

Managing What Matters Most for IBM DB2 UDB Distributed Systems Performance

“Measure. Improve. Repeat.”

3 Steps for achieving
Breakthrough Performance Results
with IBM DB2 UDB V8.x

Updated February 1, 2006

Scott Hayes
President & CEO, Database-Brothers Inc.
IBM DB2 GOLD Consultant
Scott.Hayes@Database-Brothers.com

Lather. Rinse. Repeat. This is perhaps the simplest methodology I've ever read, and you can find these instructions on the back of most shampoo bottles. The programmer in me thinks, at first glance, that this looks like an infinite loop. Where's the exit? I assure you that I found an exit from this methodology after the hot water ran out and my shower turned cold!

In this paper, my goal is to have you take away an equally simple methodology for improving the performance of your IBM DB2 UDB V8.x for Linux, UNIX, and Windows (distributed) databases. *Measure, Improve, and Repeat.* It's a catchy little jingle and actual process that can yield dramatically improved business results for your organization. And I will be sure to provide you with criteria for exiting the apparent infinite loop.

Business application performance begins with data and its databases. Data is at the core of every business function. If data retrieval is slow, or data entry is slow, this slows down applications, slower applications cause less productive workers, potentially dissatisfied customers, and ultimately compromise a company's ability to thrive and grow. Perhaps it is the importance of data in every organization that places the problem monkey on the DBA's back whenever there are application performance issues. And the only way to move the monkey to the appropriate person's back (the network folks, the application team) is to definitively prove, via quantitative measurements, that the database is in excellent health and free of any performance deficiencies.

At a 20,000 foot view, the process we need to follow for tuning DB2 UDB includes taking performance cost measurements to establish a benchmark and find opportunities for improvement (some people call these problems), followed by implementing physical design and configuration changes to address the identified opportunities, then lastly re-measuring to validate and verify that actual performance improvements were achieved. *Measure, Improve, and Repeat.* Let's look more closely at the process.

Step 1: Measure

It is pretty easy to become overwhelmed by the configuration settings and all of the performance data provided by DB2 UDB. Consider a typical robust vendor application database with 4 bufferpools, 70 tablespaces, and 3,000 tables of which 2,400 are active. The application is executing 10 transactions per second and you've been asked (okay, your manager is lurking behind your chair impatiently pacing back and forth) to increase the transaction rate to 30 transactions per second without a hardware upgrade. Oh, and yes, you've got a generous 60 minutes to complete this task or your job will be outsourced. No pressure, no problem. You have one hour to triple performance. Let's have a look at the magnitude of the data you are up against, then we'll focus on **measuring what matters most.**

Step 1a: Enable the Monitor Switches

To retrieve performance data from DB2 UDB, you'll need to first enable its monitors by using the “**Update Monitor Switches Using Bufferpool On Table On Sort On Uow On Lock On Statement On**” command at a command prompt. Alternatively, these switches can be turned on by default at the Database Manager (DBM) configuration.

Step 1b: Retrieve performance data from DB2

To retrieve Database Manager performance data, issue the following command at a command prompt:

```
$ db2 "get snapshot for database manager"
```

To retrieve Database performance data, issue the following command at a command prompt:

```
$ db2 "get snapshot for database on DBNAME"
```

To retrieve Bufferpool performance data, use this command:

```
$ db2 "get snapshot for bufferpools on DBNAME"
```

To obtain Tablespace performance data, use the command:

```
$ db2 "get snapshot for tablespaces on DBNAME"
```

To obtain Table performance data, use the command:

```
$ db2 "get snapshot for tables on DBNAME"
```

To obtain dynamic SQL performance data, use the command:

```
$ db2 "get snapshot for dynamic sql on DBNAME"
```

To retrieve configuration information for the database manager, use the command:

```
$ db2 "get dbm cfg"
```

To retrieve configuration information for the database, use the command:

```
$ db2 "get db cfg for DBNAME"
```

And if you are really brave, you can obtain all of the performance data with a single command. I suggest you redirect the command's output to a file for viewing with your favorite editor. Here's an example:

```
$ db2 "get snapshot for all on DBNAME" > allsnap.txt
```

The file 'allsnap.txt' will contain 1000's of lines of data. Using the typical database described earlier, the configurations will give you about 150 lines of output, the database manager snapshot provides about 50 lines of output, the database snapshot gives about 110 lines, each bufferpool snapshot delivers about 30 data elements (4 bufferpools X 30

= 120 total lines), each tablespace snapshot yields roughly 30 data elements (70 tablespaces X 30 = 2,100 lines of raw data), each table snapshot provides 7 lines of output if the status of the table is normal (2,400 active tables X 7 = 16,800 lines of data), and the dynamic SQL snapshot provides about 14 lines per statement. Since the most recently executed statements are returned by the snapshot command depending on the size of the cache and the size of the statements, we will likely receive another 2,800 to 28,000 lines of output here. Add it all up, and the 'allsnap.txt' file may likely contain some 50,000 pieces of raw performance data. Are you intimidated or overwhelmed yet? Somehow, this reminds me of drinking from a fire hose.

Step 1c: Measure What Matters Most

For the first iteration of “*Measure, Improve, Repeat*”, it is critically important to identify **tables with high read I/O rates** and **SQL having expensive execution costs**. After you are confident that table I/O is in good health and that there are no SQL performance problems, then it becomes appropriate to broaden the scope of performance measurements to include memory related metrics such as bufferpool, sort, and cache performance. A common mistake is to begin by throwing more memory at DB2’s caches, but this temporary tuning solution is like only giving morphine to a cancer patient while not attempting to cure the cancer. I hate to use such a vivid and graphic analogy, but core physical design problems creating costly SQL executions will prohibit most applications from failing to scale no matter how much memory is thrown at DB2 or how many CPUs are added. For dramatic emphasis effect [insert picture of a man banging his fist on a desk], there are just two things to monitor and measure that are **really important**:

1. The average number of read I/O’s for each table, per transaction.
2. Equalized SQL Workload Costs and the identification of SQL having:
 - a. The highest percentage of CPU time used
 - b. The highest percentage of Sort time used
 - c. The least efficient index utilization
 - d. The highest average elapsed times

Step 1c(1) - Table I/O Costs

For each table available in the TABLES snapshot, compute the average number of rows read per database transaction. This is done by simply taking a DATABASE snapshot immediately followed by a TABLES snapshot. Add “Commit statements attempted” and “Rollback statements attempted” together giving “Database Transactions”. Next, for each table returned by the TABLES snapshot, divide “Rows Read” by “Database Transactions”. Sort the results in descending sequence for each table by “Rows Read” so that the tables with the highest Rows Read counts appear at the top of your analysis.

The astute reader will probably voice concern at this point because not every transaction accesses every table. Indeed, you are quite right, and that is the beauty of this

performance metric. Because not every transaction accesses every table, we expect “Rows Read” per “Transaction” (RR/TX) to be a rather small, single digit, integer value. Values of zero, one, and two are very common for most tables, especially if they are in good I/O health.

RR/TX is particularly important for an OLTP database. Use the Chart 1 as a healthy I/O level guideline.

Chart 1 - Health Guidelines for Rows Read / TX (RR/TX)

Rows Read / TX	OLTP	Data Warehouse
< 10	Excellent	Excellent
> 9 and < 50	Okay, but possible room for improvement.	Very Good
> 49 and < 100	Poor, and likely room for improvements.	Good
> 99 and < 1,000	Very Poor, improvements are urgent!	Okay, but possible room for improvement.
> 999 and < 10,000	CRISIS! Application performance failures, if not already self-evident, are lurking. Improve or fail.	Okay, but likely room for improvements.
> 9,999	Are you still in business? Management may be throwing money at CPU upgrades in futile attempts to sustain performance rates.	Fair. There is likely room for physical design improvements – indexes, MQTs, ASTs, and/or MDC tables.

Once you have completed the “Rows Read per Transaction” analysis and ranked the tables from the highest Read activity to the least, the next step involves identifying the SQL statements that are driving the Rows Read to the top five tables, top ten tables, or any top ranked table where there is likely room for improvement (see Chart 1).

Obtaining SQL Performance Data

You can obtain SQL performance data from Snapshots, Event Monitors, or both. Since Snapshots will only show you dynamic SQL and only the statements that were recently executed (or are currently in progress), I strongly prefer capturing SQL performance data from a Statement Event Monitor. Think of Statement Event Monitors as an accounting trace. Every SQL statement, including both dynamic and static, will be captured *upon statement completion*. And since accurately understanding the costs of the entire workload are important, you don’t want to miss a single statement.



DB2 UDB Event Monitors can have their output written to files, pipes, or, beginning with DB2 V8, tables.

Pipes are simply a memory address that a program must continuously read. Pipes have very low overhead, but unless you are a very clever programmer, writing a program to rapidly process the pipe contents is extremely difficult. The IBM DB2 supplied program “db2evmon” is capable of processing a pipe Event Monitor.

Files can be defined as BLOCKED or NONBLOCKED. Blocked files ensure that no SQL performance data is lost, but have the risk of slowing down overall database performance if DB2’s buffers become full. Non-blocked files will avoid the risk of slowing down the database if DB2’s buffers become full, but introduce the risk of potentially losing SQL performance data. Once the raw data is captured to a file, you can use the IBM DB2 supplied program “db2evmon” to process and format the file contents. Make sure your printer paper trays are full.

Statement event monitors written to TABLES will cause four insert statements (within the monitored database) for each SQL statement executed in the database. Since DB2 guarantees data integrity, these inserts are logged. The additional overhead of four inserts to every one SQL, plus logging costs, to an already strained database (that’s why you’re reading this, right?) are enormous. Think about it. There will be a 400% explosion of SQL work when DB2 has to perform a total of five statements instead of just one. In one test that ran 100,000 SQL statements through DB2, the elapsed time overhead was 68%. Of course, if you can afford this performance hit, accessing the SQL performance data will be the easiest if it is written to DB2 tables.

In the absence of a vendor tool that can automate this process, I tend to favor SQL Statement Event Monitors written to Blocked Files. Chart 2 shows sample command syntax for creating and processing Statement Event Monitors in a UNIX environment. Chart 3 shows sample commands for a Windows environment.

Chart 2 – UNIX/Linux Environment SQL Event Monitor Commands

```
$ mkdir -p /home/db2inst/sqllevt  
$ db2 "connect to DBNAME"  
$ db2 "create event monitor GETSQL for statements write to file  
'/home/db2inst/sqllevt' maxfiles 1 maxfilesize 2048 blocked replace manualstart"  
$ db2 "set event monitor GETSQL state = 1" # This turns on the capture
```

The event monitor will shut off automatically when one 8MB file is filled up. To shut off SQL Event collection sooner (try to capture 10-15 minutes of SQL activity during a peak period), use the command:

```
$ db2 "set event monitor GETSQL state = 0"
```

Now format the file using the IBM supplied "db2evmon" program:

```
$ db2evmon -path /home/db2inst/sqllevt > sqlreport.txt
```

Use your favorite editor or the **more** command to browse the sqlreport.txt file.

Chart 3 – Windows Environment SQL Event Monitor Commands

```
C:> mkdir c:\sqllevt  
C:> db2 "connect to DBNAME"  
C:> db2 "create event monitor GETSQL for statements write to file 'C:\sqllevt'  
maxfiles 1 maxfilesize 2048 blocked replace manualstart"  
C:> db2 "set event monitor GETSQL state = 1" # This turns on the capture
```

The event monitor will shut off automatically when one 8MB file is filled up. To shut off SQL Event collection sooner (try to capture 10-15 minutes of SQL activity during a peak period), use the command:

```
C:> db2 "set event monitor GETSQL state = 0"
```

Now format the file using the IBM supplied "db2evmon" program:

```
C:> db2evmon -path C:\sqllevt > sqlreport.txt
```

Use your favorite editor or Microsoft Wordpad to browse the sqlreport.txt file.

When you browse through the "db2evmon" report file, look for the SQL Event records with "Operation: Close". Chart 4 shows sample output from the "db2evmon" program.

Chart 4 – Sample “db2evmon” report output

```
Type      : Dynamic
Operation: Close
Section   : 201
Creator   : NULLID
Package   : SQLC2E03
Consistency Token : AAAAAJHR
Package Version ID :
Cursor    : SQLCUR201
Cursor was blocking: TRUE
Text      : select firstnme, lastname from employee where empno >
'000100' order by lastname desc
-----
Start Time: 04-13-2004 18:30:39.959874
Stop Time:  04-13-2004 18:30:39.962569
Exec Time:  0.002695 seconds
Number of Agents created: 1
User CPU: 0.000010 seconds
System CPU: 0.000010 seconds
Fetch Count: 24
Sorts: 1
Total sort time: 1
Sort overflows: 0
Rows read: 32
Rows written: 0
Internal rows deleted: 0
Internal rows updated: 0
Internal rows inserted: 0
Bufferpool data logical reads: 0
Bufferpool data physical reads: 0
Bufferpool temporary data logical reads: 0
Bufferpool temporary data physical reads: 0
Bufferpool index logical reads: 0
Bufferpool index physical reads: 0
Bufferpool temporary index logical reads: 0
Bufferpool temporary index physical reads: 0
SQLCA:
  sqlcode: 0
  sqlstate: 00000
```

Notice the detailed granularity of data provided in Chart 4. For each SQL statement executed, we can find the total CPU time used (User + System), when the statement started, stopped, its elapsed time (Exec Time), the number of sorts, sort overflows, rows read, rows fetched, sort time (ms), and detailed bufferpool statistics.

Now that we've discussed and demonstrated the raw SQL performance data that is available in DB2 UDB, we need to next turn our attention to SQL Equalization and Cost Aggregation in order to truly understand the costs of an SQL workload.

SQL Equalization and Workload Cost Aggregation

In July 2000, I coined the term “SQL Equalization” to describe a process for recognizing patterns in SQL statements independent of literal values, and grouping and aggregating the execution costs of similar looking SQL statements together. By doing so, an accurate execution frequency count can be determined within any database SQL workload. Plus, with the execution resource costs (CPU time, Sort time, Rows Read, etc) accurately totaled for each unique looking statement, we can then compare the total resources used for a statement against the total of all resources used by the database to determine percentages of resources used. By dividing the total resource costs for each statement by its frequency count, we can learn the average resources used per execution. You can learn more about SQL Equalization and Cost Aggregation by reviewing US Patent #6,772,411 (see <http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PALL&p=1&u=/netahtml/srchnum.htm&r=1&f=G&l=50&s1=6772411.WKU.&OS=PN/6772411&RS=PN/6772411>).

The SQL cost percentages and averages are invaluable information because they will help you accurately pinpoint, with laser focus, SQL statements in your workload that require tuning.

SQL Equalization is best explained by way of an example. Chart 5 provides a table of SQL performance data that was harvested from “db2evmon” report output. This fictitious application is a high volume OLTP application that repeats many transactions each day.

Chart 5 – Sample raw SQL workload

Total CPU is the sum of User CPU plus System CPU for each individual statement.

SQL Statement Text	Execution Count	Total CPU
Select * from TB1 where Eno = 220	1	.1
Select * from TB1 where Eno = 230	1	.1
Select * from TB1 where Eno = 260	1	.1
Select * from VW1 where Pno > 100	1	.2
Select * from TB1 where Eno = 700	1	.1
Select * from TB1 where Eno = 750	1	.1
Select * from TB1 where Eno = 760	1	.1
Select * from VW1 where Pno > 400	1	.2
Database Total	8	1.0

When the raw data from “db2evmon” reports is examined (as found in Chart 5), our eyes are instinctively drawn to the SQL statement which includes “Pno > [some value]” because its CPU cost, at .2 seconds, is twice the cost of all of the other statements in the workload. However, the SQL Equalization process provides some very interesting

insights as to where the true application workload costs lie. Chart 6 illustrates the invaluable information provided by SQL Equalization.

Chart 6 – SQL Equalization Illustrated

SQL Statement Text	Exec Count	Total CPU	% CPU	Avg CPU
Select * from TB1 where Eno = [some value]	6	.6	60%	.1
Select * from VW1 where Pno > [some value]	2	.4	40%	.2
Database Total	8	1.0	100%	.125

After pattern matching the SQL statements, independent of literal values which occur in the majority of today’s applications, we can find the total, relative percentage, and average costs for each statement. As it turns out, because of its frequency of execution, the SQL statement containing “Eno = [some value]” is actually the most expensive statement in the entire application workload. Because of its higher cost, it would be best to choose that statement for tuning.

This, of course, is a grossly simplified application used to illustrate the invaluable insight that can be gained by the SQL Equalization process. Another tangible benefit of SQL Equalization is workload compression. Perhaps you have read a Reader’s Digest magazine. In each issue, they take long novels or books and summarize these into shorter stories. This is a brilliant solution for people who are short on resources, like time. In a similar manner, SQL Equalization with Cost Aggregation will minimize resource costs such as CPU, memory, and disk space used to retain performance history.

A robust, high volume, OLTP application will execute the same transactions day after day. These transactions will run millions of SQL statements every day. However, after applying SQL Equalization techniques, you may likely find that your application actually only runs 977 different SQL statements each day. With costs grouped and aggregated, most database administrators will normally find that the top five most costly statements typically consume 75% or more of the total CPU time. Now which statements will you tune?

Another advantage of SQL Equalization is that the compressed workload can be used as very accurate input to the IBM DB2 supplied Design Advisor (the “db2advis” program). Once you have completed a SQL Equalization analysis of your application’s workload, consider creating a file that can be used as input to program “db2advis”. Chart 7 shows a sample input file to “db2advis” based on the SQL Equalization analysis found in Chart 6.

Chart 7 – Example flat file input into the Design Advisor (“db2adviz”)

```
--#SET FREQUENCY 6
Select * from TB1 where Eno = 1;
--#SET FREQUENCY 2
Select * from VW1 where Pno > 1;

-- The value of 1 is used to represent “some numeric value”. If you have
-- character data, I suggest using the value of ‘?’ to represent “some character value”.
-- Dates, Times, and Timestamps also frequently appear in SQL. You can use
-- ‘1001-01-01’ or another valid date to represent “some date value” for purposes
-- of Explain and the Design Advisor.
```

Step 1c(2) – Find High Cost SQL for Tables with High I/O

If the table PROD.ACCOUNTING had the highest number of Rows Read and 2,000 Rows Read per Transaction, the next step is to measure the SQL costs of SQL accessing the PROD.ACCOUNTING table. This SQL analysis should include any SQL that references a table via a view name or an alias name. Since views can be created upon views, and views on top of those views, we need a recursive query to find all of the view names that could drive I/O to the PROD.ACCOUNTING table. A sample recursive query is shown in Chart 8. A similar query should be run to recursively find all ALIAS names since aliases can be created upon aliases which are created upon table or view names.

Chart 8 – Recursive Query to find all Views dependent upon a table

```
WITH RVIEW (VIEWNAME, BNAME) AS
(
  SELECT ROOT.VIEWNAME, ROOT.BNAME
  FROM SYSCAT.VIEWDEP ROOT
  WHERE ROOT.BNAME = 'ACCOUNTING'
  AND ROOT.BNAME != ROOT.VIEWNAME
  AND ROOT.DTYPE = 'V'
UNION ALL
  SELECT CHILD.VIEWNAME, CHILD.BNAME
  FROM RVIEW PARENT, SYSCAT.VIEWDEP CHILD
  WHERE PARENT.VIEWNAME = CHILD.BNAME
) SELECT DISTINCT VIEWNAME FROM RVIEW;
```

For the sake of discussion, assume that the PROD.ACCOUNTING table has an alias of “TB1” and two views, “VW1” and “VW2”, created upon it. The SQL text search list, when reading the db2evmon report, must now include all, any, and only the SQL

containing FROM clause references to TB1, VW1, VW2, and/or PROD.ACCOUNTING. SQL referencing these names, and only these names, should be included in your SQL Equalization and Cost Aggregation Analysis. By using only these object names, you will substantially reduce the size of the haystack (workload) and make the needle or needles (most costly SQL) easier to find.

For review, Chart 9 shows a summary of the steps to follow to effectively measure the performance of a database.

Chart 9 – MEASURE – Steps Summary

1. Sort all active tables in descending sequence by Rows Read, and compute Rows Read per Transaction for each table. For each table having an opportunity for improvement (See Chart 1), do the following:
 - a. Determine related View and Alias names for the Table
 - b. Capture SQL event data from DB2 to a NONBLOCKED file, and then format the raw file contents with program “db2evmon”. Try to capture about 15-30 minutes of data during a peak period.
 - c. Scan the “db2evmon” report for SQL text containing the table name, or a related alias or view name, and, for each qualifying SQL statement, include this statement in your SQL Equalization and Cost Aggregation Analysis.
 - d. Work to IMPROVE to efficiency of SQL statements having the highest percentage of CPU, sort, and elapsed time costs, the worst index efficiency ratings, and the highest average elapsed times (See next Section, IMPROVE)

Step 2: Improve

By this point, you should have identified one or more SQL statements that have excessively high CPU costs, sort costs, I/O costs, and/or elapsed times. These statements were identified by applying the SQL Equalization and Cost Aggregation technique to all SQL statements accessing a Table with a high average Rows Read per Transaction value. Now it’s time to improve the efficiency and performance of these SQL statements.

Step 2(1): Explain

DB2 provides a number of tools for obtaining Explain information for any given statement. Visual Explain can be invoked from the Control Centre and provides a very good graphical representation of the access paths and costs of execution for each step. For command line explains, program dynexpln provides a simple, quick & dirty, character based explain for most statements, and db2exfmt provides a very informative and detailed description for a statement's access strategy. The usage of these tools is assumed and beyond the scope of this paper.

When reviewing Explain information from any tool, look for expensive operations such as table scans (TBSCAN), sorts, hash joins, and multiple index access. Where these costly operations exist, attempt to reduce the costs of these operations by providing indexes, clustering indexes, or composite indexes.

Hash Joins

Hash joins will be performed by DB2 when an equal predicate is present between two columns having identical data types, and one, or both, of the columns is not indexed. As such, the presence of a hash join strongly suggests the need for an index on the un-indexed column or columns.

Table Scans (TBSCAN)

Table scans are incredibly expensive to execute, especially in a high volume OLTP database. The absence of an index to narrow the data search forces DB2 to examine every row in the table to determine qualifying rows. Even scans against small tables can be very expensive if the SQL driving the I/O is executed frequently. At one site, SQL accessing a table with only 32 rows consumed 34% of all CPU time on an SMP 4-way machine. After adding an index on this 32 row table, CPU consumption of this statement became negligible. Let SQL Equalization with Cost Aggregation be your guide, and do not be fooled by the myth that small tables do not require indexes.

Sort is a Four Letter Word

Sorts, and especially sorts that overflow the SORTHEAP into TEMPSPACE, are very expensive to execute and detrimental to performance. Even small sorts, if frequently executed via transactions, can consume enormous CPU resources and substantially degrade an application's performance and ability to scale. A 7 row sort against a 350 row table used 68% of all CPU time for an SQL statement that was executed frequently. Use clustering indexes or Multi-Dimensional Clustering (MDC) tables to avoid or reduce sort costs.

Multiple Index Access

Multiple index access will be used by DB2 when a single index cannot be used to efficiently narrow the data search. With multiple index access, one or more indexes are used to partially qualify the data search, then the Row IDs (RIDs) are sorted and AND-ed. Sorts are expensive, and using multiple indexes will likely result in higher logical and physical I/O's to arrive at the qualifying RIDs. A single composite index which includes the columns used from the multiple indexes always provides a more efficient access path.

Step 2(2): Engage the IBM DB2 Design Advisor

After you have explained the most costly SQL, ask the IBM DB2 supplied Design Advisor for physical design assistance. A graphical user interface is available within the Control Centre, but most DBAs prefer to use the command line interface to this tool.

To use the command line interface, create a flat file containing the most costly statements and their respective frequencies of execution as shown in Chart 7. Use this file as input to the “db2advis” program with the following example syntax:

```
$ db2advis -d DBNAME -i FLATFILE.SQL -t 5 -m MICP > suggest.txt
```

As shown, the Design Advisor tool will generate recommendations for the SQL contained in file FLATFILE.SQL (-i) within database DBNAME (-d). It will be allowed up to five minutes (-t) to generate its recommendations. It will generate recommendations for materialized query (MQT) tables (M), Indexes (I), Multi-Dimensional Clustering (MDC) tables (C), and multi-partition partitioning keys (P). The default, if -m is not specified, is to generate indexes recommendations only (I).

Indexes

In making its index recommendations, the Design Advisor highly favors composite indexes. In fact, sometimes it can be too aggressive about including columns in an index in pursuit of Index Only Access. When creating indexes, make certain that the created indexes:

- Will not have a cardinality of 1 if the table cardinality is greater than 100
- Will have relatively high cardinality. The index cardinality should be at least 75%, or higher, of the table cardinality
- Will not have skewed distributions of index values. Skew exists when a few of the index values occur with much greater frequency than most other values in the table.
- Will not be redundant with other indexes that already exist

Multi-Dimensional Clustering (MDC) Tables

Oddly, the Design Advisor will only consider suggesting Clustering Indexes when the (C) option for Multi-Dimensional Clustering tables is specified, and even then a Clustering Index will only be proposed when the Design Advisor does not believe that an MDC table would be beneficial. Since MDC tables require that the clustering dimensions have very low cardinality, be rather wary of MDC suggestions that may include dimensions with potentially modest to high cardinality. For that matter, especially in an OLTP environment, the best bet would be to use SQL Equalization and Cost Aggregation to find the SQL with the highest sort costs against a table, and then create a Clustering index that will mitigate that sort cost.

Materialized Query Tables (MQTs)

In making Materialized Query Table (MQT) recommendations, the Design Advisor will be looking for SQL statements executed with high frequency that have a high derivation costs. An MQT may be recommended to abate the high cost of completing a join between tables or the grouping and aggregation of data (formerly called Automatic Summary Tables, or ASTs).

Partitioning Keys

The partitioning key advice for multi-partition databases may be extremely valuable if the Design Advisor is successful at co-locating data in different tables across partitions. Co-location of data avoids the broadcast of data to remote partitions. Unfortunately, a very accurate and representative SQL workload must be provided to the Design Advisor for it to learn the lowest cost partitioning key scheme.

Design Advisor Summary

By and large, the old adage of “Garbage In, Garbage Out” applies to the Design Advisor. Its recommendations will only be as good as the SQL query workload mix, with execution frequencies, that are passed into it.

MDC, MQT, and Partitioning Key performance solutions will be most applicable to Data Warehouse databases. Index recommendations are valuable for both OLTP and Data Warehouse. The Design Advisor can be used very effectively for single SQL statements that have individually been identified as having high aggregate costs across the entire SQL workload, or the SQL workload that is driving high I/O rates to a particular table.

The physical design changes that are ultimately implemented should be carefully chosen such that they address the most costly SQL (high CPU %, high Sort Time %, high average elapsed times, and/or high Rows Read / Rows Fetched) and the Design Advisor can provide excellent draft solutions.

Step 2(3): Additional Physical Design Considerations and Tasks

It is quite likely that a physical design change can be found to cure a performance problem at least 95% of the time. In fact, in more than seven years of performance troubleshooting for companies in crisis, I have never had to re-write SQL or manually adjust optimizer statistics to trick the derived access plan towards a better solution.

Dismissing the Myths

I hear from industry peers that, on very rare occasions, re-writing the SQL statement can be helpful, but this solution only works if you have the liberty to modify the SQL in a home grown application or vendor application extension. Since the DB2 optimizer re-writes all SQL during the optimization process, isn't re-writing re-written SQL redundant? While the need for re-writing SQL for Oracle is a commonly accepted and expected practice, it is very rarely necessary or useful for DB2 UDB.

DB2 provides the ability to manually update the catalog statistics which influence its optimizer. Just because you can do something doesn't mean that you should do it. During the last 7 years, I can only think of one company in Massachusetts that regularly modified the catalog statistics in pursuit of an SQL performance solution. Of course, their hands were tied because their vendor told them they couldn't change the SQL and they couldn't change the index design. I guess when you're thirsty that even pond water looks good. Avoid tinkering with the catalog statistics if at all possible, and, better yet, make sure that the catalog statistics are kept current.

Best Practices Checklist

- Identify tables with high Rows Read per Transaction
- Use SQL Equalization and Cost Aggregation to find the highest cost SQL that is driving the I/O to identified tables
- Use DB2 Explain to examine the access strategy
- Use the DB2 Design Advisor to obtain draft physical design changes for the identified costly SQL. Implement Index changes.
- Ensure that there are no indexes with a cardinality of 1 on tables with greater than 1,000 rows.
- Ensure that indexes on tables have a relatively high cardinality (75-85% of table cardinality) and do not suffer from value skew (for example, an index on US_STATE_ABBV on the US_POPULATION table would suffer from skew because a third of U.S. residents live in California while only a small fraction live in Rhode Island). Indexes with low cardinality and skew are CPU costly to maintain on SQL Insert, Update, and Delete statements.
- Drop Redundant Indexes. The first index listed below is redundant with the second:
 - Create index IX1 on TB1 (c1, c3); [drop IX1]
 - Create index IX2 on TB1 (c1, c3, c4);
- If tables in a data warehouse are read-only, alter the LOCKSIZE of the table to TABLE to save the CPU expense of maintaining Cursor Stability or other row locks
- If a table has very high insert activity, consider altering the table to enable APPEND ON mode. Clustering indexes are prohibited when APPEND is ON.

Step 3: Repeat

Database tuning is an iterative process. After making any change, return to Step 1 and re-measure performance. Look for reductions in Rows Read per Transaction for the table that had physical design changes in Step 2. Also review the Equalized SQL workload for the table and verify that one or more of the following improvements occurred:

- ◇ The percentage of CPU consumption was reduced.
- ◇ The percentage of sort time consumption was reduced.
- ◇ The average rows read per statement execution was reduced.
- ◇ The average statement elapsed time was reduced.
- ◇ Index Efficiency was improved. IXEFF = Rows Read / Rows Fetched. Smaller numbers are better.

Invariably, you will find that you were very successful in reducing table I/O and improving the performance of the SQL workload. As you review the ranking of tables with highest Read I/O to least, you will likely find the tables you made physical design changes to have dropped substantially in the list. The good news is that there will

probably be a new table name at the top of the list of tables having the highest Read I/O, so you have the opportunity to repeat Step 2 and improve the performance of SQL that is accessing this new, top ranked, highest Read I/O table.

Ah yes, it would seem that the DBA's job is never done. *Measure. Improve. Repeat.* When does it end? Begin by applying the rules of thumb in Chart 1 for your type of database, and then consider the following additional criteria:

- Phone rage ends (your phone stops ringing off the hook with complaints)
- After SQL Equalizing the entire SQL workload and aggregating statement costs:
 - No single statement in the workload uses more than 10% of total CPU time
 - No single statement in the workload has an average elapsed time that exceeds 50% of any Service Level Agreement transaction response time requirement
 - No single statement in the workload has an average Rows Read per Execution exceeding 100 rows (OLTP only)

When the above guidelines are achieved, you can confidently exit the *Measure, Improve, and Repeat* loop. That is, you're done for now. Next week, after the application data grows or more users are added to the system, you should conduct periodic measurement reviews to ensure that the database is still performing within the best practices guidelines.

Additional Measurements and Improvements

The astute reader will notice that so far there hasn't been hardly any mention of tuning the 100+ database and database manager configuration parameters. That's because 75-80% of performance improvements typically come from the physical design changes discussed in the methodology above. Configuration tuning should be viewed as the icing on the cake. With this in mind, consider the following tuning guidelines for various, relatively important, configuration parameters.

MAXFILOP

The default value for database configuration parameter MAXFILOP is far too small for most databases, and, if this value is too small, then DB2 spends a lot of extra CPU processing time closing and opening files to be a good citizen amongst other processes running in the operating system. This slows down both OLTP response times and Data Warehouse query results.

DBA's should issue the command "db2 get snapshot for database on DBNAME" and look at the value for "Number of files closed". If this value is greater than zero, then keep incrementally increasing MAXFILOP until DB2 stops closing files.

MAXAGENTS

Depending on the number of users accessing the database, there has to be enough DB2 Agents available to process the workload. If there are too few Agents available, then DB2 starts stealing Agents from one user's connection to process the work of another DB2 user. This stealing is like corporate employee raiding. It's expensive to lose valuable employees and expensive to train new employees... there's a lot of overhead that just isn't relevant to getting the work at hand done.

DBA's should issue the command "db2 get snapshot for database manager" and look at the value for "Agents stolen from another application". If this value is greater than zero, then more Agents need to be provided to the database manager by incrementally increasing the value of MAXAGENTS until the thefts stop.

CATALOGCACHE_SZ

Having sufficient memory allocated to the CATALOGCACHE_SZ database configuration parameter is beneficial to OLTP and DW databases as well. When preparing execution strategies for SQL statements, DB2 checks this cache to learn about the definition of the database, tablespaces, tables, indexes, and views. If all the required information to develop the plan is available in the cache, disk I/Os can be avoided which shortens plan preparation times.

Having a high package cache hit ratio of 95% or better is particularly key for today's OLTP database applications. Keep increasing the CATALOGCACHE_SZ until this goal is reached. Issue the command "db2 get snapshot for database on DBNAME" and compute the hit ratio using the following formula:

$$100 - ((\text{Catalog cache inserts} \times 100) / \text{Catalog cache lookups})$$

The CATALOGCACHE_SZ also needs to be increased if the value of "Catalog cache overflows" is greater than zero. And, if the value of "Catalog cache heap full" is greater than zero, then both DBHEAP and CATALOGCACHE_SZ should be proportionally increased.

LOCKTIMEOUT

One more parameter that has universal applicability to both OLTP and DW databases is LOCKTIMEOUT. The default value is -1, which means that user connections can wait infinitely to get the locks they need, and this can be a real problem if someone leaves their desk for the night without releasing locks that are held by their DB2 work. In a Data Warehouse, a value of 60 seconds is a good guideline.

And in an OLTP database, set LOCKTIMEOUT to 10 seconds. If your transaction can't get the locks it needs to complete its work, and it may be holding locks of its own while it waits, then timeout quickly and get out of the way of other transactions before the entire house of cards falls down. By house of cards, I mean that there are so many transactions waiting on locks they can't obtain, that the whole application locks up, or performs horribly slowly.

TEMPSPACE TABLESPACES

Remember to define TEMPSPACE tablespaces so that they have at least three or four containers across different disks, and set the PREFETCHSIZE to a multiple of EXTENTSIZE, where the multiplier is equal to the number of containers. By so doing, parallel I/O will be enabled for larger sorts, joins, and other database functions requiring substantial TEMPSPACE space.

And even though an OLTP application may be finely tuned to avoid large sorts, the occasional decision support queries do find their way into OLTP databases so be sure to tune the TEMPSPACE according to the guideline above for both OLTP and Data Warehouse databases.

MINCOMMIT

MINCOMMIT is a powerful tuning tool for OLTP databases that process high volumes of transactions per second. When properly set, I/Os to DB2's logs are grouped together resulting in fewer log I/Os. To set MINCOMMIT, determine how many transactions per second the database is performing, and divide this value by ten:

$$\text{Commits} + \text{Rollbacks (Transactions)} / 10 = \text{MINCOMMIT}$$

By way of example, if MINCOMMIT is set to 5, then 5 transactions have to be ready to commit before the commit is performed. DB2 will wait up to one second for the 5 transactions to be ready.

In a Data Warehouse database, set MINCOMMIT to 1. Also, beware that MINCOMMIT > 1 can be painful if there are not sufficient numbers of concurrent transactions per second to warrant it. One client accidentally added six hours elapsed time to a Siebel batch process that did frequent commits by merely setting MINCOMMIT > 1.

LOGBUFSZ

While on the subject of saving log I/O's, the default LOGBUFSZ value of 8 is drastically too small. This buffer should be increased to at least 256 for most OLTP databases.

Data Warehouse databases can benefit from some extra LOGBUFSZ memory as well. 128 is a good guideline.

INTRA_PARALLEL

For an OLTP database, set the database manager configuration value for INTRA_PARALLEL to NO. Using CPU parallelism to have Coordinator Agents manage Sub-Agents in pursuit of retrieving a small answer set for well tuned and indexed SQL would be a waste of CPU resources, and can actually slow down transaction throughput rates. Using a value of YES would be akin to having too many cooks in the kitchen.

For a Data Warehouse database, you may want to use an INTRA_PARALLEL value of YES to enable CPU parallelism, and be sure to set the database configuration parameter DFT_DEGREE to the value of ANY or -1. As such, the degree of parallelism will be computed by DB2 based on the number of system CPUs and other available resources. However, do not use INTRA_PARALLEL YES if there will be several concurrently executing queries. Only use YES when the typical number of concurrently executing queries is small.

MAX_QUERYDEGREE

Since parallelism is generally undesirable in well-tuned OLTP databases, set MAX_QUERYDEGREE to the value of 1.

For a Data Warehouse database, a proactive, cautious DBA will take care to set MAX_QUERYDEGREE equal to the number of CPUs on the system. This will prevent rogue users from “accidentally” setting their CURRENT DEGREE too high. Performance tests performed have shown disastrous results when a user tries to inflate his or her CURRENT DEGREE in excess of the number of CPUs.

SHEAPTHRES & SORTHEAP

The database manager configuration parameter SHEAPTHRES works in concert with the database configuration parameter SORTHEAP to govern sort memory. SHEAPTHRES dictates a soft limit for total sort memory used by an entire instance, and SORTHEAP controls the limit on memory for any one sort. The respective DB2 UNIX default values of 20,000 and 256 provide for up to (20000/256) 78 concurrent 1 MB sorts. OLTP databases should perform as few sorts as possible, and the sorts should be very small. I have seen a simple two-column sort of two-five rows consume 33% of the CPU time on a machine with four CPUs. Believe me when I tell you sorts are costly to application OLTP databases, for a small sort almost put one company out of business by sorting ten

lousy rows each transaction. The key to avoiding sorts is having the right CLUSTERING indexes defined. A well-tuned OLTP database can use a SORTHEAP value as small as 128, which doubles the number of concurrent sorts without increasing the SHEAPTHRES memory. As a general guideline, the default value for SORTHEAP of 256 is usually adequate for well tuned OLTP databases.

On the other hand, Data Warehouse databases perform a lot of sorts, and many of them can be very large. SORTHEAP memory is also used for hash joins which are enabled by default in DB2 V8+. For a Data Warehouse database, at a minimum, double or triple the SHEAPTHRES (40960-61,440) and set the SORTHEAP size between 4096 and 8192. If real memory is available, some clients use even larger values for these configuration parameters.

FCM_NUM_BUFFERS

This parameter specifies the number of 4k buffers created for the FCM Buffer Pool. With intra-partition parallelism enabled or when using EEE, Fast Communications Manager (FCM) manages the communications between parallel agents. A shortage of FCM_NUM_BUFFERS can cause serious performance degradation, like DB2 stops responding! Error messages indicating FCM resource shortages are written to the db2diag.log file. FCM resources can be monitored via database manager snapshot monitoring. Start with the default but monitor frequently. Use the following formula to determine whether or not to increase the number of buffers: $FCM_NUM_BUFFERS - FCM_BUFF_FREE_BOTTOM$. If $FCM_BUFF_FREE_BOTTOM < 20$ percent of $FCM_NUM_BUFFERS$, increase $FCM_NUM_BUFFERS$ until $> \text{ or } =$ to 20 percent $FCM_NUM_BUFFERS$. This will ensure that an adequate number of buffers are always available.

Since I don't recommend the use of intra-partition parallelism with OLTP, FCM buffers do not come into play unless the database has multiple partitions due to its size. Again, make sure you set INTRA_PARALLEL to NO for OLTP. And, if multiple partitions are not being used, an AIX memory segment can be freed up for buffer pool or other use by setting the registry valuable `DB2_FORCE_FCM_BP=NO`.

DFT_QUERYOPT

The SQL executed by OLTP databases should be short, sweet, relatively simple, properly indexed, and well tuned (see the Measure, Improve, and Repeat methodology above). Selecting an access strategy (plan) shouldn't take much brainpower on DB2's part. Set the database configuration parameter DFT_QUERYOPT to a value of 1 so that DB2 spends minimal time preparing its access strategy for a given SQL statement. It would be tragic to spend a few seconds thinking about an SQL's access strategy when actual execution only takes one quarter of a second. The default value is 5. The smaller the

value, the less time DB2 spends pondering its plan, which means execution of the SQL can begin sooner.

To the contrary, SQL in a Data Warehouse database is very complex and often consumes large quantities of CPU and I/O resources, so a few extra seconds of time spent on preparing the SQL access strategy could be a very wise investment, especially if DB2's pondering more creative alternatives yields an access strategy that trims minutes or hours off of a query's elapsed time. In a DW database, set DFT_QUERYOPT to a higher value of 5 or 7. Be wary of using the value of 9 as a number of customer sites have reported disappointing results with this value.

Lastly, if your database is an OLTP database that is blessed with occasional decision support queries, a compromise value of 3, or the default value of 5, may be appropriate for DFT_QUERYOPT.

CHNGPGS_THRESH

The default value for CHNGPGS_THRESH is 60%, meaning that when 60% of the pages become dirty in the bufferpools, then the NUM_IO_CLEANERS begin asynchronously writing the changed pages out to disk. For a DW database, the default value of 60% usually delivers good results.

For a high transaction volume OLTP database that runs lots of DML SQL (Inserts, Updates, and Deletes), lowering the CHNGPGS_THRESH from 60% to 50% or 40% can be beneficial. If left at 60%, over half of all bufferpool pages are dirty before DB2 starts writing them out to disk, and, when that threshold is reached, some users experience a spike in transaction elapsed times while this surge-like tidal wave of write I/O occurs to disk. By lowering the CHNGPGS_THRESH, the tidal waves of write I/Os are smaller and hence transaction response times remain more consistent. I'm not aware of any DB2 customer that has this set lower than 30% though, so don't over do it.

NUM_IO_CLEANERS

While we're on the subject of write I/O agents, let's cover that database configuration parameter next. Performance experience tells us that asynchronous write I/Os are usually at least twice as fast as synchronous write I/Os, thus it is important to try to achieve an asynchronous write percentage (AWP) of 95% or higher. The formula for AWP is given in Chart 10.

The default value of 1 is rarely sufficient. I usually suggest incrementing the number of NUM_IO_CLEANERS by one until either 95% of writes are performed asynchronously or the number of NUM_IO_CLEANERS is equal to the number of CPUs on the DB2 server. If the latter limit is reached, consider making gradual reductions in CHNGPGS_THRESH until AWP > 95% is reached, but set CHNGPGS_THRESH no lower than 30%.

Chart 10: Asynchronous Write Percentage (AWP) Formula

$$\text{AWP} = \left(\frac{\text{Asynchronous pool data page writes} + \text{Asynchronous pool index page writes}}{\text{Buffer pool data writes} + \text{Buffer pool index writes}} \right) \times 100$$

This formula applies equally well to database, bufferpool, and tablespace snapshot data. Use the database AWP to guide your NUM_IO_CLEANER and CHNGPGS_THRES tuning activities.

NUM_IO_CLEANERS will be used by DB2, in a Data Warehouse database, for writing to TEMPSPACE, temporary intermediate tables, index creations, and more. Set NUM_IO_CLEANERS equal to the number of CPUs on the DB2 server.

NUM_IO_SERVERS

To achieve maximum I/O parallelism in a Data Warehouse database, it's imperative to have enough NUM_IO_SERVERS available, but not too many either. IO_SERVERS are used to prefetch data into DB2's bufferpools. To set NUM_IO_SERVERS, add up the number of physical disk devices (arms) on the DB2 server and use that value, but not more than four to six times the number of CPUs.

A well-tuned OLTP application database should not have the same need to perform extreme I/O parallelism. Set NUM_IO_SERVERS equal to the number of CPUs so that

prefetch Agents may be available for occasional ad hoc Data Warehouse type queries, but not less than the default value of three.

RAID DISK and PARALLELISM

For regular SCSI or IDE disk drives, tablespaces should have multiple containers on different disks. For RAID devices where several disk devices appear as one disk to the operating system, be sure to do the following:

1. db2set DB2_STRIPED_CONTAINERS=YES (do this BEFORE creating tablespaces or BEFORE a re-directed restore)
2. db2set DB2_PARALLEL_IO=* (or use TablespaceID numbers for tablespaces residing on the RAID devices, for example DB2_PARALLEL_IO=4,5,6,7,8,10,12,13)
3. Alter the tablespace PREFETCHSIZE for each tablespace residing on RAID devices such that the PREFETCHSIZE is a multiple of the EXTENTSIZE. Most companies use a multiple of three to five with very good success, and four is my favorite.

Proper container placement and alignment of the extent with the raid stripe size can reduce I/O times by 50%. DB2 V8.2 (Stinger) will automatically align DB2's extents with the raid stripe size, so it will no longer be necessary to set the DB2_STRIPED_CONTAINERS environment variable.

Final Thoughts

The methodology of measure, improve, and repeat will help you achieve and sustain database performance that meets and often exceeds expectations. If you can successfully deploy the methodology on an ongoing basis, not only will you ensure that Service Level Agreements are met, but it is likely that unnecessary hardware upgrades will be avoided.

While IBM would like you to think its DB2 database is self-tuning and self-healing, remember that IBM is foremost a hardware vendor with an "On Demand" strategy for enabling CPU upgrades. This reminds me of having the fox guard the hen house. You can do better, and I am confident that with the right methodology, discipline, and tools you will.

Scott Hayes is a well known DB2 UDB performance expert and the former President and CEO of Database-GUYS Inc (DGI). He is a regular speaker at International DB2 User Group conferences, IBM DB2/Data Management conferences, and is frequently sought as a guest speaker for regional DB2 user group meetings. Scott is an IBM DB2 Gold Consultant (an elite, exclusive group of IBM recognized top DB2 advocates), has obtained Advanced IBM DB2 Certifications, and is widely regarded by the worldwide DB2 community as one of the top performance experts for DB2 on distributed platforms (UNIX, Linux, and Windows).