IBM DB2® Universal Database™
DB2 Problem Determination Tutorial Series

Performance Problem Determination

Table of Contents

Performance Problem Determination

Introduction

Target audience

This tutorial is geared towards individuals with little or no knowledge in regards to debugging performance issues. It is useful if the reader has reviewed how to issue SQL and DB2 commands, has the ability to install and configure a basic database and instance, and has a high-level understanding of DB2's transaction model.

This tutorial explores techniques and methods that are available outside the GUI interface. That is, it focuses on command line syntax, techniques and interpretations.

Tutorial objectives

This tutorial will teach you to identify and debug DB2 UDB performance issues. The tutorial is divided into modules that examine the major factors influencing performance. The tutorial will also cover ways to exploit DB2's monitoring facilities and various OS level diagnostics for isolating potential performance areas. Finally you will get a brief introduction and overview the DB2 UDB query optimizer, and learn how to use the decisions made by the optimizer to further examine performance concerns and issues.

The tutorial combines relational database theory, DB2 design and hands-on examples that allow you to explore each topic more deeply. The final goal is, of course, to provide the basic foundation necessary to effectively isolate, intelligently adjust, accurately monitor, and exploit the performance capabilities of DB2.

Tutorial setup

The provided database image contains a modified version of the SAMPLE database. This has undergone necessary changes in order to work through the exercises of the Optimizer component of the tutorial. To use the image:

1. Please download PERFTUT.0.rbahra.NODE0000.CATN0000.20021125175204.001.gz from
   ftp://ftp.software.ibm.com/ps/products/db2/info/vr8/tutorials/PERFTUT.0.rbahra.NODE0000.CATN0000.20021125175204.001.gz

2. Uncompress the file into a temporary directory using:
   ```
   gzip -d PERFTUT.0.rbahra.NODE0000.CATN0000.20021125175204.001.gz
   ```

3. After the file has been uncompressed, you can restore the image using the following command:
   ```
   db2 restore db perftut into perftut
   ```

This command assumes that no database with the name perftut has been previously created and it also assumes that the command is run from the temporary directory used in step (2). If the name perftut is not usuable or acceptable, please feel free to change the restore command:

```
db2 restore db perftut into <changed database name>
```

It is assumed that you have access to a running version of DB2 and instance owner privileges for the database that you plan to test with.

Tutorial conventions

This tutorial uses a few simple typesetting conventions:
Boldface - emphasis on a particular field or value
Italics - additional comments and details not fundamental to the topic
`Monospace` - commands, queries and resulting output

About the author

Raj Bahra, the author, has been employed at the IBM Toronto Labs since 1999. His main area of focus recently has been on the DB2 query optimizer and assisting the DB2 user community with optimizer-specific issues.

You can reach Raj by locating his e-mail address in the *IBM Global Directory* at http://www.ibm.com/contact/employees/us .

Roadmap to debugging performance issues

Introduction

The tutorial is modular. Each major section can be reviewed independently or the tutorial can be read as a whole. This section serves as a roadmap for the tutorial. It also presents a problem solving methodology for debugging performance problems. You can use this as a checklist to follow when troubleshooting performance issues.

Initial analysis

Once you receive report of poor performance, ask the following questions to quickly determine the best place to start looking for a potential cause.

When did the problem first occur?
If the problem has been occurring for some time, and if a database monitor schedule has been implemented, you can use historical data to find differences. This will allow you to focus on changes in system behavior and then focus on why these changes were introduced. Monitoring is discussed later in this tutorial.

Is the performance issue constant or intermittent?
If the poor performance is continual, check to see if the system has started to handle more load or if a shared database resource has become a bottleneck. Other potential causes of performance degradation include increased user activity, multiple large applications, or removal of hardware devices. If performance is poor only for brief periods begin by looking for common applications or utilities running at these times. If users report that a group of applications are experiencing performance issues, then you can begin your analysis by focusing on these applications.

Does the problem appear to be system-wide or isolated to DB2 and its applications?
System-wide performance problems suggest an issue outside of DB2. It's possible that something at the operating system level needs to be addressed. If the poor performance is confined to DB2 applications, obviously we'll focus on what DB2 is doing.

If isolated to one application, is there a particular query that appears to be problematic?
If one application is problematic, then you can further examine if users are reporting a query or set of queries that are experiencing a slowdown. You might be able to isolate the issue to one application and a potential group of queries.

Is there any commonality to the poor performance or does it appear to be random?
You should determine if any common database tables, table space, indexes, etc. are involved. If so, this suggests that these objects are a point of contention. Other areas to potentially focus on are referential integrity constraints, foreign key cascades, locking issues etc.

These brief initial questions are invaluable in developing an action plan to help debug a performance problem.

Looking at DB2

It's possible that after your initial review, DB2 is identified as a strong possibility behind the performance issues. In this case, ask the following questions:

Are users reporting abnormally long response time for all applications?

1. Have you detected a large amount of I/O? Here are some possible causes:
    a. Overloaded devices. The technique to determine this will be discussed in the OS monitoring section.
    b. Heavy SORTS or use of TEMPSPACE. The techniques used to determine this will be covered in the sorting and optimizer sections.
    c. Table scan of a large table. The optimizer section provide techniques to determine this.
    d. Insufficient buffer pool. This is covered in the section on buffer pool monitoring.

2. Have you detected a high CPU usage? Possible causes include:
    a. Heavy SORTS. The techniques used to determine this are covered in the tutorial sections on sorting and on the optimizer.
    b. DB2 agent restrictions. Please refer to the buffer pool section, specifically the discussion of asynchronous I/O.
    c. Insufficient buffer pool sizes. Please refer to the buffer pool section, specifically hit ratios.

3. Both (1) and (2)?
    a. Increased number of users? Please refer to the tuning practices section.
    b. Large spilled SORTS. The techniques used to determine this are covered in the sections on sorting and optimizer.
    c. Application design?

4. Neither (1) and (2)?
    a. Are applications waiting on locks? Refer to the locking and concurrency section.
    b. Are applications waiting on data? The tutorial section on buffer pools provides techniques to see if data is being made available at reasonable rates.

Are users reporting abnormally long response time for one application or query?
Possible causes include:

1. SORTS - refer to the sorting and optimizer sections.
2. Concurrency issues - refer to the locking and concurrency section.
3. Stale statistics - refer to the optimizer section.

These are general questions to ask yourself when faced with a performance issue. This list is by no means exhaustive given the size and complexity of modern systems, performance issues can surface from a multitude of possibilities. Regardless, these questions will start you on a logical path to determining where the problem came from.

DB2 monitoring

Purpose of monitoring

Database monitoring is an important part of tuning a database server. You should develop a database monitoring plan. This not only allows for easier detection of a problem, but also enables you to refer back to historical data to isolate a change, extrapolate the effects of a change, better understand user and application activity and behaviors, and, of course, improve database performance!

Of course, monitoring does involve overhead and this should be taken into account when developing the monitoring plan. The frequency and volume of monitoring needs to be controlled.

Techniques for monitoring the database

DB2 has three techniques to monitor the database. This tutorial focuses on the first two:

- Snapshot monitoring - provides query operational database status for an instant in time. Snapshot monitoring is useful in determining the current state of the database and its applications.
- Event monitoring - provides query operational database status over a period of time for a specific activity. Event monitoring is useful in identifying database or application state changes. It is used to gather details about changes regarding different events such as connections, tables, statements, etc.
- Graphical performance monitor - receives data from the snapshot and event monitors. This utility is not covered in this tutorial.

In order to allow for effective memory usage, it is possible to configure how much memory is used for database monitoring. The MONHEAPSZ parameter, database manager field specifies how much memory is allocated on a DB2START and is used when monitoring takes place. Setting this parameter to 0 (zero) will not allow any monitoring to take place.

SYSADM, SYSCTRL, or SYSMAINT authorization is required in order to use monitoring tools.

Levels of monitoring

The monitoring utilities collect information at different levels within the system:

- *Database Manager* - the monitor gathers details from the time the instance is started till the time it is stopped.

- *Database* - this level of monitoring begins on the first connection or activation, and will continue until deactivation or the last the connection is terminated.
- *Application* - this level of monitoring begins when an application connects to the database and continues until it disconnects from the database.

Also, you can further break down the monitoring details by splitting certain items into functional groups:

SORTS
LOCKS
TABLE ACTIVITY
BUFFERPOOL ACTIVITY
UNIT OF WORK
SQL STATEMENTS

So, for example, if you notice long lock waits, it would be logical to monitor the LOCK group in an attempt to isolate the problem.

The path to monitoring

The following commands track current monitor settings, enabling functional groups, and "clear out" existing monitor data. These commands can be executed from a DB2 CLP session:

`GET MONITOR SWITCHES`
Returns the current state of the functional groups listed earlier. If you see a group turned on, you know that this group will be monitored.

`UPDATE MONITOR SWITCHES USING <switch name> ON | OFF`
This allows you to adjust the state of each of the functional groups. That is, you have the ability to enable or disable which groups are monitored via this command.

`RESET MONITOR ALL | FOR DATABASE <database alias>`
This allows you to reset monitor elements to zero, thereby allowing you to start fresh.

Snapshot monitoring

The parameters of the GET SNAPSHOT command determine the level of detail, and the type of monitoring. Some of the statistics returned by this command provide point-in-time information, others provide cumulative information since the last `reset monitor` command.

An example of a cumulative counter would be `deadlocks detected`. This holds the number of deadlocks since the last reset monitor command. An example of an event-specific field would be `lock list` memory in use. This field would contain the number of bytes currently used for the

lock list and can change from one iteration of a snapshot to the next, as applications connect and disconnect and obtain and release locks respectively.

Please refer to the *System Monitoring and Tuning Guide* for further details on the get snapshot command.

As an example, try using the **perfpd** database to take a snapshot: From a CLP session, issue the following commands:

```
db2 connect to perfpd
db2 reset monitor all
db2 get snapshot for database on perfpd
db2 "select * from syscat.bufferpools"
db2 get snapshot for database on perfpd
```

Creating an event monitor

The Event Monitor facility within DB2 also allows for a great deal of flexibility. The initial step is to first create an event monitor.

As an example, try creating an event monitor for the perfpd database:

```
db2 create event monitor evmonex1 for statements write to file
'./testfile' maxfiles 3 maxfilesize 1024 nonblocked append
```

The above command will create an event monitor called evmonex1 to capture all statement events. A maximum of 3 files of 4MB each will be created.

For details on the command syntax, see the *DB2 System Monitor Guide and Reference*.

Formatting an event monitor

The output files from a create event monitor command are written out in an unreadable format. In order to make the output readable, you need to run a tool called db2evmon:

```
db2evmon <database alias> <event monitor name>
```

As an exercise, try formatting the output of the evmonex1 event monitor created earlier.

```
db2evmon perfpd evmonex1
```


Operating system (OS) - Level monitoring

This section will review a few tools available at the operating system level on AIX. These serve as an excellent way to confirm findings, or even identify potential problems.

IOSTAT

You can use this utility to measure input/output (I/O) wait times. A higher IOWAIT will generally result in slower performance. As an example, run the following command:

```
iostat 5
```

%iowait - The percentage of time the cpu is idle while waiting on local I/O

%idle - The percentage of time the cpu is idle while not waiting on local I/O
The time is attributed to I/O wait when no processes are ready for the CPU but at least one process is waiting on I/O. In general, if I/O wait is 20% to 25% or higher, investigation of a disk bottleneck is in order.

%tm_act - percentage of time the disk is busy
Look for busy vs. idle drives. Moving data from more busy to less busy drives may help alleviate a disk bottleneck. Using this as a debugging tool is presented later in the tutorial.

Buffer pools

An introduction to buffer pools and isolating bottlenecks

Buffer pool contention can be a significant factor in database performance. This section of the tutorial covers the basic concepts and then focuses on best practices and identifying potential factors that could result poor performance.

Introduction

Very simply, buffer pools act as the 'in memory' work area for the database. The server uses the buffer pool to perform all transactional activity (reads, writes, updates, deletes etc.). Data is copied from disk to the buffer pool based on the requests made by applications. So, for example, if you were to issue an SQL query to select the number of sales for product *foo* from table *ABC*, DB2 will start to extract data from table ABC for the product foo into the buffer pool. In simpler terms, you can consider it a staging area.

Data can make its way to the buffer pool in one of two ways: via asynchronous I/O or via synchronous I/O. Data pages are in placed into the buffer pool by either **db2agents** (synchronous I/O) or by the **pre-fetchers** (asynchronous I/O). When pages from the buffer pool are written to disk either a db2agent (synchronous I/O) or **page cleaners** (asynchronous I/O) are involved.

db2agents are the most important part of the DB2 processing model. These agents are responsible for handling the operations of the database.

Pre-fetchers read data pages into the buffer pool anticipating their need by an application (asynchronously). These appear in a UNIX® implementation with a process name of **db2pfchr**. In most situations, these pages are read just before they are needed (the desired case). However, pre-fetchers can cause unnecessary I/O by reading pages into the buffer pool that will not be used. For example, an application starts reading through a table. DB2 detects this table read request and pre-fetching beings, but the application fills an application buffer and stops reading. Meanwhile, pre-fetching has been done for a number of additional pages. I/O has occurred for pages that will not be used and the buffer pool is partially taken up with those pages.

Page Cleaners monitor the buffer pool and asynchronously write pages to disk. The process name for these DB2 processes is **db2pgclnr**. Their goals are:

- Ensure that agents will always find free pages in the buffer pool. If an agent does not find free pages in the buffer pool, it must clean them itself, and the associated application will have a poorer response.
- Speed database recovery, if a system crash occurs. The more pages that have been written to disk, the smaller the number of log file records that must be processed to recover the database.

Although modified pages are written out to disk, the pages are not removed from the buffer pool right away, until the space is needed to read in new pages.

To exploit DB2's parallel processing capabilities, have db2agents perform query processing and keep the pre-fetchers and I/O cleaners busy performing I/O tasks. This will maximize throughput by exploiting the specialized abilities of each type of process.

Buffer pools best practices

On a dedicated database server, a general rule of thumb is to have 75% of main memory assigned to the database's buffer pools. Sufficiently large buffer pools help reduce the I/O demands on the system.

The DB2 query optimizer takes buffer pool sizes into account when generating access plans. If the size of the buffer pools is modified, consider rebinding static packages to reflect the change.

The sizes of all buffer pools must also be taken into account. A system with a limited amount of memory or a system running other applications using large portions of memory may experience excessive page swapping. Page swapping is a result of insufficient memory to hold the data page being accessed. This results in pages being written to temporary disk storage in order to make room for other pages.

The I/O cleaners should be configured such that you have one or two more than the number of physical devices on which the database exists.

Bufferpool monitoring - hit ratio

As the buffer pool is intended to hold copies of data pages in memory versus reading the pages from disk, a measurement of the buffer pool's effectiveness would be to see how frequently pages requested were already in the buffer pool. The hit ratio measures the effectiveness. The general equation to calculate hit ratio is:

```
 (1-((pool_data_physical_reads+pool_index_physical_reads)/
(pool_data_logical_reads+pool_index_logical_reads))) * 100
```

The closer the value is to 100 percent the lower the frequency of disk I/O and hence the lower the overhead of reading data. This equation takes into account all of the pages (index and data) that are cached by the buffer pool. The equation can be simplified to only include index details or data page details, as long as the 'physical over logical' relationship is preserved.

These fields can be examined on the database, table space, or buffer pool functional levels.

Given the following sample values, determine the buffer pool hit ratio.

```
Buffer pool data logical reads             = 701865
Buffer pool data physical reads            = 50842
Buffer pool index logical reads            = 444859
Buffer pool index physical reads           = 24174
```

The buffer pool hit ratio for the above values is approximately 93.4. This value is quite close to 100 and suggests that this buffer pool is running quite efficiently. By tracking the buffer pool hit ratio on a daily or weekly basis, you can begin to keep a historical record of buffer pool efficiency. This record can be used as reference in the event performance issues are reported for your database.

If you do find that the hit ratio is quite low for a particular buffer pool, this may suggest that the buffer pool is being poorly used. You could try one of the following in an attempt to correct this:

- Split the data and indexes into two different buffer pools and monitor them separately.
- Use one buffer pool, but increase its size until the index hit ratio stops increasing. The index buffer pool hit ratio can be calculated as follows:

```
(1 - ((pool_index_p_reads) / (pool_index_l_reads))) * 100%
```

The first method is often more effective, but because it requires indexes and data to reside in different table spaces, it may not be an option for existing databases. It also requires tuning two buffer pools instead of one, which can be a more difficult task, particularly when memory is constrained.

Locking - lock waits, timeouts, escalations and deadlocks

Lock wait and lock timeout

An application that makes a request for a lock that is not compatible with the existing locks on the object, or a lock request not already satisfied will be placed into a lock request pending queue. The lock request will continue to be held for the waiting application until either timeout period is exceeded or a deadlock is the cause of the result.

The LOCKTIMEOUT database parameter, measured in seconds, is used to configure how long a period an application will wait for a lock to be made available. If the timeout period is exceeded, the waiting application receives an SQL0911 R.C. 68 error message and the application's unit of work is automatically rolled back by the database manager.

The default value for LOCKTIMEOUT is -1, which causes lock timeouts to be disabled. That is, an application that is waiting for a lock will continue to wait indefinitely or until the lock is released. For transaction environments, the recommended starting value is 30 seconds, however, tuning may be necessary in order to find a more appropriate value for this field.

Lock escalations

Lock escalation can occur in two different scenarios:

1. A single application requests a lock that will exceed its allowable number of locks.
2. An application triggers lock escalation because the maximum number of database locks on the system has been exceeded.

In both cases, the database manager will attempt to free up memory allocated to locking by obtaining table locks and releasing existing row locks. The desired effect is to make more lock memory available for additional applications.

The following two database parameters have a direct affect on lock escalation:
locklist - the total number of 4k pages allocated for lock storage (please refer to the memory model)
maxlocks - the allowable percentage of locklist that can be used by a single application

Tuning and monitoring may be necessary to find a balance for these values. Workload, and query behavior dictate locking patterns and how applications will use lock memory.

Deadlocks

Deadlocks occur when applications cannot complete a unit of work due to conflicting lock requests that cannot be resolved until the unit of work is completed. The DB2 deadlock detector is responsible for handling deadlocks. When a deadlock is detected, the deadlock detector will choose a victim that will be automatically rolled back and issued an SQL0911 R.C. 2. By rolling back the victim, the lock conflict is removed, and the other application can continue processing.

The frequency at which the DB2 deadlock detector checks for deadlocks can be controlled via DLCHKTIME (measured in milliseconds, from 1000 to 600000). Setting this value high will cause the deadlock check time to be increased but the overhead of deadlock detection will be removed. This could potentially result in applications stuck in deadlock for prolonged periods of time. Setting the deadlock check time to a smaller value allows for deadlocks to be detected sooner, however it also introduces additional overhead for the checking.

Tracking lock waits and timeouts

You can easily track lock waits and timeouts from the output of the snapshot monitor that you learned to run earlier in this tutorial. The specific field to look at is `Number of Lock Timeouts`. Here is a truncated example of snapshot output:

```
Locks held by application                  = 46
Lock waits since connect                   = 12
Time application waited on locks (ms)      = 96443
Deadlocks detected                         = 0
Lock escalations                           = 0
Exclusive lock escalations                 = 0
Number of Lock Timeouts since connected    = 5
Total time UOW waited on locks (ms)        = Not Collected
```

In the example, the application experienced 5 lock timeouts during its connection to the database. Also, we see that the application waited 96443 milliseconds for locks to be released. The average wait time can be calculated as follows:

```
(Time application waited for Locks / Lock waits since connect)
```

In the above case we see that each lock wait had an average time of 8036 milliseconds. In some environments (for example decision support systems) this might be considered normal, in others (for example online transaction processing) it would be considered excessive. Of course, user needs and business requirements should be used as measures.

In cases where you find lock timeouts becoming excessive when compared to normal operating levels, you may have an application that is holding locks for an extremely long period of time. It is also possible that if the number of lock timeouts is extremely low but the response time for queries is sluggish, the lock timeout parameter may be set too high.

In order to identify the cause of a lock timeout and lock wait, let's try the following example:

Please note, to simplify and not introduce unnecessary workload onto your environment, we're going to model a long running query by disabling autocommit.

1. Open two DB2 command windows, and connect to perfdb from both windows
2. In the first window, disable auto commit:
   a. `db2 update command options using c off`
3. Verify that autocommit has been turned off:
   a. `db2 list command options`
   b. `<check for Auto-Commit as turned OFF>`
4. From the first window, issue the following query:
   a. `db2 "insert into rbahra.table1 values (1,1,1)"`
5. From the second window issue the following query:
   a. `db2 "select * from rbahra.table1"`
6. Obtain a snapshot
   a. `db2 get snapshot for applications on perfdb`
7. View the snapshot
8. Issue a commit in both windows:
   a. `db2 commit`
9. Enable autocommit.

Look for the "`Application Status`" field. You should see one of the applications in UOW waiting, and the other in Lock Wait. In the above case, you can see that application issuing the SELECT had made a lock request, but the application performing the insert was holding the requested lock. In the example, you had forced the application to never release locks until a commit, but the principle can be modeled to be the same in cases where locks are released after long periods of time. This simple example demonstrates how to use a snapshot to find an application in Lock Wait. The next course of action may be to focus on what locks the application was waiting for and why those locks were not being released. Again, in this simplified case we actually forced this to happen, however areas of focus may be the frequency of commits for the application, the locking semantics (isolation level) used or the size of an application's unit of work.

Tracking deadlocks

Snapshots provide valuable insight into the number of deadlocks that have occurred. If you find that the number is abnormally large, you should consider enabling an event monitor to track the deadlock events more closely. The following event monitor output shows two applications in deadlock:

```
499) Deadlocked Connection ...
  Appl Id: *LOCAL.db2inst1.020921174012
  Appl Seq number: 0001
  Appl Id of connection holding the lock: *LOCAL.db2inst1.020921174009
  Seq. no. of connection holding the lock: 0001
  Lock wait start time: 09-21-2002 17:36:28.722587
  Requesting lock as part of escalation: FALSE
  Deadlock detection time: 09-21-2002 17:37:43.379432
  Table of lock waited on      : SALES
  Schema of lock waited on     : TRACKINGDB
  Tablespace of lock waited on : TRACKSPACE
  Type of lock: Row
  Mode of lock: X
  Mode application requested on lock: X
  Node lock occured on: 0
  Lock object name: 1298696
  Application Handle: 7

500) Deadlocked Connection ...
  Appl Id: *LOCAL.db2inst1.020921174009
  Appl Seq number: 0001
  Appl Id of connection holding the lock: *LOCAL.db2inst1.020921174012
  Seq. no. of connection holding the lock: 0001
  Lock wait start time: 09-21-2002 17:37:12.756160
  Requesting lock as part of escalation: FALSE
  Deadlock detection time: 09-21-2002 17:37:43.379432
  Table of lock waited on      : INVENTORY
  Schema of lock waited on     : TRACKINGDB
  Tablespace of lock waited on : INVNTSPACE
  Type of lock: Row
  Mode of lock: X
  Mode application requested on lock: X
  Node lock occured on: 0
  Lock object name: 1298695
  Application Handle: 187
```

As you can see in the above output, both applications are requesting and holding exclusive locks. This example shows how to identify both a deadlock event and the applications involved. You could potentially correct this issue by adjusting the lock mode of the application that is by adjusting the degree of how the data is locked. This is referred to as the isolation level. The *DB2 Administration Guide* is an excellent reference for different isolation levels and their effect on concurrency.

Sorting

Background

Queries often require a sort operation, for example in the following circumstances:

1. No index exists that would satisfy the sort order
2. An index exists, but sorting is determined to be more efficient
3. During creation of an index, assuming the indexsort (configuration parameter) is set to yes

For the purposes of performance improvement, the main area of focus for sorting is differentiating between piped and non-piped sorts. If a sort can be performed without requiring a temporary table to store a sorted list of data, it is called a piped sort. A non-piped sort requires a temporary table in order to return. A piped sort will always perform better than a non-piped sort due to decreased processing and I/O overhead.

Two parameters affect sorting:

- Sort heap size (sortheap) - amount of memory used for each sort
- Sort heap threshold (sortheapthres) - this is the total amount of available memory for sorting. When the total amount of memory used for private sorts exceeds this limit, subsequent sorts will received a reduced amount of memory. When the total amount of memory used for shared sorts exceeds this limit, no further shared sorts can be performed.

Please refer to the *DB2 Administration Guide* for more details on the database configuration parameters.

Best practices

Start by setting the sortheap equal to the average size of sort memory. This can be determined via snapshot monitoring to find the largest size of sortheap memory used.

For sheapthres, start by multiplying the sortheap values times the average number of concurrent sorts. Again, concurrent sorts can be determined via snapshot monitoring.

Counting the number of sort operations

The `Total Sorts` field in a snapshot or event monitor output is extremely useful for watching how many sorts have been executed. This parameter is tracked at the database level and the application level, thereby allowing us the flexibility to see if any particular application happens to be performing a larger number of sorts. You can then focus on that application to identify which queries are sorting more than others. For example:

1. Connect to perftut
2. Reset the monitors:
   a. `db2 reset monitor all`
3. Get a snapshot
   a. `db2 get snapshot for applications on perftut > sortstep3.out`
4. Issue the following query:
   a. `db2 "select * from rbahra.sales" > out.txt`
5. Get a snapshot
   a. `db2 get snapshot for applications on perftut > sortstep5.out`
6. Issue the following query:
   a. `db2 "select * from rbahra.sales order by sales" > out.txt`
7. Get a snapshot
   a. `db2 get snapshot for applications on perftut > sortstep7.out`
8. Issue the following query:
   a. `db2 "select * from rbahra.sales order by sales" > out.txt`
9. Get a snapshot
   a. `db2 get snapshot for applications on perftut > sortstep9.out`
10. Issue the following query:
    a. `db2 "select * from rbahra.sales" > out.txt`
11. Get a snapshot
    a. `db2 get snapshot for applications on perftut > sortstep11.out`

The above sequence of queries represents an application issuing queries against the database. Using the snapshots and tracking the Total Sorts statistic, you can see that certain queries used a sort and others did not. If the Total Sorts field remains unchanged, a sort did not occur; if it changes then we know that the previous query performed a sort. In this example, you should see that sortstep3.out indicated a total of zero sorts had occurred because you had reset the monitor data. The snapshot called sortstep5.out should show a Total Sorts value of 0, sortstep7.out will show 1 sort, sortstep9.out will show two and sortstep11.out will show two. This indicates that the query issued at step 4 did not perform a sort, the queries at steps 6 and 8 performed a sort, and the query at step 10 did not perform a sort.

From here, you could then move onto finding ways to eliminate the sort (to be discussed in the Optimizer section of this tutorial).

Sort overflows

This item can be tracked via the Sort Overflows field, available at the application and database level. This is really just an extension of tracking the total number of sorts. A large number of sort overflows could cause significant increases in query run times.

Let's try an example to capture details on sorts that overflow:

1. Connect to perftut
2. Reset the monitor
    a. `db2 reset monitor all`
3. Issue the following query:
    a. `db2 "select * from rbahra.sales order by sales" > out.txt`
4. Obtain an application snapshot:
    a. `db2 get snapshot for applications on perftut > sortsnap.out`
5. View the output file sortsnap.out.

Focus on the `Sort Overflows` field. We should see a non-zero value. From here you see that the query in the above example performed a sort that spilled to disk. The Total Sort Time field will let you know how long the sort took. If you find that the ratio of Sort Overflows to Total Sorts is quite high, it may be necessary to focus on sortheap and try to make more memory available for a sort operation.

Tracking and indicators of I/O contention

Tracking and indicators of I/O contention

If you find that items such as locking, buffer pool hit ratios, degree of sorting of no major concern, it's possible that the I/O subsystem needs to be reviewed. The **iostat** can help identify this. Here is an example of IOSTAT output:

```
tty:      tin         tout   avg-cpu:  % user    % sys     % idle    %
iowait
          0.4         38.6                5.5      6.5       2.8
85.1

Disks:        % tm_act     Kbps      tps     Kb_read      Kb_wrtn
hdisk1           0.1       1.2       0.2     2029368     10049580
hdisk3           0.1       1.3       0.1     5349500      7472208
hdisk0           1.7      10.6       1.8    20896559     84776678
hdisk2           0.0       0.0       0.0           0            0
hdisk6          43.5      27.9       1.3  1122139024   1256198880
hdisk7           0.2       5.7       0.3    17206032     39197748
hdisk8          39.4      33.9       0.2  2314800744   2124324356
```

The above output shows that two out of eight devices are extremely busy, as shown in the tm_act columns. A %tm_act value greater than 35% suggests a potential I/O bottleneck. In a situation such as this, you would be advised to split the table spaces across multiple devices. In fact, this is such a simple and quick utility that on first report of a performance problem, consider collecting iostat output immediately.

Access plans and the query optimizer

Introduction to the DB2 query optimizer

DB2 comes with a very powerful query optimization algorithm. This cost based algorithm will attempt to determine the cheapest way to perform a query against a database. Items such as the database configuration, database physical layout, table relationships, data distribution are all considered when finding the optimal access plan for a query.

When a query is issued against the database, several major events occur:

- Generation of query graph model - breaks down the query and isolates items such as tables, predicates, relationships between objects and so on. The goal is to start the transformation of the user-readable SQL into internal DB2 structures that are easier to process.
- Query rewrite phase - takes the original query and breaks it down to a more "DB2 friendly" layout. Items such as table relationships and redundant predicates are identified and exploited in order to improve query performance.
- Query optimization - goes through a series of tests that will examine the cost effectiveness of performing certain access operations. A relational operation such as table scans versus index scans, or different join techniques are considered and assigned a cost, with the cheaper operation emerging as the winner.

Access strategy

The optimizer considers many items when generating the most efficient access plan. These include:

- I/O device characteristics
- SQL statement
- Sort heap
- Catalog statistics
- Application workload
- Index definitions
- CPU capability
- Buffer pool and associated memory sizes
- Locking and concurrency capabilities>

Effectively designing and tuning a system is key to improved access plans - and faster query run times.

Optimization level

When an SQL query is compiled by DB2, a number of optimization techniques are used. Using more optimization techniques increases compile times and potentially system resource usage. Depending on the nature of the environment, it may be necessary to limit the number of techniques applied to optimizing the query. This adjustment can be performed via the DFT_QUERYOPT configuration parameter. The optimization level can be set to one of the following values:

0 - use minimal optimization
1 - use a degree of optimization roughly comparable to older versions of DB2, plus some additional techniques not used on these older versions
2 - use feature of optimization level 5, but simplified join techniques
3 - perform a moderate amount of optimization, similar to DB2 for MVS
5 - use a significant amount of optimization, with heuristic rules -- default setting
7 - use a significant amount of optimization without heuristic rules
9 - use all available techniques

If you know that the statements being executed within an application are relatively simple, for example that there are few joins, you may not want to use too many resources to optimize for higher speed. That is, you may find that the run-time differences between using a lower optimization level versus a higher one are minimal. If you have a case where extremely complex queries are being executed, a higher optimization level may be desirable, as more potentially cost effective access paths will be considered.

Update statistics

The DB2 optimizer bases its decisions based on database statistics that are stored in the system catalog. As such, it is good practice to ensure the statistics are current for the database.

RUNSTATS is a DB2 utility that allows you to update table statistics. The command line syntax for this can be found in the *DB2 Command Reference*.

Viewing the access plan

The optimizer allows us to view what plan it chose to use. First we have to create the EXPLAIN tables for the database, as follows:

```
db2 connect to perfpd
db2 -tvf $HOME/sqllib/misc/EXPLAIN.DDL
```

This will create EXPLAIN tables, which will then be populated by the optimizer when a request to capture the access plan is made. Using the **db2exfmt** tool, you can extract this information and generate a copy of the access plan used by the optimizer. To do so, let's try generating the access plan for the <query name> query:


```
db2 connect to perfpd
db2 set current explain mode explain
db2 "select * from rbahra.sales"
db2 set current explain mode no
db2exfmt -d perfpd -g TIC -w -1 -s % -n % -o db2exfmt.out
```

If you view the contents of db2exfmt.out, you will see a copy of the access plan that DB2 used when you ran the select query. Let's look at what each section shows us.

Structure of an access plan

The output of db2exfmt shows a lot of information about the database and the query. The first section provides configuration details:

```
Database Context:
----------------
        Parallelism:            Intra-Partition Parallelism
        CPU Speed:              1.054902e-06
        Comm Speed:             0
        Buffer Pool size:       210000
        Sort Heap size:         776
        Database Heap size:     4096
        Lock List size:         1076
        Maximum Lock List:      6
        Average Applications:   50
        Locks Available:        7295
```

This section is useful for keeping track of what settings were used for the buffer pool, sort heap and so on when the query was run. The next major section provides package details:

```
Package Context:
---------------
        SQL Type:               Dynamic
        Optimization Level:     5
        Blocking:               Block All Cursors
        Isolation Level:        Cursor Stability

---------------- STATEMENT 1  SECTION 201 ----------------
        QUERYNO:                1
        QUERYTAG:               CLP
        Statement Type:         Select
        Updatable:              No
        Deletable:              No
        Query Degree:           1
```

You can see the section number, and where the query was issued from (QUERYTAG). This is extremely useful when taking event monitor outputs and matching sections from the event monitor to the access plan. This allows us to better track a particular query and its related events as seen in the monitoring tools.

The next section shows us the original query and its optimized version:

```
Original Statement:
------------------
select count(distinct items)
from perfdb.sales
where items in
   (select id from perfdb.description)


Optimized Statement:
-------------------
SELECT Q4.$C0
FROM
   (SELECT COUNT(Q3.$C0)
   FROM
      (SELECT DISTINCT Q2.ITEMS
      FROM PERFDB.DESCRIPTION AS Q1, PERFDB.SALES AS Q2
      WHERE (Q2.ITEMS = Q1.ID)) AS Q3) AS Q4
```

The rewritten query shows us how DB2 interpreted the query and what relationships between objects it was able to find. The next section shows us a fragment of the access plan:

```
RETURN
(   1)
 Cost
  I/O
  |
   1
GRPBY             <- groupby to handle the distinct
(   2)
449.438
14.2765
  |
  15
LMTQ              <- table queue due to SMP requirements
(   3)
449.434
14.2765
  |
383.332
MSJOIN            <- Merge Scan Join for Q2.ITEMS=Q1.ID
(   5)
448.972
14.2765
/---+---\
```

And the remaining two sections provide more detail on the access plan operations and objects used by the query.

An access plan operator

To better understand access plans, you need to first understand these basic components:

```
Cardinality
<PLAN OPERATOR>
Cost
```



```
I/O Cost
```

The card or cardinality represents the number of rows that DB2 had estimated to be the output of the operator. The cost represents the CPU cost of this and previous operations, and IO cost represents the cost of the operator on the system's IO subsystems. Taking an example from the previously generated plan, you see that:

```
383.332     <- Cardinality
MSJOIN      <- Plan Operator
(   5)
448.972     <- Cost
14.2765     <- I/O Cost
/---+---\
```

A few of the plan operators you may see include:

SORT - indicates that a sort took place
NLJN - represents nested loop join technique used by DB2
TBSCAN - indicates that the optimizer chose to perform a table scan to either apply a predicate or find data
ISCAN - index scan, similar to a table scan but applied against an index

Tracking cardinality - how to measure its accuracy

You may often find yourself reviewing an access plan that suggests that only a few rows will be returned. You need to examine and evaluate the cardinality estimates to determine how accurate these really are. For example:

```
                100
               FETCH
      /-------------+-----------\
      100                       perfdb
      ISCAN                     Table1
      |                         1000
Index: perfdb
      Index1
      1000
```

The above plan fragment indicates that the FETCH operation will return an expected 100 rows. If you suspect that more than 100 rows should be returned (or less… the principle is the same), you can determine and compare the "true cardinality" with the estimated. To do so, you need to first isolate the predicates applied at the ISCAN. You can find these in the detailed operator section of the access plan (this has been reduced to show only the important portions):

```
        3) IXSCAN: (Index Scan)
                Cumulative Total Cost:          30.9743
                Cumulative CPU Cost:            149356
                Cumulative I/O Cost:            1
                Cumulative Re-Total Cost:       4.15582
                Cumulative Re-CPU Cost:         103895
                Cumulative Re-I/O Cost:         0
                Cumulative First Row Cost:      30.9054
                Estimated Bufferpool Buffers:   2

                Predicates:
                ----------
                2) Start Key Predicate
                        Relational Operator:            Equal (=)
                        Subquery Input Required:        No
                        Filter Factor:                  0.1

                        Predicate Text:
                        --------------
                        (Q2.column1 = 5)
```

Taking the predicate applied here, you can formulate an SQL statement to test the validity of the cardinality estimate:

```
db2 select count(*) from table1 where column1=5
```

If you find that the output of the above query generates a value significantly different than the estimated value, the statistics may have become stale and a RUNSTATS may be necessary.

You should try the following example:

```
db2 connect to perftut
db2 set current explain mode explain
db2 "select * from rbahra.org where deptnumb=20"
db2 set current explain mode no
db2exfmt -d perftut -1 -g TIC -o cardexf1.out
```

The resulting access plan should show a RETURN with an estimated cardinality of 40. Now formulate the select statement to determine the true card. Hint: look at the operator details of TBSCAN 2.

When you issue this select statement you should see that the true row count is 1! Update the statistics for rbahra.org, and rerun the test. You should now see the correct cardinality.

Tracking sorts that spilled to disk

Another useful technique of isolating query bottlenecks is to determine if a sort spilled. This is an extension of the techniques developed for verifying cardinalities. Examine the change in operator costing. Here is an example:


```
3.65665e+07
  TBSCAN
  (   15)
6.87408e+06
1.45951e+06
    |
3.65665e+07
  SORT
  (   16)
6.14826e+06
1.30119e+06
    |
3.65665e+07
  TBSCAN
  (   17)
2.00653e+06
1.14286e+06
```

You see that from the SORT(16) operator to the TBSCAN(15) operator, the cost went from 6.14e6 to 6.87e6. This jump in COST for the operation directly following a sort indicates that the SORT has spilled to disk, and has created temporary tables in order to handle overflow rows. To determine the estimated number of pages expected to spill, jump to the operator details section of the plan:

```
15) TBSCAN: (Table Scan)
                .
                .
                .
Estimated Bufferpool Buffers:    163976
```

The estimated buffer pool buffers indicate that 163976 pages were expected to spill. Possible corrections include increasing sortheap or defining an index.

Try the following example:

```
db2 connect to perftut
db2 set current explain mode explain
db2 "select * from rbahra.sales order by sales"
db2exfmt -d perftut -1 -g TIC -o sortexf1.out
```

Concentrate on the SORT(3) and TBSCAN(2) operator:

```
124272
TBSCAN
(   2)
46558.2
 4067
  |
124272
SORT
(   3)
40710.4
 2670
```

The above TBSCAN(2) shows an increase in cost. Viewing the operator details, you should see:

```
2) TBSCAN: (Table Scan)
              Cumulative Total Cost:          46558.2
              Cumulative CPU Cost:            8.30099e+08
              Cumulative I/O Cost:            4067
              Cumulative Re-Total Cost:       5890.99
              Cumulative Re-CPU Cost:         2.98685e+08
              Cumulative Re-I/O Cost:         1397
              Cumulative First Row Cost:      41142.4
              Estimated Bufferpool Buffers:   1397
```

This shows that this sort is expected to spill to disk. One technique to correct this is to either define an index on the SORTKEY or increase SORTHEAP.

The SORTKEY can be identified from the SORT operator details:

```
SORTKEY : (Sort Key column)
          1: SALES(A)
```

Try increasing the SORTHEAP to 2500 pages and generate the resulting access plan. Does the table scan above the sort show a spill? In some cases increasing SORTHEAP may not be possible, for this scenario you should consider defining an index based on the SORTKEY.

Tuning practices

Introduction

This section briefly covers some strategies and techniques for performance problem debugging.

Environmental controls

This applies more to database design considerations but is an important step to ensuring a better understanding of the database's role. When designing a database, you should always keep in mind what the application workload expectations are, the nature of the queries (simple selects in high frequency or low-frequency extremely, complicated SQL operations) and the future demands of the system.

Accurate modeling of expected data - both in the volume of data and the type of data is and important factor in predicting query access plans. As discussed, the DB2 query optimizer uses statistics to make decisions. Generating data models that follow expected statistical patterns is one step in providing insight into what types of access plans to expect.

Accurate data modeling can also be extended to monitoring run-time performance and allows for testing of potential applications in "close to real-world" environments. This can help identify existing points of contention and even expose potential issues to expect in the future.

Make incremental changes

In attempting to address performance issues in a highly tuned environment, it is important to remember that changes need to be performed in steps and within reason. If for example, you find that application concurrency is a concern, it is not always desirable to adjust the maxlocks and locktimeout values at the same time. Consider adjusting just one of the parameters, measure the effectiveness of the change, and determine if the change was useful. Modifying many parameters at once may produce excellent results or poor results; either way, you won't know which parameters made the difference.

Tracking the database

Keep records! To simplify, just keep old snapshots, event monitor or access plan outputs available. Then when you see a performance hit, you're better equipped with measurements from when things ran well. A side-by-side comparison is much easier than trying to piece things together when the things go wrong.

Conclusion

Wrap up

You've been introduced to a myriad of topics and techniques. This tutorial is not an exhaustive examination of debugging performance issues, but you should now feel better prepared to examine performance issues. Regardless of the nature or behavior of a user's complaint, you can start the investigation.

For more information

The *DB2 Administration Guide*, and *System Monitoring Guide and Reference* are excellent resources of more detailed information on the topics covered here.

Feedback

Please take a moment to give us your thoughts on this tutorial by completing the feedback form on the DB2 Technical Support site at http://www.ibm.com/software/data/db2/udb/winos2unix/support