This article explains step-by-step techniques that can be followed to monitor and tune an IBM® DB2® Universal Database™ (UDB) database server. Using the supplied sample Java™ program "PERFORMER," you can learn these hands-on techniques, and experiment with various scenarios on your own system, using the Java program to simulate a workload executing SQL against a database. Many factors can affect the performance of a database server. This article focuses on how to tune some of the important DB2 UDB configuration parameters, as well as steps to capture and fix "bad queries."

Introduction

Performance is one of the most important aspects when you are working with database systems. There are three main areas that affect your system's performance: CPU usage, I/O utilization, and memory usage, as you can see in Figure 1:

Figure 1. Areas affecting performance

Depending on your system resources, there will always be some trade-offs between CPU, memory, and I/O in your tuning decisions. You will need to balance these three areas to tune for optimal performance. We'll discuss some ways to achieve this.
Understanding the database workloads is also critical for effectively configuring your database for optimal performance. There are three types of database workloads:

- Online Transaction Processing (OLTP), consisting of many transactions of varying complexity, mostly small. OLTP transactions include selects, inserts, updates, and deletes, which normally complete in seconds or sub-seconds
- Decision Support Systems (DSS), usually select-only transactions with large queries accessing a considerable amount of data
- A mix of the two above.

General tuning tips are the same regardless of the database workload, but differences do exist between OLTP and DSS workloads. This article addresses the monitoring and tuning tips in OLTP environments only.

Many factors can affect the performance of a database server, such as hardware system design, database object design, data storage management, application design, and so forth. This article focuses on how to tune DB2 UDB configuration parameters, as well as steps to capture and fix "bad queries."

The examples in this article are specific to DB2 UDB Enterprise Server Edition Version 8.2 running on Microsoft Windows® platforms. However, the concepts and information provided are relevant to any platform.

**The "PERFORMER" sample Java program and the "BANK" database**

In this article, we provide performance examples using a database we call BANK, and a simple stand-alone Java program we call PERFORMER. The files and instructions required to create and populate the BANK database and to run PERFORMER can be found in the download section of this article. It should take you at most five minutes to have the environment ready.

For simplicity, the BANK database consists only of two tables, account and auditlog. The PERFORMER program is a JDBC™ application that uses the Type 2 Universal driver and establishes 10 connections to BANK as soon as it is invoked. The interface is very straightforward. Simply specify the duration for your test and click on Run. When the program executes, the same transaction with random parameters is executed over and over in each of the 10 connections. As you tune the database, the Number of completed transactions field should increase from one test to another. This is your main objective.

The transaction issued by PERFORMER consists of the following queries:

- SELECT NAME, BALANCE FROM ACCOUNT WHERE ACCT_ID = <value>
- UPDATE ACCOUNT SET BALANCE = ? WHERE ACCT_ID = ?
- INSERT INTO AUDITLOG VALUES (?, ?, ?, ?, ?)

We encourage you to work with PERFORMER (in a TEST environment!) as you read through the article.
Establishing a baseline

Before tuning a database, you should always establish a baseline. For this particular scenario, let's keep track of the memory consumption, and the number of completed transactions when the database is not tuned. To start with a clean environment, perform the following:

1. Close any open windows.
2. Open the DB2 command window and move it to the bottom center of your desktop. Then issue this command: `db2stop force`
3. Open the Windows Task Manager and put it on the top right corner of your desktop. Have the Performance tab selected.
4. From the DB2 command window, issue: `db2start`. At the same time, see how much memory usage increases in the task manager. Starting an instance consumes some memory, but not much.
5. From the DB2 command window issue: `db2 list applications`. You should receive a message indicating no data is being returned by the Database System Monitor.
6. Start the PERFORMER program by executing the file performer.bat, and see how the memory increases in the task manager. As mentioned earlier, PERFORMER establishes 10 connections as soon as it is invoked. When the first connection is established against a database, the database memory is allocated. In addition, each connection is associated to a DB2 agent which also incurs some memory cost. Move the PERFORMER window to the top left corner.
7. From the DB2 command window, issue: `db2 list applications`. You should now see the ten connections to the BANK database by the java.exe application (PERFORMER).

Your desktop should look as shown in Figure 2. Keep this window arrangement while reading the article.
To establish a baseline, run PERFORMER for 10 seconds, five to eight times, and write down your results. You will note that without any tuning, the number of completed transactions improves from one run to another, but eventually it reaches a limit. This is normal. During the first runs, not much information has been cached to memory; therefore, the first runs pay the I/O price of bringing some of the pages to memory. In tuning exercises, the phrase "priming the database" refers to the first run(s) to allocate all memory and bring some of the data pages to memory. These runs are normally not included in the final performance result numbers.

At this point we have the baseline for completed transactions when the database is not tuned. You should now take a full DB2 snapshot; however, we have not discussed this topic yet. Therefore, this is something you may want to try after reading the entire article, repeating this exercise from the beginning.

**DB2 UDB monitoring**

Database system monitoring is a vital part of tuning a database server. A collection of monitoring data serves as a baseline that can be used to compare current versus past performance. This allows for easier and faster detection of a performance problem. Monitoring data also helps you to understand the effects of changes to parameters and applications, and allows for the analysis of trends as the system grows.
Overview of techniques for monitoring the database server

DB2 UDB provides several tools that can be used to monitor database server activities, or to analyze how an SQL statement accesses data; each serves a different purpose. The different monitoring tools are shown in Table 1.

Table 1. Overview of DB2 UDB monitor tools

<table>
<thead>
<tr>
<th>Monitor tool</th>
<th>Monitor information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snapshot monitor</td>
<td>Captures a picture of the state of database activity at a particular point in time (the moment the snapshot is taken)</td>
</tr>
<tr>
<td>Event monitor</td>
<td>Logs data as specific database events occur, such as statement execution, transaction completion, or when an application disconnects. The STATEMENT and DEADLOCK event monitors are commonly used in performance tuning practice.</td>
</tr>
<tr>
<td>SQL explain facility</td>
<td>Captures information about the access plan and environment of SQL statements, that is, how individual SQL statements are executed to access the data</td>
</tr>
<tr>
<td>db2batch</td>
<td>Monitors the performance characteristics and execution duration of SQL statements. It reads SQL statements from either a flat file or standard input, dynamically prepares and describes the statements, and returns performance benchmark information, such as the SQL statement Prepare Time, Execute Time, and Fetch Time, and so on</td>
</tr>
</tbody>
</table>

In this article we use the snapshot monitor and the SQL explain facility.

Snapshot monitor

The snapshot monitor collects information at various levels, as shown in Table 2:

Table 2. Snapshot monitor levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Information captured</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database manager</td>
<td>Captures statistics for the active database manager instance</td>
</tr>
<tr>
<td>Database</td>
<td>Provides general statistics for all active databases on the current database partition</td>
</tr>
<tr>
<td>Application</td>
<td>Provides information about all active applications that are connected to a database on the current database partition</td>
</tr>
</tbody>
</table>

Within each database, the snapshot monitor collects information based on the functional group levels shown in Table 3.

Table 3. Snapshot functional group levels

<table>
<thead>
<tr>
<th>Functional Group Level</th>
<th>Information captured</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer pool activity</td>
<td>Number of reads and writes, time taken</td>
</tr>
<tr>
<td>Locks</td>
<td>Number of locks held, number of deadlocks</td>
</tr>
<tr>
<td>Sorts</td>
<td>Number of heaps used, overflows, sorts performance</td>
</tr>
<tr>
<td>SQL statements</td>
<td>Start time, stop time, statement identification</td>
</tr>
<tr>
<td>Table activity</td>
<td>Measure activity (rows read, rows written)</td>
</tr>
</tbody>
</table>
While there is some basic information collected for each of the functional group levels shown in Table 3 by default, it is possible to collect more detailed statistics at each level by turning on a snapshot monitor switch. Since monitoring involves overhead, you should only turn on the monitor switch that matters most in your monitoring tasks. On the other hand, if you are working on a test system, we recommend you turn on all the monitor switches while tuning your system.

The monitor switches are turned on or off at either instance or application level by using the UPDATE DBM CFG or UPDATE MONITOR SWITCHES commands, respectively. When a monitor switch is turned on at the application level, like the DB2 Command Window, the monitor will only be applicable to that particular session. For example, to turn on the monitor switches of BUFFERPOOL, SORT, and STATEMENT at the application level, issue the following command from a DB2 command window:

```
% db2 update monitor switches using BUFFERPOOL ON SORT ON STATEMENT ON
```

Since you should be working with PERFORMER in a test system, for the sake of simplicity (so you don't have to keep track of which session has the monitors on), turn all the monitors on at the instance level by issuing the following commands from the DB2 command window, as shown in Listing 1. (The TIMESTAMP and HEALTH MONITOR switches are ON by default):

**Listing 1. Commands to turn on all the monitor switches at the instance level**

```
% db2 update dbm cfg using DFT_MON_BUFPOOL   ON
% db2 update dbm cfg using DFT_MON_LOCK      ON
% db2 update dbm cfg using DFT_MON_SORT      ON
% db2 update dbm cfg using DFT_MON_STMT      ON
% db2 update dbm cfg using DFT_MON_TABLE     ON
% db2 update dbm cfg using DFT_MON_TIMESTAMP ON
% db2 update dbm cfg using HEALTH_MON        ON
```

You may also need to increase the monitor heap size, as this is the memory area used to collect monitor information. The following command is recommended:

```
% db2 update dbm cfg using MON_HEAP_SZ 1024
```

The above database manager configuration changes require a restart of the instance in order to take effect.

To find out the current monitor switch settings, issue:

```
% db2 get monitor switches
```

The output of this command is shown in Listing 2.
Listing 2. Output of the get monitor switches command

<table>
<thead>
<tr>
<th>Monitor Recording Switches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer Pool Activity Information (BUFFERPOOL) = ON 07/27/2005 09:48</td>
</tr>
<tr>
<td>Lock Information (LOCK) = ON 07/27/2005 09:48</td>
</tr>
<tr>
<td>Sorting Information (SORT) = ON 07/27/2005 09:48</td>
</tr>
<tr>
<td>SQL Statement Information (STATEMENT) = ON 07/27/2005 09:48</td>
</tr>
<tr>
<td>Table Activity Information (TABLE) = ON 07/27/2005 09:48</td>
</tr>
<tr>
<td>Take Timestamp Information (TIMESTAMP) = ON 07/27/2005 09:48</td>
</tr>
<tr>
<td>Unit of Work Information (UOW) = ON 07/27/2005 09:48</td>
</tr>
</tbody>
</table>

After the proper monitor switches are on, you can use the GET SNAPSHOT command from the same DB2 command window to collect monitoring statistics information. As an example, try using the BANK database to take a snapshot. From the DB2 command window, issue the commands as shown here:

% db2 connect to bank
% db2 select * from sysibm.sysdummy1
% db2 get snapshot for all on bank

Note that the SELECT statement above is captured by the snapshot. We discuss snapshots in more detail in the next sections below.

SQL explain facility

The SQL explain facility provides detailed information about the access plan that the query optimizer chooses for an SQL statement. This information is stored in EXPLAIN tables, and can later be formatted using tools like Visual Explain, db2expln, dynexpln, and db2exfmt to present it in a visual-friendly manner.

EXPLAIN tables may be created automatically the first time you use Visual Explain. If they have not been created, you can manually create them as follows:

% cd <db2 install path>\sqllib\misc
% db2 connect to bank
% db2 -tvf EXPLAIN.DDL

In this article, we use the db2exfmt facility. For example, to explain a dynamic SQL statement using db2exfmt, follow these steps from the DB2 command window:

% db2 connect to <database_name>
% db2 set current explain mode explain
% db2 -tvf <Input file with an SQL statement ended with a semicolon>
% db2 set current explain mode no
% db2exfmt -d <dbname> -g TIC -w -1 -n % -s % -# 0 -o <output file>

The output of the db2exfmt facility includes the information shown in Table 4:

Table 4. Overview of the db2exfmt output

<table>
<thead>
<tr>
<th>Section name</th>
<th>Content</th>
</tr>
</thead>
</table>

DB2 UDB performance tuning scenarios: Part 1, DB2 UDB OLTP tuning illustrated with a Java program
### Overview
- DB2 UDB version and release level, as well as the date and time when the explain tool was run

### Database context
- Configuration parameters that the optimizer takes into account to determine the access plan with the least resource cost, including Parallelism, CPU Speed, Communication Speed, Buffer Pool size, Sort Heap size, Database Heap size, Lock List size, Maximum Lock List, Average Applications, and Locks Available

### Package context
- SQL type (dynamic or static), optimization level, isolation level, as well as the degree of intra-parallelism used by the statement

### Original statement
- SQL statement as invoked by the application

### Optimized statement
- The rewritten version of SQL statement transformed by the optimizer from the original statement with the same query result, but allowing optimal performance

### Access plan
- The least expensive path that allows DB2 UDB to access the data in order to resolve the SQL statement

### Operator descriptions
- It shows what is happening at each stage (operator) of the access plan

### Objects used in access plan
- Tables and/or indexes used in the access plan.

We provide more examples about db2exfmt in sections below.

## Tuning the DB2 UDB configuration parameters

DB2 UDB "out of the box" parameter values are based on a system that uses 256 MB of RAM and a single disk. If you have a larger system, you need to modify these parameters to best take advantage of your system resources. You can determine a good starting point for tuning your configuration by using the Configuration Advisor, which recommends database parameter values based on your system resources. To run the configuration advisor, use the `autoconfigure` command, or invoke it through the Control Center, by right clicking on the desired database to configure, and choosing `Configuration Advisor`.

For illustration purposes, we do not run the Configuration Advisor in our examples, but we manually tune different parameters. The configuration parameters described here include:

- Buffer pools
- Asynchronous page cleaners and I/O servers
- Sorts
- Log buffer size
- Grouping COMMIT statements

### Buffer pools

Buffer pools act as the "in memory" work area for the database to perform most data manipulation (except large objects and long field data) during all transactional activities such as reads, writes, updates, and deletes. Each database requires at least one buffer pool. For a database that has table spaces of more than one page size, additional buffer pools with matching page sizes need to be created.

Buffer pool contention can be a significant factor in database performance. If the buffer pool is large enough to keep the required data in memory, less disk activity will occur. Conversely, if the
buffer pool is not large enough, the overall performance of the database can be severely curtailed, and the database manager can become I/O bound as a result of the high amount of disk activity required by the applications.

**Configuring**

To determine the buffer pool size in the database, take a look at Listing 3.

**Listing 3. Determining buffer pool size**

```
% db2 connect to bank
% db2 "select BPNAME,NPAGES,PAGESIZE from SYSCAT.BUFFERPOOLS"

<table>
<thead>
<tr>
<th>BPNAME</th>
<th>NPAGES</th>
<th>PAGESIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBMDEFAULTBP</td>
<td>250</td>
<td>4096</td>
</tr>
</tbody>
</table>
```

1 record(s) selected.

The output shows that the BANK database has a single buffer pool (IBMDEFAULTBP) defined, which has 250 pages (NPAGES) with a 4096-byte page size (PAGESIZE). The default size for IBMDEFAULTBP on Windows is 250 pages, while on UNIX® platforms, it's 1000 pages. When NPAGES has value of -1, the buffer pool size is determined by the BUFFPAGE parameter in the database configuration. For example,

```
% db2 get db cfg for bank
...
Buffer pool size (pages) (BUFFPAGE) = 250
```

**Monitoring**

As the buffer pool is intended to hold pages in memory for data manipulation required by the database server versus reading the pages from disk, an important measurement of the buffer pool's effectiveness is to see how frequently pages requested are already in the buffer pool. The buffer pool hit ratio measures such effectiveness, and it can be calculated as follows:

\[
\left(1 - \frac{(\text{Buffer pool data physical reads} + \text{Buffer pool index physical reads})}{(\text{Buffer pool data logical reads} + \text{Buffer pool index logical reads})}\right) \times 100\%
\]

The closer the ratio is to 100 percent, the lower the frequency of disk I/O, and therefore the less overhead of reading data.

A common way to monitor the buffer pool activities is using the buffer pool snapshot as follows:

(Note: make sure that buffer pool monitor switch is ON. Otherwise, most of the buffer pool statistics are not collected during snapshot monitoring.)

```
% db2 get snapshot for bufferpools on <database_name>
```

The important elements of the buffer pool snapshot that should be focused on are below:

- **Buffer pool data logical reads** = 16359
- **Buffer pool data physical reads** = 209
- **Buffer pool index logical reads** = 90
- **Buffer pool index physical reads** = 52
The above example shows a good buffer pool hit ratio of 98.4%.

Examples

Launch PERFORMER from the command prompt:

```
% performer
```

Figure 3 shows the number of transactions completed with one buffer pool of 250 4K pages: (Note: Performance results may differ on your system.)

**Figure 3. Performance results with one buffer pool of 250 4K memory pages**

```
After the test run is done, obtain a buffer pool snapshot as follows:
% db2 get snapshot for bufferpools on bank
Buffer pool data logical reads = 183925
Buffer pool data physical reads = 273548
Buffer pool index logical reads = 82
Buffer pool index physical reads = 52

The buffer pool hit ratio = (1- (273548 + 52) / (183925 + 82)) * 100% = 48.69%
With the buffer pool size of 250 pages, the application test run experienced very poor buffer pool hit ratio of 48.69%.

Increase the buffer pool size online as follows:
% db2 connect to bank
% db2 "alter bufferpool IBMDEFAULTBP immediate size 12000"

Review the task manager as you issue the above command so you see how the memory usage increases.
Monitoring information is cumulative unless you reset the monitors. Since we want to obtain a new buffer pool hit ratio after changing the buffer pool size, reset all the monitors with this command prior to the next test run:
\% db2 reset monitor all

Click the "Reset" button on the PERFORMER screen, and then click "Run." With the bigger buffer pool of 12000 4K pages, the performance result improves from 43 completed transactions per 10 seconds to 66 completed transactions, as shown in Figure 4.

**Figure 4. Performance results with one buffer pool of 12000 4K memory pages**

![Figure 4. Performance results with one buffer pool of 12000 4K memory pages](image)

Take a new snapshot, and calculate the new bufferpool hit ratio as follows:
\% db2 connect to bank
\% db2 get snapshot for bufferpools on bank

Buffer pool data logical reads = 269482
Buffer pool data physical reads = 1838
Buffer pool index logical reads = 82
Buffer pool index physical reads = 50

The buffer pool hit ratio = \(1 - (1838 + 50) / (269482 + 82)\) * 100% = 99.29%

The buffer pool hit ratio is now showing excellent results. In general, a buffer pool hit ratio above 80% is considered good. If your system has a low buffer pool hit ratio, you can increase the buffer pool size further to achieve better application performance result. What happens if you increase the buffer pool size to 20,000 4K pages and run PERFORMER again? Would the performance improve?

For this particular case, the performance would likely not improve. The BANK database has an approximate size of 36 MB (A full backup of the database can provide a quick way to measure the database size). This means that the entire database could fit in memory. A size of 12000 pages (48 MB) or 20000 pages (80 MB) would not make a difference, as the entire database can also
fit in 48MB. In fact, depending on the type of SQL performed by your application, increasing the buffer pool size over a certain limit may not improve performance any further. We recommend you keep increasing the buffer pool size until you see diminishing improvements.

Compared to other memory buffers, buffer pools have the most significant impact on database performance. However, keep in mind that buffer pools are part of the database shared memory set. In 32-bit database environment, DB2 UDB has a limit on the database shared memory size of 1.75 GB on AIX®, 3.35 GB on Sun Solaris, a value between 0.75 GB and 1 GB on HP-UX, 1.75 GB on Linux, and 2 GB or 3 GB if 3 GB switch enabled in boot.ini on Windows; therefore, you need to balance the configuration of buffer pools and other database shared memory buffers. With 64-bit environments, this is not a concern.

### Asynchronous I/O servers and page cleaners

DB2 UDB encourages the asynchronous I/O access of a page read and written between the buffer pool and the disk for optimal performance.

I/O servers asynchronously read data pages from the disk into the buffer pool anticipating their need by an application: this is called "prefetching." Prefetching can improve the database performance because the pages will be found in the buffer pool when the agent accesses them, reducing the time the application waits for the page to be read from the disk into the buffer pool.

Page cleaners, on the other hand, write changed pages from the buffer pool to disk before the space in the buffer pool is required by a database agent. As a result, database agents should not have to wait for changed pages to be written out so that they might use the space in the buffer pool. This improves overall database performance. Page cleaners can be triggered by several reasons, for example, when a changed pages threshold is reached.

### Configuring

The number of I/O servers (prefetchers) for a database can be configured using the NUM_IOSERVERS database configuration parameter. In order to fully exploit all the I/O devices in the system, a good value to use is generally one or two more than the number of physical devices on which the database resides. It is better to configure additional I/O servers, since there is minimal overhead associated with each I/O server and any unused I/O servers will remain idle.

The NUM_IOCLEANERS database configuration parameter allows you to specify the number of asynchronous page cleaners for a database. Consider the following factors when setting the value for this parameter:

- **Application type**
  - If it is a query-only database that will not have updates, set this parameter to be zero (0). The exception would be if the query work load results in many TEMP tables being created (you can determine this by using the explain utility). If transactions are run against the database, set this parameter to be between one and the number of physical storage devices used for the database.
- **Workload**
Environments with high update transaction rates might require more page cleaners to be configured.

- Buffer pool sizes
  Environments with large buffer pools might also require more page cleaners to be configured.

Also keep in mind that having too many page cleaners may overwhelm the run queue on the database server and cause the significant performance degradation. Hence, as a rule of thumb, you may consider setting the number of page cleaners equal to the number of CPUs on the database server.

The changed pages threshold parameter (CHNGPGS_THRESH) determines the percentage of changed pages at which the asynchronous page cleaners should start.

**Monitoring**

As the activities of asynchronous page cleaners and I/O servers are so tightly tied with buffer pool activities, you can once again leverage the buffer pool snapshot to measure the effectiveness of the page cleaners and prefetchers.

This time the important elements of the buffer pool snapshot that should be focused on are below:

- Buffer pool data logical reads = 2700145
- Buffer pool data writes = 0
- Buffer pool index logical reads = 95
- Asynchronous pool data page reads = 85
- Asynchronous pool data page writes = 0
- Buffer pool index writes = 0
- Asynchronous pool index page reads = 0
- Asynchronous pool index page writes = 0

The prefetching activity can be validated by the amount of asynchronous versus synchronous read I/O. The asynchronous read ratio is calculated as follows:

\[
\text{Asynchronous read ratio} = \left( \frac{\text{Asynchronous pool data page reads} + \text{Asynchronous pool index page reads}}{\text{Buffer pool data logical reads} + \text{Buffer pool index logical reads}} \right) \times 100\%
\]

The small value of asynchronous read ratio could be caused by a number of reasons, such as:

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

The effectiveness of the asynchronous page cleaners is measured by the ratio of asynchronous data and index page writes. The number of asynchronous page cleaners (NUM_IOCLEANERS) can be reduced if both of the following conditions are true:

- **Buffer pool data writes** is approximately equal to **Asynchronous pool data page writes**.
- **Buffer pool index writes** is approximately equal to **Asynchronous pool index page writes**.

The parameter should be increased if either of the following conditions is true: 
• Buffer pool data writes is much greater than Asynchronous pool data page writes.
• Buffer pool index writes is much greater than Asynchronous pool index page writes.

Examples

First, to verify the current settings of the NUM_IOSERVERS and NUM_IOCLEANERS parameters:

```
% db2 get db cfg for bank
Number of asynchronous page cleaners (NUM_IOCLEANERS) = 1
Number of I/O servers (NUM_IOSERVERS) = 3
```

Reset the snapshot monitor prior to the next PERFORMER run:

```
% db2 reset monitor all
```

The buffer pool snapshot is used here to verify whether the NUM_IOSERVERS and
NUM_IOCLEANERS parameters are properly configured on your system by applying the rules
discussed earlier.

```
% db2 get snapshot for bufferpools on bank
Buffer pool data logical reads = 269482
Buffer pool data writes = 0
Buffer pool index logical reads = 82
Asynchronous pool data page reads = 1236
Asynchronous pool data page writes = 0
Buffer pool index writes = 0
Asynchronous pool index page reads = 0
Asynchronous pool index page writes = 0
```

The asynchronous read ratio = \((1236 + 0) / (269482 + 82)\) * 100% = 0.004%

The reason why the asynchronous read ratio is so small is because the table spaces in the
database are set up with only one container each so that prefetching cannot take place. Therefore,
the current setting of the NUM_IOSERVERS parameter can remain the same.

You then need to review the monitoring data of Buffer pool data writes (0) versus Asynchronous
pool data page writes (0) and Buffer pool index writes (0) versus Asynchronous pool index page
writes (0). By applying the rules discussed earlier, this health check exercise indicates that no
further tuning effort is required for the NUM_IOCLEANERS parameter.

The default value for CHNGPGS_THRESH of 60% is normally too high for OLTP workloads. A
value between 20% and 40% is better. For example, if you had a 2 GB buffer pool, when 60%
of changed pages is reached, 1.2 GB (60% of 2 GB) would be written to disk as page cleaners
are triggered. This can cause overall slow down in your system as this happens. By setting
CHNGPGS_THRESH to a lower amount like 20%, the page cleaners will be triggered more often,
but less data will be written to disk, and the slowdown may be unnoticeable by your users.

Sorts

DB2 UDB has two basic types of sorts: shared sorts and private sorts.
Shared sorts are only available when the INTRA_PARALLEL database manager configuration parameter is ON or the concentrator is enabled (that is, when MAX_CONNECTIONS is greater than MAX_COORDAGENTS). They are often used when it is desirable to have multiple subagents feeding or fetching from a sort. The memory used for shared sorts is allocated from the database shared memory set.

When both the INTRA_PARALLEL parameter is OFF and the concentrator is disabled, all sorts are private. The load and create index operations always use private sorts for index key sorting, regardless of the value of the INTRA_PARALLEL parameter. The memory used for private sorts is allocated from an agent's private memory. So a private sort can only be accessed by a single agent.

For shared sorts, the SHEAPTHRES_SHR database configuration parameter is a database-wide hard limit on the total amount of database shared memory that can be used for sorting at any one time. When the total amount of shared memory for active shared sorts reaches this limit, subsequent sorts will fail with SQL0955. If the value of SHEAPTHRES_SHR is zero, the threshold for shared sort memory will be equal to the value of the SHEAPTHRES database manager configuration parameter, which is also used to represent the sort memory.

In both shared and private sorts, the SORTHEAP database configuration parameter is the maximum number of 4 K memory pages that will be used for a single sort.

Also, sorting involves two steps:

- **A sort phase**
  A sort can be overflowed or non-overflowed. If the sorted data cannot fit entirely into the sort heap, which is a block of memory that is allocated each time a sort is performed, it overflows into a temporary table owned by the database. Sorts that do not overflow always perform better than those that do.

- **Return of the results of the sort phase**
  The return can be piped or non-piped. If sorted information can return directly without requiring a temporary table to store a final, sorted list of data, it is a piped sort. If the sorted information requires a temporary table to be returned, it is a non-piped sort. A piped sort always performs better than a non-piped sort.

### Configuring SHEAPTHRES:

- **In general SHEAPTHRES should be set to a multiple of SORTHEAP**
- **A general rule of thumb is to set SHEAPTHRES to at least 10 * SORTHEAP.**
- **For index creation with INTRA_PARALLEL ON, make sure SHEAPTHRES >= d * SORTHEAP, where d is the SMP degree.**
- **For LOAD, all index key sorting takes place simultaneously in the private memory space of one process: db2lrid in SMP, db2lfrm0 in serial. Make sure SHEAPTHRES >= n * SORTHEAP, where n is the number of indexes on the table being loaded. Also keep in mind**
that on AIX, there is approximately a max of 250MB of private virtual memory for index key sorting in LOAD (this is a memory bottleneck).

Configuring SORTHEAP:

- Increasing the value of SORTHEAP can improve sort performance drastically because more sorting will be done in memory and there will be less disk I/O.
- Underconfiguring SORTHEAP can lead to performance degradation due to increased spilling and I/O.
- Overconfiguring SORTHEAP can lead to capacity problems as SHEAPTHRES will be exceeded more quickly. Also, as SHEAPTHRES_SHR is approached (for shared sorts) or SHEAPTHRES is exceeded (for private sorts), the amount of memory allocated to new sorts will continually decrease, which can also lead to performance degradation.

Monitoring
Besides the buffer pool, sort is another significant factor that impacts the database performance. DB2 UDB monitors quite a few aspects of sort activities.

The database manager snapshot shows the following monitoring elements:
% db2 get snapshot for database manager
  Private Sort heap allocated = 0
  Private Sort heap high water mark = 80
  Post threshold sorts = 0
  Piped sorts requested = 167
  Piped sorts accepted = 167

The database snapshot shows the following monitoring elements
% db2 get snapshot for database on <database_name>
  Total Private Sort heap allocated = 0
  Total Shared Sort heap allocated = 0
  Shared Sort heap high water mark = 0
  Total sorts = 170
  Total sort time (ms) = 5015
  Sort overflows = 93
  Active sorts = 0
  Commit statements attempted = 446
  Rollback statements attempted = 24

Several key formulas are listed below for evaluating the performance of the sort activities:

Post threshold sorts is the number of sorts that have requested heaps after the sort heap threshold has been exceeded. Sorts that start after the sort heap threshold has been reached may not receive an optimum amount of memory to execute.

Post threshold sorts ratio = (Post threshold sorts / Total sorts) * 100%

If the value of Post threshold sorts ratio is high, you should either increase the sort heap threshold (sheapthres), or adjust applications to use fewer or smaller sorts via SQL query changes.
**Piped sorts ratio** = (Piped sorts accepted / Piped sorts requested) * 100%

If the value of *Piped sorts ratio* is low, you should consider increasing the sort heap threshold (sheapthres) for better sort performance.

**Sort overflow ratio** = (Sort overflows / Total sorts) * 100%

If the value of *Sort overflow ratio* is high, you should increase the sort heap (SORTHEAP), and/or the sort heap threshold (sheapthres) as well.

**Sorts per transaction** = Total sorts / (Commit statements attempted + Rollback statements attempted)

When there are three or more sorts per transaction, instead of tuning the sort heap or the sort heap threshold, you should come to the problem source to identify the poorly performing SQL statements via dynamic SQL snapshots, and add proper indexes if necessary.

**Examples**

The PERFORMER program is an OLTP application, so it doesn't have complex SELECT queries, and has minimal sorting. The example here is designed for you to review sort tuning techniques outside the PERFORMER program.

The pre-exercise steps here intentionally decrease the size of IBMDEFAULTBP back to 250 pages and SORTHEAP to 16 pages so that you can review the related performance tuning techniques in a small database environment.

% db2 connect to bank
% db2 "alter bufferpool IBMDEFAULTBP size 250"
% db2 update db cfg for bank using SORTHEAP 16
% db2 force applications all
% db2 connect to bank

To reset the snapshot monitor, issue:
% db2 reset monitor all

You can create a SQL DDL file, order_by.ddl, containing this SQL:
```
SELECT NAME,BALANCE FROM ACCOUNT ORDER BY BALANCE;
```

To trigger sorting activities within the database, repeat the following command multiple times, for example, 3 times:
% db2 -tvf order_by.ddl

The database manager snapshot can be used to determine the *Post threshold sorts* activity:

% db2 get snapshot for database manager

...  
**Private Sort heap allocated** = 0  
**Private Sort heap high water mark** = 256  
**Post threshold sorts** = 0
Piped sorts requested = 3
Piped sorts accepted = 3

There was no Post threshold sorts activity. Moreover, the piped sorts are 100% accommodated, piped sorts ratio = (3 / 3) * 100% = 100%.

You can then leverage the database snapshot to identify the sort activities at the database level:
% db2 get snapshot for database on bank
...
Total Private Sort heap allocated = 0
Total Shared Sort heap allocated = 0
Shared Sort heap high water mark = 0
Total sorts = 3
Total sort time (ms) = 1097
Sort overflows = 3
Active sorts = 0
Commit statements attempted = 4
Rollback statements attempted = 5

The sort overflow ratio = (3 / 3) * 100% = 100%. Due to the high sort overflow ratio, you need to increase the sort heap (SORTHEAP), and/or the sort heap threshold (SHEAPTHRES). The evaluation of the sorts per transaction is not necessary in this example.

Per the health check on the sort activities via snapshot monitoring, you now need to increase the sort heap (SORTHEAP), and/or the sort heap threshold (SHEAPTHRES).

To increase the SORTHEAP parameter, issue:
% db2 update db cfg for bank using SORTHEAP 400
% db2 force applications all

Upon each change, use snapshot monitoring to determine whether further sort tuning is required. If so, repeat the same steps.

**Log buffer size**
The log buffer acts as a staging area in memory to hold log records instead of having the DB2 UDB engine write each log record directly to disk.

The log records are written to disk when one of the following occurs:

- A transaction commits or a group of transactions commit, as defined by the MINCOMMIT configuration parameter
- The log buffer is full
- As a result of some other internal database manager event.

**Configuring**
The log buffer size is defined by the LOGBUFSZ database parameter. Increase the size of the log buffer area if there is considerable read activity on a dedicated log disk, or there is high disk
utilization. When increasing the value of the LOGBUFSZ parameter, you should also consider the DBHEAP parameter since the log buffer uses space from the database heap area.

**Monitoring**
You can use the database snapshot to determine whether the LOGBUFSZ database parameter is optimal by viewing the following snapshot elements:

% db2 get snapshot for database on <database_name>
Log pages read = 0
Log pages written = 6721

*Log pages read* is the number of log pages read from disk by the logger, while *Log pages written* is the number of log pages written to disk by the logger. The ratio of the number of *Log pages read* to the number of *Log pages written* should be as small as possible. Ideally, there should be no Log pages read. If you see a large number of Log pages read, it indicates that the value of the LOGBUFSZ database parameter should be increased.

**Examples**
The default size of the log buffer (LOGBUFSZ) below is normally too small for database systems with many update workloads.

% db2 get db cfg for bank
Log buffer size (4KB) (LOGBUFSZ) = 8

The PERFORMER program doesn't have many updates. So you won't exercise the LOGBUFSZ tuning tips with PERFORMER here. However, you can still try it with your own application(s). If your application happens to have a high update workload, the database snapshot monitoring may reveal a large number of log pages read. If so, you can consider increasing the LOGBUFSZ database parameter. Typically, it should be increased to a minimum of 256 pages.

**Number of commits to group**
In an environment in which many short concurrent transactions occur, by default each COMMIT statement triggers one log buffer flush to disk. As a result, the logger process frequently writes small amounts of log data to disk, with additional delay caused by log I/O overhead. Commit grouping allows the writing of the log buffer data to disk until a minimum number of commits have been requested. This feature can result in improved performance by reducing the log I/O overhead.

This grouping of commits will only occur when the value of the MINCOMMIT parameter is greater than one, and when the number of applications connected to the database is greater than or equal to the value of this parameter. When commit grouping is being performed, application commit requests can be held until either one second has elapsed or the number of commit requests equals the value of the MINCOMMIT parameter.

**Configuring**
Increase the MINCOMMIT parameter from its default value if multiple read/write applications typically request concurrent database commits. This will result in more efficient logging file I/O as it will occur less frequently and write more log records each time it does occur.
You could also sample the number of transactions per second and adjust this parameter to accommodate the peak number of transactions per second (or some large percentage of it). Accommodating peak activity would minimize the overhead of writing log records during transaction intensive periods.

**Monitoring**

The database snapshot monitor can be used to determine the number of transactions performed per second, as follows:

```
% db2 get snapshot for database on <database_name>
Last reset timestamp = 07/30/2005 15:54:22.392292
Snapshot timestamp = 07/30/2005 19:24:10.858723
Commit statements attempted = 13784
Rollback statements attempted = 134
```

The number of transactions per second:

\[
((\text{Commit statements attempted} + \text{Rollback statements attempted}) / (\text{Snapshot timestamp} - \text{Last reset timestamp}))
\]

**Examples**

The MINCOMMIT parameter can have both positive and negative impact on database performance, and needs to be set correctly.

```
% db2 get snapshot for database on bank
Last reset timestamp = 07/30/2005 19:32:45.570089
Snapshot timestamp = 07/30/2005 19:33:14.650596
Commit statements attempted = 7374
Rollback statements attempted = 1
```

The number of transactions per second = \((7374 + 1) / (19:33:14.650596 - 19:32:45.570089)\) = 254.13 transactions per second.

Considering that the PERFORMER program invokes the small number of 10 concurrent application connections only, while there are a large number of short transactions (that is, 254 transactions per second), the default value of MINCOMMIT parameter should not be changed; otherwise, those short transactions will be dramatically increased. Test it yourself!

**Database query optimizer**

DB2 UDB comes with a powerful cost-based query optimizer that determines the best strategy to access data. The DB2 UDB query optimizer always attempts to determine the cheapest way to perform a query against a database by rewriting the original query into an optimized form, generating alternative query execution plans, modeling I/O, CPU, memory, and communication usage of each alternative, and selecting the minimal cost access plan for execution.

**Updating catalog statistics**

The query optimizer uses the SYSSTAT catalog views in the database to retrieve the statistical information of the database objects and determine the best way to access the data. If current
statistics are not available, the optimizer may choose an inefficient access plan based on inaccurate default statistics.

By default for a newly created database, none of the database object statistics is collected and stored in SYSSTAT catalog views until the `runstats` command is performed. If the catalog statistics have not been populated, columns like CARD, NPAGES, FPAGES, and so on, in the SYSSTAT catalog views, would have a value of -1. An example is shown in Listing 4:

**Listing 4. Determining if RUNSTATS has been run**

```sql
% db2 connect to bank
% db2 describe table sysstat.tables
% db2 "select tabname,card,npages,fpages from sysstat.tables"
```

<table>
<thead>
<tr>
<th>TABNAME</th>
<th>CARD</th>
<th>NPAGES</th>
<th>FPAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCOUNT</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>AUDITLOG</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

It is highly recommended that you use the runstats command to collect current statistics on tables and indexes, especially if significant update activity has occurred or new indexes have been created since the last time the runstats command was executed. This provides the optimizer with the most accurate information with which to determine the best access plan. For example:

```sql
% db2 runstats on table <table_name> with distribution and detailed indexes all
```

where `<table_name>` is a fully qualified table name, that is, it includes the schema name.

Sometimes you may need to perform the runstats operation against all the tables within the database. The easiest way to do so is by using the reorgchk command:

```sql
% db2 reorgchk update statistics on table all
```

If you have trouble identifying whether your database contains up-to-date statistics on tables and indexes, you can issue the following command to verify the most recent time you performed the runstats operation:

```sql
% db2 "select name, stats_time from sysibm.systables"
```

If the runstats has not yet been run, you will see "-" for the `stats_time` column. Otherwise, it returns the time stamp of the last time that runstats was run.

We will test PERFORMER after running runstats in a section below.

**Catching bad queries**

An application can run hundreds of different SQL statements; if only one of them is incorrectly coded or not tuned to its best, the entire system’s performance can be impacted. How can you catch these bad queries?

The dynamic SQL statement snapshot provides information about the dynamic SQL statements run by your application. Follow these steps to reset the monitors, run PERFORMER, and then obtain a dynamic SQL statement snapshot:
% db2 reset monitor all
% performer (and click on the "Run" button)
% db2 get snapshot for dynamic sql on bank

Listing 5 shows part of the dynamic SQL snapshot output.

**Listing 5. The Dynamic SQL snapshot**

<table>
<thead>
<tr>
<th></th>
<th>= BANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database name</td>
<td>= C:\DB2\NODE0000\SQL00006\</td>
</tr>
<tr>
<td>Database path</td>
<td></td>
</tr>
<tr>
<td>Number of executions</td>
<td>= 1</td>
</tr>
<tr>
<td>Number of compilations</td>
<td>= 1</td>
</tr>
<tr>
<td>Worst preparation time (ms)</td>
<td>= 0</td>
</tr>
<tr>
<td>Best preparation time (ms)</td>
<td>= 0</td>
</tr>
<tr>
<td>Internal rows deleted</td>
<td>= 0</td>
</tr>
<tr>
<td>Internal rows inserted</td>
<td>= 0</td>
</tr>
<tr>
<td>Rows read</td>
<td>= 100000</td>
</tr>
<tr>
<td>Internal rows updated</td>
<td>= 0</td>
</tr>
<tr>
<td>Rows written</td>
<td>= 0</td>
</tr>
<tr>
<td>Statement sorts</td>
<td>= 0</td>
</tr>
<tr>
<td>Statement sort overflows</td>
<td>= 0</td>
</tr>
<tr>
<td>Total sort time</td>
<td>= 0</td>
</tr>
<tr>
<td>Buffer pool data logical reads</td>
<td>= 1820</td>
</tr>
<tr>
<td>Buffer pool data physical reads</td>
<td>= 1727</td>
</tr>
<tr>
<td>Buffer pool temporary data logical reads</td>
<td>= 0</td>
</tr>
<tr>
<td>Buffer pool temporary data physical reads</td>
<td>= 0</td>
</tr>
<tr>
<td>Buffer pool index logical reads</td>
<td>= 0</td>
</tr>
<tr>
<td>Buffer pool index physical reads</td>
<td>= 0</td>
</tr>
<tr>
<td>Buffer pool temporary index logical reads</td>
<td>= 0</td>
</tr>
<tr>
<td>Buffer pool temporary index physical reads</td>
<td>= 0</td>
</tr>
<tr>
<td>Total execution time (sec.ms)</td>
<td>= 0.493177</td>
</tr>
<tr>
<td>Total user cpu time (sec.ms)</td>
<td>= 0.040057</td>
</tr>
<tr>
<td>Total system cpu time (sec.ms)</td>
<td>= 0.000000</td>
</tr>
<tr>
<td>Statement text</td>
<td>= SELECT NAME, BALANCE FROM ACCOUNT WHERE ACCT_ID =14680</td>
</tr>
</tbody>
</table>

|                               | = BANK                                                             |
| Database name                 | = C:\DB2\NODE0000\SQL00006\                                       |
| Database path                 |                                                                  |
| Number of executions          | = 1                                                               |
| Number of compilations        | = 1                                                               |
| Worst preparation time (ms)   | = 0                                                               |
| Best preparation time (ms)    | = 0                                                               |
| Internal rows deleted         | = 0                                                               |
| Internal rows inserted        | = 0                                                               |
| Rows read                     | = 100000                                                          |
| Internal rows updated         | = 0                                                               |
| Rows written                  | = 0                                                               |
| Statement sorts               | = 0                                                               |
| Statement sort overflows      | = 0                                                               |
| Total sort time               | = 0                                                               |
| Buffer pool data logical reads| = 1820                                                             |
| Buffer pool data physical reads| = 1527                                                          |
| Buffer pool temporary data logical reads | = 0               |
| Buffer pool temporary data physical reads | = 0               |
| Buffer pool index logical reads| = 0                                                               |
| Buffer pool index physical reads| = 0                                                               |
| Buffer pool temporary index logical reads | = 0               |
| Buffer pool temporary index physical reads | = 0               |
| Total execution time (sec.ms) | = 1.034426                                                        |
| Total user cpu time (sec.ms)  | = 0.040058                                                        |
| Total system cpu time (sec.ms)| = 0.000000                                                        |
| Statement text                | = SELECT NAME, BALANCE FROM ACCOUNT WHERE ACCT_ID =47030          |
We have extracted three sections of the dynamic SQL snapshot corresponding to three different queries:

SELECT NAME, BALANCE FROM ACCOUNT WHERE ACCT_ID =14680
SELECT NAME, BALANCE FROM ACCOUNT WHERE ACCT_ID =47030
INSERT INTO AUDITLOG VALUES (?,?,?,?)

Note the two SELECT statements are exactly the same except for the value of ACCT_ID. The PERFORMER program was purposely coded without a parameter marker (?) to show what you should not be doing! The two SELECT statements will be treated as different queries by the DB2 UDB optimizer, and therefore each query will be compiled separately incurring an extra overhead as shown by these fields:

Number of executions = 1
Number of compilations = 1

On the other hand, if you review the INSERT statement you can see it uses five parameter markers. The fields:

Number of executions = 73
Number of compilations = 1

show that only one compilation was required, and that the same query with different values is executed 73 times.

Next, let's capture the bad SQL statement. Using operating system commands, you can redirect the output of the snapshot command to a file as follows:

% db2 get snapshot for dynamic sql on bank > snap1.txt
You can then use operating system or editor commands to look for the field "Total execution time." Sort the results in descending order, and start analyzing the query that has the highest value. For example, in our Windows system we have a UNIX simulator installed; therefore, we can use the grep command as follows:

```
% grep -i 'Total execution' snap1.txt
```

This would provide a list as shown in Listing 6:

**Listing 6. Looking for bad queries**

```
Total execution time (sec.ms)      = 0.471460
Total execution time (sec.ms)      = 1.034426
Total execution time (sec.ms)      = 0.354776
Total execution time (sec.ms)      = 0.197684
Total execution time (sec.ms)      = 0.820391
Total execution time (sec.ms)      = 0.900800
Total execution time (sec.ms)      = 0.532227
Total execution time (sec.ms)      = 0.436514
Total execution time (sec.ms)      = 0.445380
Total execution time (sec.ms)      = 0.943996
Total execution time (sec.ms)      = 0.369309
Total execution time (sec.ms)      = 41.844554
Total execution time (sec.ms)      = 1.058051
Total execution time (sec.ms)      = 0.394608
...
```

Though we did not sort this list, we immediately can see the query with the highest execution time is 41.844554. Listing 7 shows the section in the Dynamic SQL snapshot output for this query.

**Listing 7. Dynamic SQL snapshot section for a bad query**

```
Number of executions               = 77
Number of compilations             = 1
Worst preparation time (ms)        = 0
Best preparation time (ms)         = 0
Internal rows deleted              = 0
Internal rows inserted             = 0
Rows read                          = 7700000
Internal rows updated              = 0
Rows written                       = 77
Statement sorts                    = 0
Statement sort overflows           = 0
Total sort time                    = 0
Buffer pool data logical reads     = 140140
Buffer pool data physical reads    = 124493
Buffer pool temporary data logical reads = 0
Buffer pool temporary data physical reads  = 0
Buffer pool index logical reads    = 0
Buffer pool index physical reads   = 0
Buffer pool temporary index logical reads = 0
Buffer pool temporary index physical reads = 0
Total execution time (sec.ms)      = 41.844554
Total user cpu time (sec.ms)       = 2.743941
Total system cpu time (sec.ms)     = 1.061525
Statement text                     = UPDATE ACCOUNT SET BALANCE = ?, WHERE ACCT_ID = ?
```

Analyzing this output, we notice that the query has executed 77 times, so the total execution of 41.844554 is for the 77 executions. Dividing 41.844554 by 77 we get that the execution per query
is 0.5434358, so it is not as bad as it looks. Can the query be improved? Certainly. The number of rows read (7700000) versus the number of rows written (77) show that a table scan is likely happening. This is not necessarily the most expensive query, however. If we repeat the same procedure as before, we find that the query with this execution time:

```
Total execution time (sec.ms) = 1.034426
```

is actually the query that took the most time. The query in question is:

```
SELECT NAME, BALANCE FROM ACCOUNT WHERE ACCT_ID = 47030
```

and the full section for that query in the snapshot was included earlier in Listing 5. The next section will show how this query can be fixed.

In Part 2 of this article series, we describe the use of table snapshot functions that can be used to obtain this same type of information but stored in tables that you can manipulate using SQL statements.

**Understanding access plan**

You've learned how to capture the access plan of a given query using the SQL explain utility (db2exfmt) in the DB2 UDB monitoring section. This section focuses on how to understand the access plan itself so that you can identify the root cause(s) of the slow query, and then fix the problem.

The example below introduces the access plan basics by analyzing the bad SQL statement we captured in the previous section.

Create a script file, `select.ddl` containing the following SELECT statement:

```
SELECT NAME, BALANCE from ACCOUNT WHERE ACCT_ID=47030;
```

You may execute the following steps in order to generate the access plan for the above SELECT statement:

```
% db2 connect to bank
% db2 set current explain mode explain
% db2 -tvf select.ddl
% db2 set current explain mode no
% db2exfmt -d bank -g TIC -w -1 -n % -s % -# 0 -o exfmt_runstats.out
```

**Structure of the db2exfmt output**

From the `exfmt_runstats.out` file, you can see that the output of db2exfmt shows a lot of valuable information about the database environment and the query. Some important sections are listed below.

The `Database Context` section lists the configuration parameters that the optimizer takes into account to determine the access plan with the least resource cost. This is shown in Listing 8.
Listing 8. The Database Context section of a db2exfmt output

Database Context:

```
------------------
  Parallelism:     None
  CPU Speed:      9.053265e-007
  Comm Speed:     0
  Buffer Pool size: 250
  Sort Heap size: 400
  Database Heap size: 600
  Lock List size: 50
  Maximum Lock List: 22
  Average Applications: 1
  Locks Available: 1122
```

The next major section provides package details, such as whether the SQL is dynamic or static, the optimization level, as well as the isolation level. More importantly you can see the section number, and where the query was issued from (QUERYTAG: CLP). This is quite useful when you match the access plan back to the event monitor outputs. It allows us to better track a particular query and the related events. Listing 9 shows this section.

Listing 9. The Package Context section of a db2exfmt output

Package Context:

```
------------------
  SQL Type: Dynamic
  Optimization Level: 5
  Blocking: Block All Cursors
  Isolation Level: Cursor Stability
------------------ STATEMENT 1 SECTION 201 ------------------
  QUERYNO: 4
  QUERYTAG: CLP
  Statement Type: Select
  Updatable: No
  Deletable: No
  Query Degree: 1
```

Listing 10 shows shows the original query statement and its optimized version.

Listing 10. The Original and optimized sections of a db2exfmt output

```
Original Statement:
------------------
  select name,balance
  from account
  where acct_id=47030

Optimized Statement:
------------------
  SELECT Q1.NAME AS "NAME", Q1.BALANCE AS "BALANCE"
  FROM CXWANG.ACCOUNT AS Q1
  WHERE (Q1.ACCT_ID = 47030)
```

Listing 11 shows the access plan for the SELECT statement.
Listing 11. The Access Plan section of a db2exfmt output

```
Access Plan:  
---------------
Total Cost: 3420.41
Query Degree: 1

Rows
RETURN
( 1 )
Cost
I/O
| 1
TBSCAN
( 2 )
3420.41
1820
| 180000
TABLE: CXWANG
ACCOUNT
```

The remaining two sections provide details on each operator of the access plan, and the tables and/or indexes used by the query.

Access plan operator
Listing 12 shows the basic component of an access plan.

Listing 12. The basic component of an access plan

```
cardinality
<access plan operator>
( # )
cost
I/O cost
```

The cardinality represents the estimated number of rows returned from the access plan operator. The access plan operator is either an action that must be performed on data, or the output from a table or an index. The cost represents the cumulative CPU cost of this and previous operations, and the I/O cost represents the cost of the operator with I/O subsystems.

The unit of cost is the timeron. A timeron does not directly equate to any actual elapsed time, but gives a rough relative estimate of the resources (cost) required by the database manager.

Here are a few examples of the access plan operators you may see in this article:

- RETURN: Represents the return of data from the query to the user.
- FETCH: fetches columns from a table using a specific record identifier.
- TBSCAN: Retrieves rows by reading all required data directly from the data pages.
- IXSCAN: Scans an index of a table with optional start/stop conditions, producing an ordered stream of rows.

Table scan versus index scan
The optimizer normally chooses a table scan if no appropriate index has been created or if an index scan would be more costly. An index scan might be more costly when the table is small and the index-clustering ratio is low, or the query requires most of the table rows.

The earlier access plan shows that the total cost of the SELECT statement is 3420.41 timerons, which is from the table scan operator (TBSCAN) against the ACCOUNT table. Since only one row is expected in the result set from this SELECT statement, the table scan here is considered an expensive operation. Alternatively, an index scan can be attempted in order to achieve better performance.

Prior to your index creation effort, you may run PERFORMER to get a performance baseline for later comparison. In our case the number of completed transaction using a 12,000 4k page bufferpool was 66 in 10 seconds as shown in Figure 4.

Create an index on the ACCOUNT table as follows:

```
% db2 connect to bank
% db2 describe table account
% db2 "create index acct_id_inx on account (acct_id)"
```

Update the catalog statistics with the runstats command so that the optimizer can take this newly created index into account:

```
% db2 runstats on table cxwang.account with distribution and detailed indexes all
```

To evaluate the new access plan after the index creation on the ACCOUNT table issue:

```
% db2 set current explain mode explain
% db2 -tvf select.ddl
% db2 set current explain mode no
% db2exfmt -d bank -g TIC -w -1 -n % -s % -# 0 -o exfmt_index.out
```

Listing 13 shows the new access plan.

**Listing 13. Access Plan after an index was created**

```
Access Plan: 
-------------
| Total Cost: | 38.6219 |
| Query Degree: | 1 |

<table>
<thead>
<tr>
<th>Rows RETURN (   1) Cost I/O</th>
<th>1 FETCH (   2) 38.6219 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>IXSCAN TABLE: CXWANG (   3) ACCOUNT</td>
<td></td>
</tr>
</tbody>
</table>
```

DB2 UDB performance tuning scenarios: Part 1, DB2 UDB OLTP tuning illustrated with a Java program
By adding the index on the ACCT_ID_INX column, the new access plan shows a significant cost reduction from 3420.41 to 38.6219 timerons.

Figure 5 shows the significant performance improvement of PERFORMER after the index creation. From 66 transactions to 2051!:

**Figure 5. Performance results with index scan on the ACCOUNT table**

![Performance Test Setup](image)

**Summary**

This article introduced DB2 UDB performance monitoring and tuning fundamentals using a sample Java program (PERFORMER). You can apply the simple step-by-step performance tuning examples to improve the performance on your own DB2 UDB database system. Moreover, you had the opportunity to learn how to evaluate and analyze access plans and fix "bad queries." In Part 2 of this article series, we will use a more complete database scenario that builds on top of the one used in this article. We will discuss other performance monitoring and tuning techniques, and the use of the Design Advisor, the activity monitor, table snapshot functions, and other tools. Part 2 also covers OLTP, DSS, and mixed workloads.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample program and database</td>
<td>performer.zip</td>
<td>18 KB</td>
</tr>
</tbody>
</table>

(Download via [HTTP](#) | [FTP](#))
Related topics

- **IBM DB2 Universal Database Administration Guide: Performance Version 8.2**: Find more detailed information about performance topics.
- **IBM DB2 Universal Database System Monitor Guide and Reference Version 8.2**: Find more detailed information about performance monitoring topics.
- **IBM DB2 Universal Database Command Reference Version 8.2**: Obtain detail syntax about the commands discussed in this article.
- The article **SQL access to DB2 monitoring data** explains how to capture DB2 snapshots using table functions.
- The article **Tuning DB2 UDB using the statement event monitor** describes a technique for analyzing output from the DB2 UDB statement event monitor in order to tune your SQL.
- The article **Best practices for tuning DB2 UDB V8 and its databases** is a handbook for high performance, describing many performance tuning techniques.

© Copyright IBM Corporation 2005
**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)