleneck that needs to be investigated.

Direct read and write requests are used for large objects (LOBs.) For LOBs of around 1MB or larger, the typical read and write times can be quite a bit longer than for buffer pool operations mentioned above, for example, in the 20-50 ms range. In such a case, a combination of long LOB write times and heavy disk utilization would suggest a bottleneck.

If at this point there appears to be a problem, either due to heavy buffer pool I/O or direct I/O, we should investigate further to identify the root cause. We might need to add more disks eventually, but first we should check with the storage administrator to determine if there are any errors which could have a performance impact (for example, rebuilding a RAID array due to a bad disk, or a failed storage adapter.) If there are no such issues which would explain the bottleneck, we move on to tuning the DB2 system, particularly focusing on how the DB2 system uses storage.

The system administrator can help map the name of one or more busy devices to the affected file system path or paths. From there, you can determine how the DB2 system uses the affected path or paths:

- Is the path used by one or more table space containers or database storage paths? This is determined by querying TBSP_NAME, TBSP_ID and CONTAINER_NAME from MON_GET_CONTAINER, and looking for the bottlenecked path or paths in CONTAINER_NAME. This may generate a list of candidate table spaces to examine, depending on the storage configuration.

- Is the path used for the transaction log? This is determined by examining the database configuration and looking for the bottlenecked path in 'Path to log files' and 'Mirror log path'.

- Is the path used for DB2 diagnostics? 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, but first, a few words about DB2 automatic storage.

When Automatic Storage (AS) is in use, the DB2 system will assume responsibility for provisioning storage for AS-enabled table spaces where AS is enabled, rather than the database administrator needing to do so. This feature reduces administration effort, as compared to database-managed storage table spaces, while maintaining the same performance and capacity benefits. Because all AS table spaces share the same filesystems and underlying devices, hot-spots are generally avoided, which is also good. However, this automation also means that it's a bit more difficult to categorize a table space disk bottleneck – is it arising from data, index or temporary data activity? – based only on what storage is affected, since there can be elements of all three in each location. Likewise, a bottleneck on one storage path may impact several table spaces. So, if a disk bottleneck occurs and AS is enabled, it may be necessary to look into at least the preliminary stages of each investigation type (data, or index, or temporary table space bottleneck?) over multiple table spaces, to get to the root cause.

System Bottleneck > Container or Storage Path Disk Bottleneck > Hot Data Container > Hot Table?

To determine what is causing the container storage path to be a bottleneck, we need to determine which tables are stored in the affected table space(s), and which ones are most active. But first, because the container path we're investigating may contain more than just regular data, we'll verify that there is a high level of data access activity present.

1. Does the table space(s) on this path show high levels of physical data reads or writes? As mentioned, even though this is a data table space container that is on this path, it could contain other things as well.

```
with <delta view mgtbsp_delta similar to above>
select m.MEMBER, m.TBSP_ID, m.TBSP_NAME,
       m.POOL_DATA_P_READS, m.POOL_DATA_WRITES
  from mgtbsp_delta as m
  where m.TBSP_ID
```

```
  in (<table space IDs of hot containers>)
order by m.POOL_DATA_P_READS + m.POOL_DATA_WRITES desc
```

2. Which tables are in these table spaces? Query SYSCAT.TABLES, matching TBSPACEIDs with TBSP_IDs from MON_GET_CONTAINER, as above.

3. Which tables are most active? Query MON_GET_TABLE, selecting ROWS_READ, ROWS_INSERTED, ROWS_UPDATED, and ROWS_DELETED for the table or tables from the table space(s) with the hot container path. Look for levels of activity that are much higher than other tables.

System Bottleneck > Container or Storage Path Disk Bottleneck > Hot Data Container > Hot Table > SQL stmt?

Drilling down further, we need to find out what is causing the high level of activity on this table. Are particular SQL statements causing high activity? Query the SQL package cache with MON_GET_PKG_CACHE_STMT , using the LIKE predicate on STMT_TEXT to identify statements that touch the table(s) in which we are interested:

```
select …
  from table(MON_GET_PKG_CACHE_STMT(NULL,NULL,NULL,-1)) as MGPCS
  where
    translate(cast(substr(STMT_TEXT, 1, 32672) as
                  varchar(32672))) like '%<hot table name>%'
    order by …
  fetch first … rows only
```

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_MODIFIED, ROWS_RETURNED, and NUM_EXECUTIONS to concentrate on those statements that are having the greatest impact on the table. We assume 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.

We include a FETCH FIRST clause, since if we have an ORDER BY clause included as well, we're only interested in the first range of statements. In general, 10 or 20 is probably enough.

Of course, this technique as shown isn't fool-proof (as shown here), since there could be views or aliases used in the SQL statement, that hide the true table name from us. In such a case, some extra steps to identify other ways that the table might be referred to might have to be taken before querying the SQL texts.

A much more powerful fool-proof way of finding what statements refer to particular tables (or indexes) is to use usage lists. These were introduced in DB2 10, and are described in the DB2 Information Center (http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.admin.dbobj.doc/doc/c0058647.html).

System Bottleneck > Container or Storage Path Disk Bottleneck > Hot Data Container > Hot Table > Hot

SQL 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`, or by comparing ROWS_READ with ROWS_RETURNED for the statement in question. Table scans are often a necessary part of 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" (https://ibm.biz/Bdx2ng) and "Physical Database Design" (https://ibm.biz/Bdx2nr).

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), a large portion of the 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, due to the less-efficient small reads the agent does. 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 or Storage Path 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.

1. Is there a high level of index read or write activity in this table space? Query MON_GET_TABLESPACE for the table space with TBSP_ID matching TBSP_ID of MON_GET_CONTAINER, from above.

```
with <delta view mgtbsp_delta similar to above>
select m.MEMBER, m.TBSP_ID, m.TBSP_NAME,
```

```
      m.POOL_INDEX_P_READS, m.POOL_INDEX_WRITES
  from mgtbsp_delta as m
  where M.TBSP_ID
        in (<table space IDs of hot containers>)
  order by m.POOL_INDEX_P_READS + m.POOL_INDEX_WRITES desc
```

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 in these table spaces are most active? Query SYSCAT.TABLES and SYSCAT.INDEXES, matching INDEX_TBSPACE with MON_GET_TABLESPACE.TBSP_NAME, from step 1. This gets us the indexes in this table space, which we join with the output from MON_GET_INDEX to find out which are most active.

```
with <delta view mgi_delta similar to above>
select
  T.TABSCHEMA, T.TABNAME, I.INDNAME,
  M.INDEX_SCANS, M.INDEX_ONLY_SCANS

from syscat.tables T, syscat.indexes I, mgi_delta as M

where T.TABNAME = I.TABNAME
  AND I.TABNAME = M.TABNAME
  AND T.TABSCHEMA = I.TABSCHEMA
  AND I.TABSCHEMA = M.TABSCHEMA
  AND I.IID = M.IID
  AND COALESCE(T.INDEX_TBSPACE, T.TBSPACE) IN

     ( <names of table spaces with hot containers> )

order by M.INDEX_SCANS desc
```

System Bottleneck > Container or Storage Path 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 pages 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 different 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 or Storage Path Disk Bottleneck > Hot Index Container > Hot Index > Hot SQL Statement?

If we cannot eliminate the index I/O bottleneck by 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 for index names as we did before with table names, 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 SQL statements that might be using those indexes. There is no guarantee that a reference to the table means that an index is used. However, if we focus on SQL statements that drive high volumes of index reads or writes (POOL_INDEX_P_READS and POOL_INDEX_WRITES from MON_GET_PKG_CACHE_STMT) we can easily narrow down the candidate statements. We can then use `db2exfmt` to confirm that the statements used the indexes we're interested in.

As mentioned above, DB2 10 introduces usage lists, which allows the user to track exactly which SQL statements refer to particular tables or indexes. If a sufficiently recent version of DB2 is being used, usage reference lists make the process of progressing from hot indexes to hot SQL statements much easier.

System Bottleneck > Container or Storage Path Disk Bottleneck > Hot Temporary Table Space Container

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 can occur when sorting activity overflows the designated in-memory buffers and must use a temporary table space instead. If the sort time and spilled sorts monitor elements are high and increasing, this might be the cause.

```
with <delta view mgw_delta similar to above>
select w.MEMBER,
  w.TOTAL_SECTION_SORTS,
  w.TOTAL_SECTION_SORT_TIME,
  w.SORT_OVERFLOWS
from mgw_delta as w
```

   The STMM tries to avoid this kind of situation; however if you are not using the STMM to control `sheapthres_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 monitoring data, and if there was evidence of high temporary data I/O, we would then drill down into the SQL monitor data via MON_GET_PKG_CACHE_STMT, looking for individual statements that caused high levels of temporary buffer pool activity.

```
with <delta view mgtbsp_delta similar to above>
select t.MEMBER, t.TBSP_NAME,
       t.POOL_TEMP_DATA_P_READS, t.POOL_DATA_WRITES
  from mgtbsp_delta as t
  where TBSP_ID in ( <table space IDs of hot containers> )
```

System Bottleneck > Container or Storage Path Disk Bottleneck > Poor Configuration?

Suppose that we have identified one or more of the above types of containers as bottlenecks, 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 or storage paths to the table space (provided they were on different disks than the existing containers, so that I/O operation capacity was increased).

2. Has a new storage path been added to an automatic storage configuration, but rebalance not been performed yet? The potential performance benefit of having a new storage path (assuming it introduces new disk spindles to the AS configuration) may not be seen until a rebalance is performed, to spread existing data equally over all paths.

3. Are there too many table spaces sharing the same disks? Many table spaces spanning the same disks is a natural occurrence when Automatic Storage is in use, but it can even happen inadvertently, with any table space type, 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.

4. Could there be issues with storage configuration below the DB2 software? Candidate causes here could include too many disks attached via too few disk controllers, incorrect configuration at the operating system level (see the discussion on `queue_depth` and `num_cmd_elems` above, for example), or storage errors like RAID array rebuilding, which have a strong performance impact. Possibly with the help of your storage administrator, you may see symptoms like very long controller-level I/O times, or very high peak I/O times, in `iostat`.

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 operation / throughput capacity to the problem table space to improve performance.

System Bottleneck > Log Disk Bottleneck?

Although container / storage path 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, a main symptom is very high disk utilization, as reported in `iostat` or `perfmon` (90% or

higher). A log bottleneck also causes long commit time as shown in MON_GET_WORKLOAD, and, most importantly, longer log write times.

If high disk utilization occurs on the log disk, we first verify that log write times are high.

```
with <delta view mgtl_delta similar to above>
select
  case when l.NUM_LOG_WRITE_IO > 1000 then
   decimal(l.LOG_WRITE_TIME) / l.NUM_LOG_WRITE_IO
  else NULL end as l.TIME_PER_LOG_WRITE_MS ,

  case when l.NUM_LOG_READ_IO > 1000 then
    decimal(l.LOG_READ_TIME) / l.NUM_LOG_READ_IO
  else NULL end as l.TIME_PER_LOG_READ_MS

from mgtl_delta as l
```

The above query uses the MON_GET_TRANSACTION_LOG table function, introduced in DB2 10. For earlier versions of DB2, you can use SNAPDB, as follows:

```
with <delta view sdb_delta similar to above>
select
  case when db.NUM_LOG_WRITE_IO > 1000 then
   (1000.0 * db.LOG_WRITE_TIME_S
           + db.LOG_WRITE_TIME_NS/1000000.0)
     / db.NUM_LOG_WRITE_IO
  else NULL end as TIME_PER_LOG_WRITE_MS ,

  case when db.NUM_LOG_READ_IO > 1000 then
    (1000.0 * db.LOG_READ_TIME_S
            + db.LOG_READ_TIME_NS/1000000.0)
     / db.NUM_LOG_READ_IO
  else NULL end AS TIME_PER_LOG_READ_MS

from sdb_delta as db
```

If the average log I/O time is in an acceptable range (for example, 2-3ms or less), then there's not likely to be much benefit in driving down log disk usage. Otherwise, we have good reason to proceed with further investigations of log performance.

As mentioned in the section on log configuration, if possible, the log should 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 dig deeper to understand the cause of the bottleneck.

1. If `iostat` or MON_GET_TRANSACTION_LOG shows that the log device is performing more than about 200 operations per second, and if the average I/O size is about 4 KB, this indicates that log activity is more dominated 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 by comparing the ratio of commits to activities completed in MON_GET_CONNECTION, and also by looking at the rate 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 can have 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 MON_GET_WORKLOAD, MON_GET_CONNECTION, and others) indicates that this is the likely cause of the problem.

System Bottleneck > Log Disk Bottleneck > Large Number of Log Writes

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 a bigger factor than high transaction rate.

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

   a. When a DB2 system updates a row in a table, it logs all column data from the first modified column to the last modified column – including any columns in between that are not 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.

d. If the 'currently committed' feature is enabled (CUR_COMMIT), DB2 needs to log the 'before' value of updated rows. This feature can be extremely useful in eliminating lock contention between readers and writers, but it does increase log write volume. The CUR_COMMIT monitor elements in MON_GET_TRANSACTION_LOG can help determine if the CUR_COMMIT feature is providing enough benefit to justify the extra log overhead.

System Bottleneck > Log Disk Bottleneck > High Volume of Data Logged

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.

System Bottleneck > Diagnostic Path Bottleneck?

Heavy disk writes on the DB2 diagnostic path – where the db2diag.log is located – can cause an overall system slowdown that can be isolated using the DIAGLOG_WRITE_WAIT_TIME element in table functions such as MON_GET_WORKLOAD_DETAILS. Like many other types of wait time, we mainly want to see if it is a large portion of TOTAL_WAIT_TIME .

```
with
mgwd_xml_delta (
  MEMBER,
  METRIC_NAME,
  VALUE )
as (
  select
    t.MEMBER,
    m.METRIC_NAME,
    sum(m.VALUE - mgwd_xml_baseline.VALUE)
  from
    table( MON_GET_WORKLOAD_DETAILS( null,-2 ) ) AS T,
    table( MON_FORMAT_XML_METRICS_BY_ROW(T.DETAILS)) AS M,
    session.mgwd_xml_baseline as mgwd_xml_baseline
  where
    t.MEMBER      = mgwd_xml_baseline.MEMBER and
    m.METRIC_NAME = mgwd_xml_baseline.METRIC_NAME
  group by t.MEMBER, m.METRIC_NAME )

select
  substr(d.METRIC_NAME, 1, 25) as METRIC_NAME,
  d.VALUE
from mgwd_xml_delta as d
where d.METRIC_NAME
```

```
  in ('TOTAL_WAIT_TIME', 'DIAGLOG_WRITE_WAIT_TIME')
```

In a partitioned database environment, all partitions generally write to the same diagnostic path, which is typically shared over the network through NFS or GPFS. 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, 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

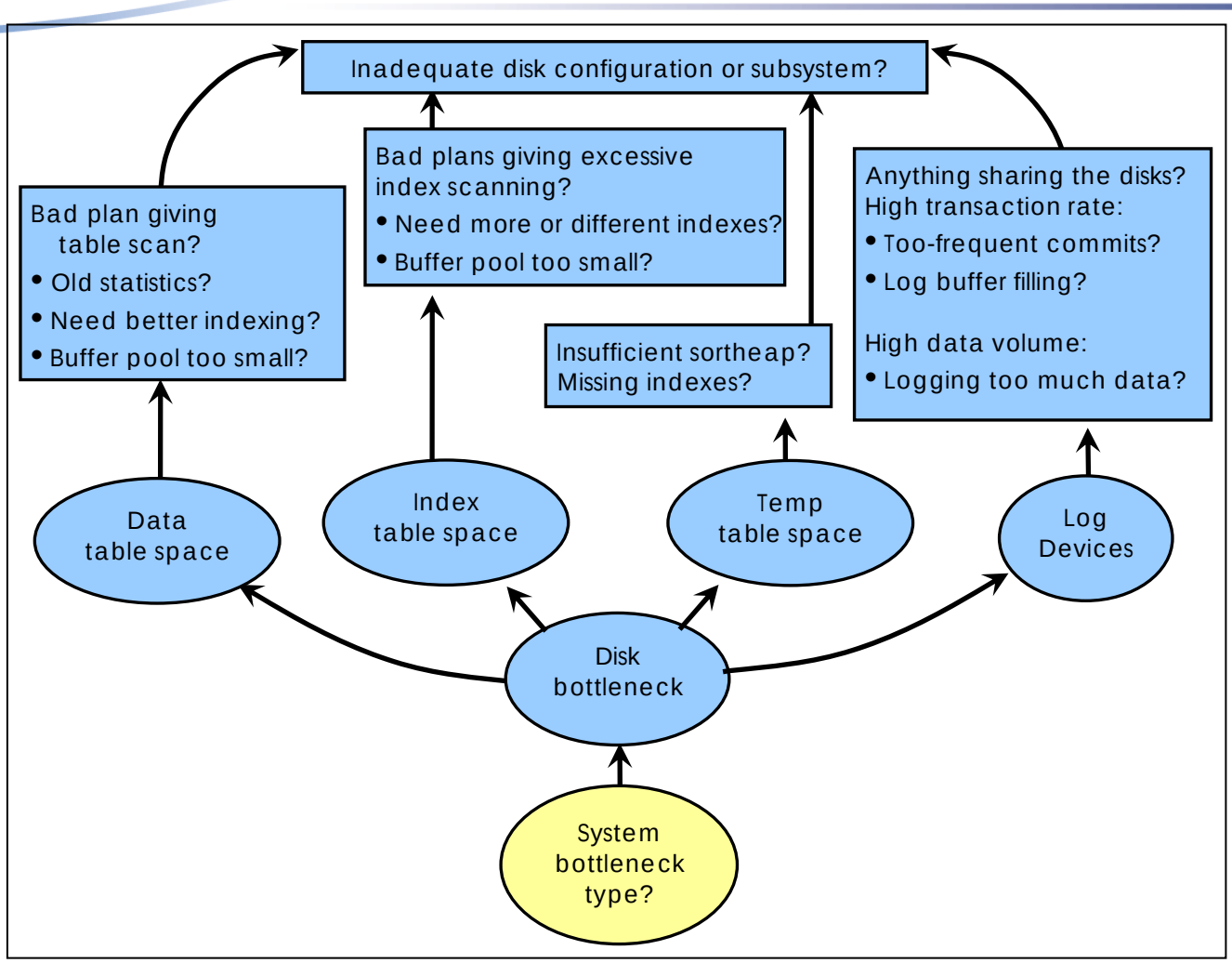


Figure 1. Illustration of different types of disk bottlenecks.

CPU bottlenecks

System Bottleneck > CPU Bottleneck?

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, and is often collected using the `vmstat` or `perfmon` commands, or via ENV_GET_SYSTEM_RESOURCES. 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 at a time. 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.

While we will focus on DB2's CPU usage, it's good practice to start out a CPU bottleneck investigation with seeing whether there are any non-DB2 processes (applications, scripts, commands, etc.) that may be running out of control and consuming too much CPU. In such cases, ideally these can be stopped and the CPU bottleneck eliminated.

System Bottleneck > CPU Bottleneck > User CPU Bottleneck

Many causes of a user CPU bottleneck on a DB2 server can be diagnosed through the DB2 monitor table functions. Drill down to find out which users are consuming the most CPU time by using the MON_GET_CONNECTION table function:

```
with <delta view mgc_delta similar to above>
select
  c.MEMBER,
  c.APPLICATION_NAME,
  c.TOTAL_CPU_TIME
from mgc_delta as c
order by c.TOTAL_CPU_TIME desc
```

Similarly, you can also determine which SQL statements are using the most CPU time from the MON_GET_PKG_CACHE_STMT:

```
with
mgpcs_delta (
  MEMBER,
  STMT_TEXT_1024,
  TOTAL_CPU_TIME )
as (
  select
    mgpcs.MEMBER,
    varchar(substr( mgpcs.STMT_TEXT,1,1024 )),
    sum( mgpcs.TOTAL_CPU_TIME
       - mgpcs_baseline.TOTAL_CPU_TIME)
  from
    table(mon_get_pkg_cache_stmt(null,null,null,-2)) as mgpcs,
    session.mgpcs_baseline as mgpcs_baseline
  where
    mgpcs.member = mgpcs_baseline.member
    and varchar(substr( mgpcs.STMT_TEXT,1,1024 )) =
        varchar(substr( mgpcs_baseline.STMT_TEXT,1,1024 ))
  group by mgpcs.MEMBER, varchar(substr( mgpcs.STMT_TEXT,1,1024
)) )
```

```
select
  mgpcs.MEMBER,
  substr( mgpcs.STMT_TEXT_1024,1,128),
  mgpcs.TOTAL_CPU_TIME
from mgpcs_delta as mgpcs
order by TOTAL_CPU_TIME 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_CPU_TIME.

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 rapidly rising number of table scans in MON_GET_TABLES
   - 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 or Optim Query Workload Tuner 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 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" (https://ibm.biz/Bdx2nr)).

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.

System Bottleneck > CPU Bottleneck > User CPU Bottleneck > Dynamic SQL without Parameter Markers?

1. Some 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 inefficientQuery =
      "SELECT language FROM "
    + "syscat.procedures "
    + "WHERE procname =  "
    + " '"
    + procNameVariable // inject literal value at prep time
    + "' ";

String betterQuery =
      "SELECT language FROM "
    + "syscat.procedures "
    + "WHERE procname = ? ";  // provide value at exec time
```

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.

Some signs that this problem might be occurring are:

a. A large number of similar but not identical statements in the package cache (seen via MON_GET_PKG_CACHE_STMT, or MONREPORT.PKGCACHE, etc.) that 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 (PKG_CACHE_INSERTS from MON_GET_WORKLOAD, etc.), even after the system reaches steady state.

c. More than 15% of DB2 processing time spent compiling SQL statements (seen via TOTAL_COMPILE_TIME divided by TOTAL_RQST_TIME, from MON_GET_WORKLOAD, etc.)

d. High and increasing memory usage by the package cache, if it has been set to automatic.

The best practice for avoiding this overhead is to ensure that applications use parameter markers for simple dynamic SQL statements. If this is not possible, then the DB2 statement concentrator may be able to reduce the impact of there being so many 'almost identical' statements. This can be a very helpful technique when the system does not also run more complex statements, with literals, that depend on distribution statistics (in such cases, the removal of literals by the statement concentrator can result in inefficient access plans and poor performance.)

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. The `db2pd -edus` command shows all of the various worker threads inside the DB2 engine (see the DB2 Process Model (http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/index.jsp?topic=%2Fcom.ibm.db2.luw.admin.perf.doc%2Fdoc%2Fc0008930.html)), including their user and system CPU usage. This is very useful in determining if any of the utility worker threads are behind the CPU bottleneck. The `db2pd -edus` command can be also used to report the top CPU-consuming DB2 threads during a specified interval (by using the *`interval`* and *`top`* options of the `db2pd -edus` command).

   Setting UTIL_IMPACT_PRIORITY can be helpful in limiting 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 making use of sampling `runstats`. In DB2 10.1, you can enable automatic `runstats` sampling for large tables by setting the database configuration parameter

`auto_sampling` to ON and setting profiles for statistics collection, to avoid `runstats` overhead during critical processing periods. Good recommendations for the latter are included in "Writing and Tuning Queries for Optimal Performance" (https://ibm.biz/Bdx2ng).

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.

System Bottleneck > CPU Bottleneck > User CPU Bottleneck > Temporary Object Cleanup Overhead

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 MON_GET_TABLE shows that there are a significant number of temporary tables being created and destroyed, a best practice is to place temporary table spaces in their own buffer pool. This eliminates the extra conflict and processing overhead, and can contribute to reduced CPU consumption.

System Bottleneck > CPU Bottleneck > System CPU Bottleneck?

Although user CPU tends to be the dominant factor in most CPU-bound environments, system CPU time can sometimes be an issue, 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 around 25,000 context switches per second per core would be considered quite high, even for large systems.

System Bottleneck > CPU Bottleneck > High System CPU > High Context Switches

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 ERP 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 or other powerful SQL construct, 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, you should generally 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) or in the OPM Overview dashboard. 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. This is the default setting in AIX 6 and later.

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

Servers with very large amounts of physical memory – 100s of 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 common OS page size is 4 KB – meaning that the OS must look after 25 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" (http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/index.jsp?topic=%2Fcom.ibm.db2.luw.admin.dbobj.doc%2Fdoc%2Ft0010405.html). The best practice on most large memory systems running AIX 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 (http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.admin.regvars.doc/doc/r0005665.html#r0005665).

On HP-UX systems, you can set the size of the base page (the smallest block of physical memory that can be allocated by the HP-UX kernel) by using the `kctune base_pagesize` command. While the default page size in HP-UX is 4 KB, the recommendation for larger systems is to increase the base page size to 16 KB (`kctune base_pagesize 16`).

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.

System CPU bottlenecks: The overall picture

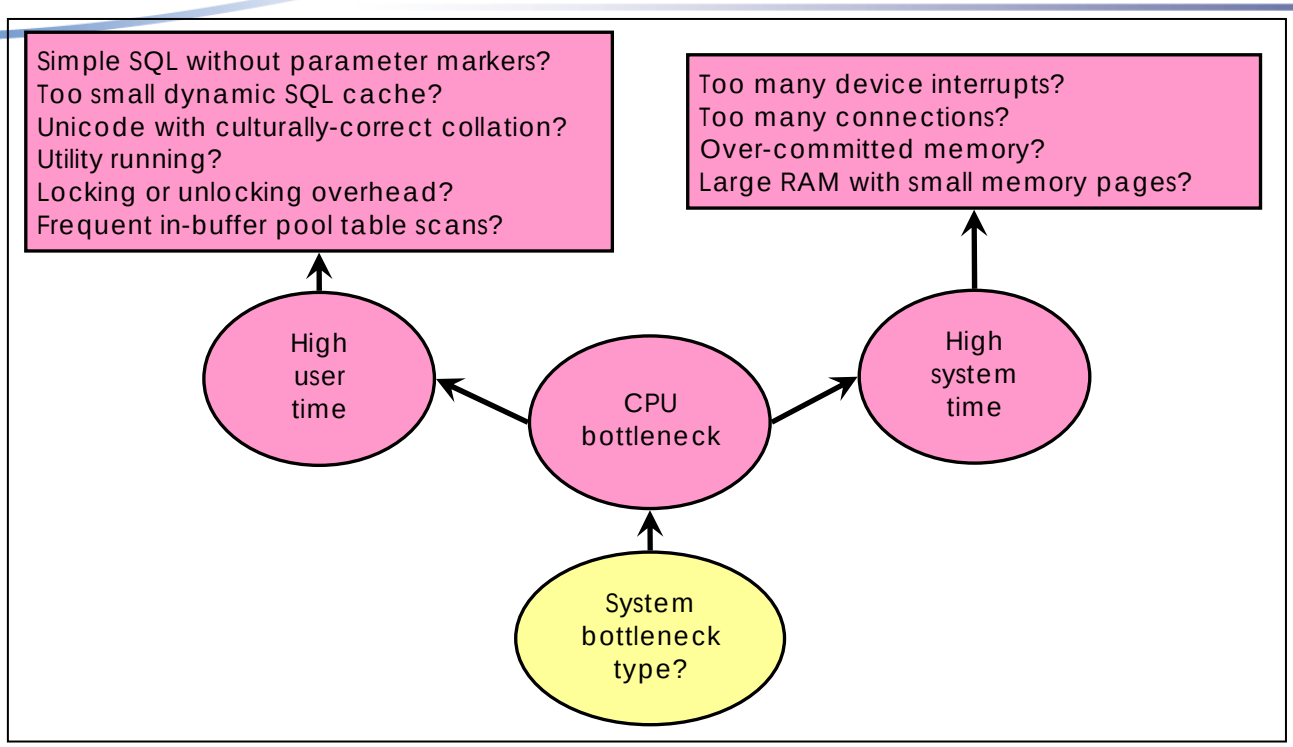


Figure 2. Illustration of different types of CPU bottlenecks

Memory bottlenecks

System Bottleneck > Memory Bottleneck?

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. Without sufficient 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.

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

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.

The multithreaded DB2 architecture that was introduced in DB2 9.5 on all platforms greatly simplifies the configuration and optimization of memory usage of DB2 servers. The overall limit of DB2 memory usage is set by a single database manager configuration parameter called INSTANCE_MEMORY. In partitioned environments,

INSTANCE_MEMORY specifies the maximum amount of memory that can be allocated for a database partition.

INSTANCE_MEMORY is set to AUTOMATIC by default. This allows instance memory to grow as needed – up to a limit between 75% and 95% of the physical RAM of the server. In environments where other applications share the same server with the DB2 system, setting a specific value for INSTANCE_MEMORY is recommended.

The easiest way to monitor the overall memory usage of the DB2 server (or partition in partitioned environments) is by using the `db2pd -dbptnmem` command (alternatively, you can query the ADMIN_GET_MEM_USAGE table function). The output of the `db2pd -dbptnmem` command lists the current memory usage as well the high-water mark (HWM) of both overall INSTANCE_MEMORY as well as for the different memory sets in DB2. The most important memory sets in a DB2 server environment are:

1. Database Manager memory set: The memory used by the DB2 instance itself (for example, for DB2 monitoring or DB2 Audit).

2. Database memory set: This memory set is usually the largest in a DB2 server. It contains all the shared database memory consumers like the buffer pools, the sort heap, the lock list, and the package cache. This memory set is controlled with the database configuration parameter DATABASE_MEMORY (default value is AUTOMATIC).

3. Application memory set: Memory used by application-specific processing like statement heap. This set is configured with the database configuration parameter APPL_MEMORY (defaulted to AUTOMATIC).

In addition to these memory sets, a DB2 server has additional memory sets like PRIVATE (general purpose), FMP (for fenced mode processing), and FCM (fast communication manager in cluster environments). In addition to the `db2pd -dbptnmem` command, you can use the MON_GET_MEMORY_SET table function to see the size of the different memory sets.

For all the major consumers in the database memory set you can enable self-tuning memory manager (STMM). STMM will then tune the different memory pools like lock list or package cache that are part of the DATABSE_MEMORY. An easy way to determine actual memory consumption of the different memory pools is with the MON_GET_MEMORY_POOL table function (alternatively, you can use the `db2pd -mempools` command).

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 might tune DB2 memory consumption right up to the value of INSTANCE_MEMORY if the workload on the database is high. This is different from the behavior when INSTANCE_MEMORY is set to AUTOMATIC, and so 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. If your DB2 database system shares the same host with another large memory consumer like WebSphere or an SAP central instance, make sure to set INSTANCE_MEMORY to a value that leaves enough memory for both DB2 and the application.

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.

2. An explicit value of INSTANCE_MEMORY 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 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 are 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 with the use of the connection concentrator.

'Lazy System' bottlenecks

System Bottleneck > Lazy System

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 DB2 monitoring data. The monitor elements LOCK_WAIT_TIME and TOTAL_ACT_TIME, available with MON_GET_WORKLOAD, indicate total lock wait time and total amount of time spent processing database activities, respectively. A high percentage of lock wait time compared to total activity time (for example, 20% or higher) indicates that locking might be a significant bottleneck.

We can drill down from high level data from MON_GET_WORKLOAD by examining the per-statement LOCK_WAIT and TOTAL_ACT_TIME columns returned from MON_GET_PKG_CACHE_STMT. Similarly, these metrics are also available at the application level in MON_GET_CONNECTION. Both of these sources are very useful in helping to identify the source of a locking bottleneck.

We can get information about individual in-flight lock waits from the MON_GET_APPL_LOCKWAIT table function. It shows information such as

- Lock mode – shared or exclusive
- Object type – row, table, and so on
- Agent ID of holder and requestor
- Time that the lock wait started

Unlike many other types of DB2 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 monitor data are most valuable if collected repeatedly 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 table functions or administrative views, and storing it in DB2 tables.

Also, unlike most other types of monitor table functions, the main overhead in lock monitoring via MON_GET_LOCKS and MON_GET_APPL_LOCKWAIT is in collecting the data. So, although it is useful to have regular lock data collected and stored for analysis, overly frequent collections 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.

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.

If it is common in your environment that performance suffers due to readers (applications selecting rows) being blocked by writers (applications updating or deleting rows), enabling 'currently committed' semantics may help reduce contention. This

feature allows readers to continue execution by providing them with the currently committed value of the locked row, rather than forcing them to wait for an as-yet uncommitted value. It is not currently fully supported in DB2 pureScale environments. See the DB2 Information Center (http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/index.jsp?topic=%2Fcom.ibm.db2.luw.admin.config.doc%2Fdoc%2Fr0053556.html) for more information about currently committed.

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. Table functions like MON_GET_WORKLOAD report a count of lock escalations (LOCK_ESCALS). This is most easily broken down to escalations in individual tables by examining MON_GET_TABLE. Helpful information can also be found at the statement level in MON_GET_PKG_CACHE_STMT.

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 Timeouts

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 a participating 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 many of the monitor table functions, such as MON_GET_WORKLOAD. 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. Regardless, 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. Sometimes, missing or stale statistics can cause inefficient access paths, resulting in more rows being looked at, potentially causing increased lock contention.

In DB2 V9.7 and later, DB2DETAILDEADLOCK is deprecated, and the new locking event monitor (CREATE EVENT MONITOR FOR LOCKING) should be used instead. Like all event monitors, it imposes a small amount of additional overhead (somewhat higher if history and values are collected too); however, the benefit from being able to

track deadlocks usually outweighs the small performance penalty. The new locking event monitor can capture lock timeouts too, which were invisible to the old DB2DETAILDEADLOCK mechanism. This can produce very helpful information to diagnose participants in the lock contention. See the DB2 Information Center (http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.admin.mon.doc/doc/c0054136.html) for more information.

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, if 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% * (pool_data_p_reads – pool_async_data_reads) /
pool_data_p_reads
```

  This can be calculated at the database level with MON_GET_WORKLOAD and other functions or at the buffer pool level with MON_GET_BUFFERPOOL. It is not available at the statement level MON_GET_PKG_CACHE_STMT since prefetch I/O (pool_async_data_reads) is not captured there.

- High and climbing 'time spent waiting for prefetch', reported in PREFETCH_WAIT_TIME in MON_GET_WORKLOAD and MON_GET_CONNECTIONS, among others. 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.

A potential cause of this problem is that the number of prefetchers (database configuration parameter NUM_IOSERVERS) is too low. The AUTOMATIC setting 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 `db2pd -edus` command. 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 PREFETCHSIZE is set correctly. The process for this is discussed thoroughly in the DB2 Information Center (http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/index.jsp?topic=%2Fcom.ibm.db2.luw.admin.perf.doc%2Fdoc%2Fc0005397.html) so we do not repeat it here. You should also consider whether DB2_PARALLEL_IO should be set in your environment. This registry variable enables parallel I/O within one table space container, in the case when the container is located over multiple physical disks. Since this situation (multiple disks underneath a single filesystem) is extremely common in most non-trivial systems, it is generally a good idea to have DB2_PARALLEL_IO set (even just to '*').

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 'background 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 straightforward 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 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 are 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 the DB2 monitoring data, the count of dirty page steals (POOL_DRTY_PG_STEAL_CLNS in MON_GET_BUFFERPOOL) 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. Dirty steals can also show up as a growing gap or difference between the following two metrics from MON_GET_BUFFERPOOL:

- POOL_DATA_WRITES (total number of database pages written from the bufferpool), and
- POOL_ASYNC_DATA_WRITES (the total number of database pages written from the bufferpool *by the page cleaners*)

Of course, there are corresponding metrics for index pages, XML pages, etc...

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). The AUTOMATIC setting of NUM_IOCLEANERS follows the best practice of one cleaner per physical CPU core (logical CPU core in DB2 9.7 and on HP-UX) in the current partition. In DB2 Version 9.5 and later, extra cleaners beyond the recommended number can actually hurt performance somewhat.

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:
    - CHNGPGS_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, CHNGPGS_THRESH is the preferred way to affect the number of clean

pages in the buffer pool. Decreasing CHNGPGS_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 slowdown 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 ratio of CLIENT_IDLE_WAIT_TIME to TOTAL_REQUEST_TIME (both from MON_GET_WORKLOAD) indicates that more time is being spent above DB2. In many healthy systems, a typical ratio is around 4-5x, but it's the increase above normal (for your system) that's most important.

- An increased time between requests made at the client side, as seen by the activity event monitor, or a CLI or JDBC trace (http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.admin.trb.doc/doc/t0020709.html). 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 degradation.

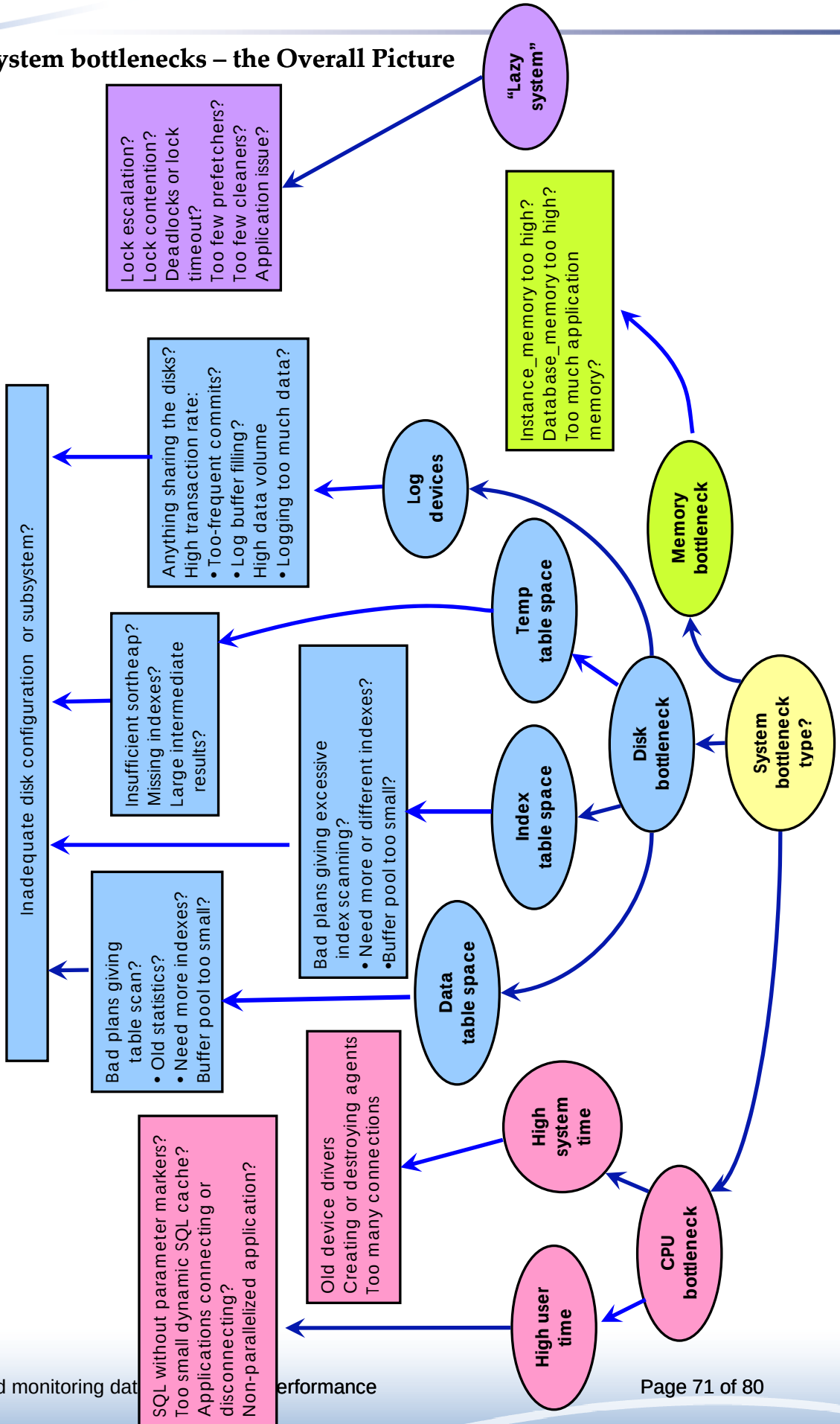
If an application-side slowdown appears to be the problem, possible causes include:

- Deployment of a new version of the application

- A network bottleneck between client and server

- Excessive load on the client system or in a tier located between client and server; for example, too many users or too many copies of the application running.

System bottlenecks – the Overall Picture



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 numeric 'application handle' which is the key to identifying monitor data that is specific to this application. We can also query the SYSIBMADM.MON_CURRENT_SQL administrative view, which returns one row for every SQL statement currently executing on the system, including identification information like APPLICATION_HANDLE, as well as basic monitoring information like ROWS_READ, etc.

2. The MON_GET_CONNECTION table function is an excellent source of application-specific monitoring data, and by specifying the application handle, we can focus on the connection of interest.

```
select * from table(MON_GET_CONNECTION( appl_handle, -1 ))
```

From MON_GET_CONNECTION and SYSIBMADM.MON_CURRENT_SQL, you can determine many important things about the application, such as the statement being executed when the monitor data was collected, 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. Furthermore, once the frequent & heavy statements are determined as above, additional statement-level detail can be obtained from MON_GET_PKG_CACHE_STMT. Although you're not really focused on changing the global CPU consumption or disk activity (remember we're assuming 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 multiple sets of connection-level and statement-level table function output, you have a similar palette 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 I/Os per second and throughput per second. In practical terms, this often comes down to the number and type (HDD, SSD, etc.) of disk spindles. If the number of available disks is low, use about 70% of each disk for data & index storage (either as individual containers or combined into an array), and use the remaining 30% as another array for database path and logs.
- Locate transaction logs on dedicated disks if possible, especially for high-write DB2 systems. If it's not possible, make sure to carefully track log write times, which we want to see in the 2-3ms range.
- Use the DB2 Data Partitioning Feature for data warehousing deployments larger than around 1TB.
- 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 autonomics provide stability and strong performance.
- In partitioned database environments, use a local rather than an NFS-mounted file system for DIAGPATH.
- If the system uses storage based on arrays of disks, set the

DB2_PARALLEL_IO registry variable.

Monitoring:

- Collect basic operational monitoring data regularly, so that background information is available in case of a problem.
- Use the monitor table functions and 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, an 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

- DB2 Best Practices
  http://www.ibm.com/developerworks/data/bestpractices/db2luw
    - Physical database design for OLTP environments: https://ibm.biz/Bdx2nr
    - Physical database design for data warehouse environments: https://ibm.biz/Bdx2np
    - Writing and tuning queries for optimal performance: https://ibm.biz/Bdx2ng
    - Database storage: https://ibm.biz/Bdx2My
- DB2 10.5 for Linux, UNIX, and Windows information center: http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/index.jsp
- DB2 for Linux, UNIX, and Windows V10.5 manuals: http://www-01.ibm.com/support/docview.wss?uid=swg27038430#manuals

Contributors

Sonia Ang
Senior Consulting IT Specialist

Edward Bernal
Websphere Performance Engineer

Dr. Toni Bollinger
IM Data Warehousing Center of Excellence

Roy Cecil
Advisory Performance Engineer
DB2 pureScale Performance

Louise Cooper
Information Management Technical Consultant
Hursley Innovation Centre

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
Senior DB2 Performance Manager

Tony Lau
Staff Performance Engineer
DB2 Performance

Scott Martin
Senior Software Engineer
Partner Ecosystems

Jacques Milman
Server Specialist
Business Intelligence & Data Warehousing

Mika Nikolopoulou
Senior Certified IT Specialist
Information Management Software

Steve Schormann
Senior Software Engineer
IBM Identity Analytics

Adam Storm
Senior Software Developer, Master Inventor
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
Armonk, NY 10504-1785
U.S.A.

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

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.

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.

Contacting IBM

To provide feedback about this paper, write to db2docs@ca.ibm.com

To contact IBM in your country or region, check the IBM Directory of Worldwide Contacts at http://www.ibm.com/planetwide

To learn more about IBM Information Management products, go to http://www.ibm.com/software/data/