# Out in Left Field

strategic tools for strategic teams

Volume Three, Issue Three

## Water, water, every where, Nor any drop to drink

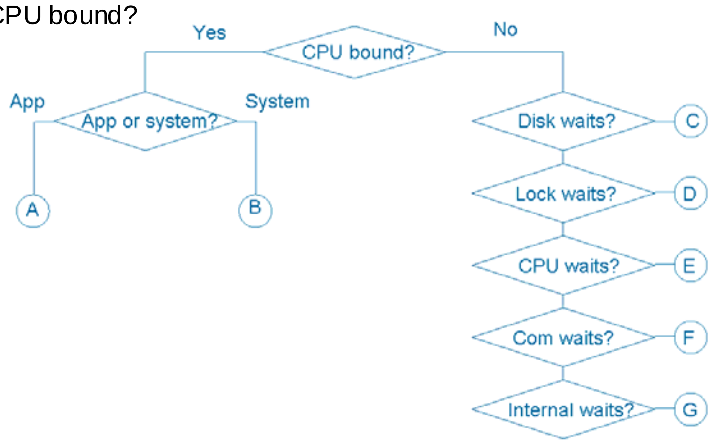
from "The Rime of the Ancient Mariner," by Samuel Taylor Coleridge

Much like the sailor surrounded by salt water he cannot drink, IT is often flooded by more data than is useful. A great example of this is the volume of information collected by performance tools. The amount of information gathered for analysis can be very intimidating and actually slow the diagnosis of problems. Furthermore, even if a software package makes it easy to generate reports, finding the important and actionable data points can be difficult.

Let's start with first principles. Only two performance metrics count in the real world; response time for interactive users and clock time for batch jobs. It is easy to get caught up in the analysis of CPU, page faults, disk I/O, or communication time and forget that the definition of a well-tuned application is based on human measurements and not performance data. Those performance numbers do matter, but only if they represent a bottleneck that increases response or clock time.

In recent years, the focus of most tuning effort has shifted away from system-level performance optimization. Most iSeries systems run very well most of the time and don't need a lot of work management tweaking. Performance challenges more often surface for a particular part of an application, for a particular user, or for a particular job. The good news is that this greatly reduces the amount of data to collect and analyze.

When any program runs, it is either using CPU or waiting. To diagnose a response or clock time problem, you need to start with this fact. The following chart shows a set of questions that can be asked and answered to drill into performance problems using this basic principle. The data collected by IBM or Centerfield performance tools can then be used to answer each question and get to the root cause of the problem so it can be corrected. The first question is very simple; Is the application CPU bound?



### A: Application code is CPU bound

If the program code is CPU bound, "hotspot" analysis must be done. This can either be done using Performance Explorer (PEX) or with instrumentation of the code. Either way, the routine or sections of code with high CPU need to be analyzed and changed. Normally the high CPU is caused by a poor algorithm or design that, once isolated, can easily be changed to be more efficient.

### B: System code is CPU bound

If you find that code written by IBM (e.g. language runtime, database requests, or SQL) is CPU bound, then you have several options. For SQL, tuning the database with appropriate indexes is typically the first choice because it is relatively low risk. In all cases, it is worth looking at the application to make sure that the use of system functions is well structured and necessary. For query requests, our HomeRun product can isolate the offending SQL statement so you have the necessary information needed to start your tuning efforts.

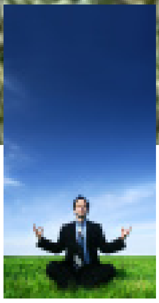
### C: Disk I/O waits

Many response and clock time problems can be tracked back to disk I/O. Because disk drives are the slowest component of today's systems this is not a big surprise. While it is not possible to eliminate disk I/O, there are many techniques to minimize the number of requests including:

- Selective use of the SETOBJACC command.
- Building optimum indexes for SQL statements and other query interfaces
- Reducing the number of unused indexes for a highly updated file
- Reorganizing a file in the same order as the most popular index or logical file
- Using larger pages for logical files

(Continued on page 4)

**Inside this issue:**



# Good SQL in Bad Programs

The support staff here at Centerfield is asked occasionally by one of our customers to help tune a long-running batch job. The request is made because they have already used the insure/INDEX and insure/ANALYSIS tools to tune the SQL but the job is still taking longer than it should. The right indexes or logical files are typically available and are being used, but the analysis shows that most of the time is still being spent in SQL statements rather than the program's logic. Given the cost of the SQL, how can the job's time be reduced to fit into an acceptable processing window?

The most likely answer is that the program logic driving the SQL needs to change. Essentially, the programs used by the batch job have efficient SQL being used in an inefficient way. The purpose of this article is to go over a few examples of this phenomenon we've seen in poorly written programs.

## Good SQL in Bad Programs

Surprisingly one of the most common reasons SQL statements run exceedingly fast is that they access empty files. In one application we analyzed, approximately 26% of the application's CPU (and 8% of the whole system's!) was used by two SQL statements. These statements were reading state tax tables that did not contain any rows. Because the application was written to use default percentages for the tax rate if the individual state values could not be found, the application function correctly. To make matters worse however, the application was using pure dynamic SQL and full opens which not only increased CPU use but also contributed significantly to the response time because of added disk I/O. By simply eliminating the part of the logic that had never been implemented, both statements could be removed and the application made much faster.

What do you do if you have a file or table that is empty most of the time but does need to be processed if it contains data? There are several very efficient options that can be used. The first is to use a database trigger. The trigger program

*(Continued on page 3)*

## CTO MEMO

We are in middle of a hot summer and it seems appropriate to talk about some of our equally hot software.

For those of you using HomeRun Version 5.0, a cumulative patch is now available for download at http://www.centerfieldtechnology.com/upgrades_isql_cume500.asp

If you are using insure/SECURITY, insure/MONITOR or insure/RESOURCES, it is important to install this cumulative patch especially if you have a high number of external connections to your system (ODBC, JDBC, DRDA, DDM, Remote Command, etc.). If you have questions about how to install this patch please call our support team. This cumulative fix does require some pre-planning so be sure get the details on the patch so you can install it as soon as it works into your system upgrade schedule.

HomeRun 6.0 is now in Beta and we are getting great feedback from the accounts giving the new release a test drive. I'd like to personally thank those of you that have taken the time to install 6.0 and take the time to give us compliments, constructive criticism, and ideas to make it even better! We are pretty excited about the initial results some of you have seen and hope to get feedback from more customers as summer begins to cool off a bit.

As we approach the start of the school year, be looking for another exciting product announcement from Centerfield. We'll give you full details in our next newsletter.

Mark L. Holm

Mark L. Holm
Chief Technology Officer
Centerfield Technology, Inc.

*(Good SQL in Bad Programs — Continued from page 2)*

can either do the processing itself or set a global variable (either in a user space or data area) that tells the program that needs to process the data that there are rows to be read. A second option is to add some additional logic to the program that inserts the data into the table and have it send a message (via message queue or data queue) to a background job that processes the information.

## Do-again SQL

Another common issue is using SQL to read and re-read data that does not change. We recently found an example of this in a web application. An Apache web server was running on a Windows platform with a Java/JDBC application to access System i data. As part of every customer's transaction, an SQL statement was used to retrieve a list of the company's warehouses from where parts could be shipped. Obviously, it was very uncommon for a warehouse to be added or removed from the list. Getting the list of warehouses was a reasonably fast operation but in reality only needed to be done when the application started instead of thousands of times a day.

When an application references data that does not change it makes much more sense to put that information into an array or result set. If the data is somewhat static, but it the program must have the latest information to function correctly, techniques like triggers or data queues (as described in the previous section) can be used to inform all users that their local copies need to be refreshed. In all of these cases avoiding the repetitive use of SQL is the best option even if it runs very quickly.

## Do-little SQL

Yet another problem is when SQL is simply used as a record-at-a-time access method instead of a set-at-a-time tool. This error often occurs when an RPG programmer takes an algorithm they have used often and replaces native database I /O with SQL. The problem with this approach is that the power of SQL is not being used but the performance overhead is introduced.

A classic example of this problem is when the program does its own file joins. Instead of using SQL to join two or more tables together, the program is written so each SQL statement gets data from a single file and the program itself implements the join logic. This approach can be much less efficient because:

- The program does not optimize the join order like the query optimizer and thus may be orders of magnitude slower than necessary.
- The program may not take full advantage of indexes because the query optimizer does not see the "whole story" since the SQL is broken up into very simplistic reads.
- Indexes and logical files are not fully utilized because selection logic may be implemented in the program rather than being pushed into the database engine. In general, if DB2 can implement selection criteria it is significantly faster than what can be done in a program.

## Summary

Next time you have a job that needs to get done much faster but the SQL appears to be running very efficiently, look for the following signs of trouble:

- The top three to five statements are executed tens or hundreds of thousands of times
- Each execution of those top statements finishes very quickly.
- The cumulative execution of those top statements chews up a large percentage of the total elapsed time of the job.

If these warning signs appear, it is best to go back to the programmer and ask them to analyze the program and see if they can reduce (or eliminate!) one or more of the most common statements. It may be time for a code review and brainstorm ways to more effectively use the power of SQL and let the database engine get the job done for you.

*(Water, water everywhere, Nor any drop to drink — Continued from page 1)*

## D: Lock waits

Record and object locks can cause excessive and unexpected delays in applications. These types of problems can be frustrating to diagnose because they tend to be intermittent and thus difficult to catch. Even if you can watch them happen, it can be tricky to find the job holding the locks in order to determine why the conflicts are occurring. To avoid lock problems the following techniques are required:

- Use of small commitment control transactions. If a database transaction locks many rows under commitment control, the likelihood of a conflict increases dramatically. By keeping the number of locks to just those required for data integrity, the chance of a conflict can be reduced.
- Avoid situations where locks are held for a long period of time. Ensuring that locks are released in a timely manner will prevent lock conflicts as well. This requires the programmer to be aware of situations where user interaction is needed while locks are held and write code to minimize or avoid those situations.
- Proper error handling. Locks acquired by a program may not be released as expected if an error occurs and the exception handling is not set up to clean them up.

## E. CPU waits

Jobs waiting to use the CPU are generally competing with high priority jobs or tasks. They may also be using a much smaller timeslice than other system jobs and be “starved” for processor time. To solve this problem, work management settings (e.g. the CLASS object) can be adjusted to ensure the job gets its fair share of system resources.

## F. Communication waits

Most major applications today use resources from multiple platforms. This introduces another potential bottleneck or delay point for the application because the remote system must communicate with a server. Normally this involves a TCP/IP connection using a socket. If the application is running remotely over the internet rather than private communication lines, the response time for those users can be unpredictable. For system administrators trying to solve performance problems, this can be a source of frustration – particularly if it appears that the server is running well but the end-user is not happy with response time. While beyond the scope of this article, communication tracing can help to diagnose problems at this level. For applications based on ODBC or JDBC several other techniques can be used to prove or disprove the problem is caused by a server or communications.

- Exit points can be used to trace requests coming into the system.
- Tools like insure/ANALYSIS can be used to ensure that SQL running on the server is returning data to the client in a reasonable amount of time
- Tools like insure/MONITOR can be used to get more visibility into the server jobs used by the remote client to help a system administrator understand what that job is doing.

## E. Internal waits

Internal waits is a term used by IBM to describe a wide variety of reasons that jobs must wait on resources within the operating system. To diagnose these types of delays advanced tools like Job Watcher (from IBM) are generally required. For some of the more common wait types, the insure/ MONITOR tool can be used to identify which jobs are waiting most often. One of the most common “internal wait” types is something called seize contention. Seize contention is generally caused by high rates of database activity or because a common user profile must be updated often (e.g. when objects are created and destroyed that the profile owns). The most common techniques to eliminate database seize contention is to convert all logical files to use the *MAX1TB option and to make sure that the journal environment has been optimized with the appropriate types of disk drives (with write cache).

## Summary

It is easy to be flooded with an overwhelming amount of data from performance tools. Because of that, it is important to identify the subset of data that is most likely to provide useful and actionable information. Hopefully this article provided some simple guidance on how to reduce the ocean of data and help you with your next performance challenge.

# DELETE DUPLICATE ROWS

Sooner or later you are going to run into situation where you must delete duplicate rows out of one of your data tables. If this cleanup is an ongoing process, an HLL program performing native I/O is a very good candidate, potentially performing better than the equivalent SQL solution. CL biased users sometimes opt for solution leveraging CPYF command. If on the other hand this is a one time cleanup process or you prefer the solution that's easier to maintain and portable to other databases, then SQL is a natural choice.

What exactly defines a duplicate row will determine what approach you should take to cleaning up duplicates. Your first thought may be that a duplicate is a row that matches every field in the row. If this truly is the case, consider using SQL's DISTINCT clause to aid you in getting rid of duplicates:

```
CREATE TABLE NEWTABLE AS (SELECT DISTINCT * FROM OLDTABLE) WITH DATA
DELETE FROM OLDTABLE
INSERT INTO OLDTABLE SELECT * FROM NEWTABLE
```

You may be wondering, why doesn't he just delete the OLDTABLE and rename the NEWTABLE to it, this would run much quicker. The problem with that approach is that often there are triggers, constraints, logical files, indexes, views etc. on the original table and all those would have to be recreated and reattached.

The obvious drawback of this approach is that a separate copy of the data is created in the process and that base table's data has to be cleared and repopulated with the INSERT statement. Another potential pitfall is that while the potentially lengthy repopulation process is taking place, the application is essentially on hold.

Another, not so obvious drawback, is that duplicate row seldom means every field is a duplicate, but rather that certain fields are duplicated while other fields in the row are relatively distinct. Fields that form this **natural primary key** determine if the row is a duplicate or not.

This is the situation in every real life scenario I've been a part of so let's discuss an appropriate solution for it. SQL is a perfect fit here even though set of fields that user/programmer/administrator thinks of as a natural key are not defined as true PRIMARY KEY or UNIQUE constraint on the table itself. The user is looking at the dirty data after the fact and needs a solution to force the data to be unique before creating a primary key or unique constraint and ensuring a natural primary key is preserved automatically in the future. Without further ado, let me offer a solution that should work in most, if not all, circumstances:

```
DELETE FROM myLib/myFile d
WHERE RRN(d) IN
   (SELECT RRN(b)
    FROM myLib/myFile b,
         (SELECT a.field1, MIN(RRN(a)) minrrn
          FROM myLib/myFile a
          GROUP BY a.field1) AS c
    WHERE b.field1 = c.field1 AND RRN(b) > c.minrrn)
```

The subselect with the yellow background is aliased as the temporary table C. This table determines what fields constitute natural primary key and pulls the minimum RRN from the set of duplicates. In my example I use a single field as the key (field1). If you needed to include multiple fields you would alter table C definition, i.e.:

```
         (SELECT a.field1, a.field2, MIN(RRN(a)) minrrn
          FROM myLib/myFile a
          GROUP BY a.field1, a.field2) AS c
```

and then ensure that same keys are joined on in the outer select, i.e.:

```
    WHERE b.field1 = c.field1 and b.field2 = c.field2 AND
          RRN(b) > c.minrrn)
```

(Continued on page 6)

# Case for the Autonomic DBA Assistant

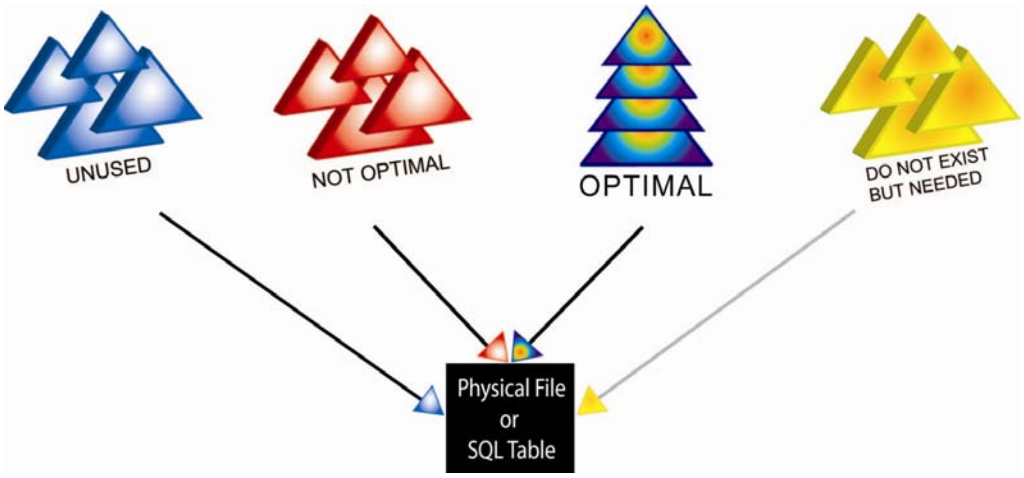
For 10 years Centerfield has provided premier database tooling to System i customers. With wide adoption of SQL and 3rd party solutions rather than in-house developed solutions, the need for proper indexing strategy arose. Centerfield filled the gap with insure/INDEX which pulls index recommendations from three different sources (query optimizer advice, temporary index builds and our proprietary "perfect index" advice engine). Stories of "magical Centerfield index" roamed through user groups, bringing batch runtimes from 6 hours down to ½ hour or 5 days to 4 hours. Everyone was happy.

Then the cost of DASD dropped and cross-platform ERP applications gained favor on the System i. Vendors saw nothing wrong with shipping 6000 SQL indexes with their application. At the same time, trusty RPG developers kept on creating keyed LFs every time they needed to perform a SETLL lookup on a file. The result is an abundance of data space index objects (keyed LF, SQL index, keyed PF, unique/primary/foreign key constraint) laying around and chances are a lot of them are not optimal or even worse, not even unused.

We, along with customers, realized a need for balanced approach to index management was necessary. It was no longer sufficient to advise needed indexes and we needed to address unused and suboptimal indexes as well.

Enter AutoDBA. We were already busy building the infrastructure for autonomic advice, autonomic execution of it, advanced filtering, rollback capability and continuous auditing. It turns out DBA autonomics fit our need for balanced index management perfectly.

To put this in perspective, let's categorize index objects into four types as outlined below:



*(Delete Duplicate Rows — Continued from page 5)*

In most cases there is no field in the user's table designated as unique constraint and the only differentiating feature I could use is the relative record number (RRN), as illustrated in my example. Unfortunately, RRNs are not keyed internally by the database. This little fact has a negative impact on performance of the example above, but it can't be helped.

To summarize my SQL approach to deleting duplicate rows:

**PROS:**

- in place deletion
- no database changes required
- easy to maintain and tweak
- easily reusable for other tables
- can be run interactively or programmatically (in batch)

**CONS:**

- potentially poor performer
- RRN is not necessarily supported on other platforms so ROWID or similar facility may need to be used instead

Hope this little tip helps you clean up that dirty data.

Elvis

# create TABLE like

A powerful but little used function of SQL is the ability to create temporary work tables from other tables or traditional DDS files. Because the full power of SQL is available to do this, very complex operations can be performed which can eliminate programming logic and centralize a data extraction process.

Here is the basic SQL syntax to create a temporary table based on an existing table or view:

```
CREATE TABLE table-name LIKE table-or-
view-name
```

## Product sales example

Let's say we need to create a table to be downloaded into a SQL Server database for further analysis, but want to ensure there is only one place that defines how the information is extracted and formatted. We might first define an SQL view to summarize the total sales for each part in our orders table. The view definition might look something like:

```
CREATE VIEW prod/partsales AS SELECT
l_partkey, SUM(l_quantity *   l_price)
as totalsales FROM prod/orderline GROUP
BY l_partkey;
```

We want to use this view as a template to store the extracted data. The SQL script create the table and populate the new table would look like this:

```
CREATE TABLE qtemp/partsales LIKE prod/
partsales;
INSERT INTO qtemp/partsales SELECT *
FROM prod/partsales;
```

The CREATE TABLE statement looks at the definition of the prod/partsales SQL view and creates an empty table that matches the view's columns and data types exactly. Because the subselect used by the INSERT statement gets all columns from the view the data will be added column-for-column without any problems.

## Second Product sales example

A second approach to getting part sales information into a temporary table is to use a single select statement. This syntax has the advantage that it also incorporates the ability to copy data to the new table as part of the creation process. The syntax of the SQL is:

```
CREATE TABLE table-name AS (subquery)
with-data-clause
```

With this flavor of SQL, the temporary table can be created in one step rather than multiple:

```
 CREATE TABLE qtemp/partsales AS
(SELECT * FROM prod/partsales) WITH
DATA
```

The use of WITH DATA tells DB2 that we want to immediately run the select statement and insert data into our temporary table.

## Suggestions

We'll finish this article by giving you several other ideas and suggestions for populating and using these temporary tables:

- Use the FETCH FIRST n ROWS clause in conjunction with ORDER BY on the select statements to extract only the highest or lowest values. This keeps the size of the table smaller and will help subsequent reporting that only needs to do exception or management reporting.

- Use RUNSQLSTM scripts to create temporary tables that can be passed into subsequent statements for processing. This idea is very similar to the idea of "piping" in DOS batch files or Unix/Linux scripts where the output of one step is used as input into the next step. This can simplify the individual SQL statements while allowing fairly complex processing to take place.

- Remember that if you do create data in a temporary table and you want to run SQL to query that data later, you need to follow the same tuning rules as you do with your other tables – they must have the proper indexes created so those queries perform well.

- If you want to extract data on a regular basis with the method described in this article, and you need to more complex calculations, we suggest that SQL views be used as the source of the information (as opposed to putting the complex SQL into the CREATE TABLE statement itself). The reason for this is that it will reduce maintenance costs in the long run to centralize any SQL into a permanent object that can be easily found (as opposed to all of the other places that the SQL may be tucked away but maybe impossible to find).

- It is generally a good idea to extract data to an agreed upon library or use a naming convention that makes the temporary copies of data easy to find and/or delete.

*(Case for the Autonomic DBA Assistant — Continued from page 6)*

## Boat anchors

Boat anchors are index objects that are truly **unused**. They're not used directly by HLL (i.e. RPG), for query implementation or query statistics. Dead weight, period.

In the case of keyed LFs, they may be a product of experimentation, one time data cleanup projects, failed architectures etc. System administrators dare not get rid of them for fear of impacting some critical business function. With SQL indexes boat anchors are even more widespread. The only purpose in life for an SQL index is to help the performance of queries (SQL & non-SQL). It achieves this by offering the query optimizer alternate query implementation plans and immediately maintained and accurate statistics. Due to the dynamic nature of the query optimizer, it is often the case that better access path exists and the index created for one purpose ends up never being used (as evidenced in our findings for many ERP implementations).

AutoDBA will show you exactly which index objects are unused and offer a staged phase out of unused index objects. The stages taken to get rid of the boat anchor index are:

- If threshold 1 is reached (i.e. 40 days), change its access path maintenance to delayed
- If threshold 2 is reached (i.e. 90 days), change its access path maintenance to rebuild
- If threshold 3 is reached (i.e. 13 months), delete the unused file

If you have a policy that no files are to be deleted, simply don't check the option to delete the file. Once the access path is switched to rebuild, all overhead of that index object is gone (the access path is no longer maintained by the DB2 when updates/inserts/deletes occur for the underlying physical file). And the nicest thing of all is, you don't have to effect any action if you choose not to, just turn us on and we'll take care of it autonomically!

## Suboptimal indexes

These are index objects that exist and are used but could be further optimized.

There is no single definition for a suboptimal index. Some examples include:

- select/omit indexes whose selection criteria includes high proportion of the underlying table's rows
- duplicate and redundant indexes that can be shared or deleted
- indexes with 4GB access path size
- indexes which could benefit from larger logical page size due to frequent serial access
- heavily used indexes that could be designated as clustered index candidates
- indexes that force access path incurring unnecessary disk I/O

This list can and most likely will be expanded in the future. The bottom line is you can squeeze more juice out of your database by making these relatively simple optimizations.

## Optimal indexes

Obviously if an index is optimal, it is being used and there is nothing we can do to it to make it work any better except to say, "Good job DBA!"

## Missing indexes

The pain of a missing index can be tremendous. Recommending needed indexes is where we have excelled in the past 10 years and nothing has changed in that respect. Wait, some things have changed. We now leverage all of the wonderful V5R4 enhancements IBM has introduced like MTIs and System Index Advice table and prioritize, filter and surface critical indexes you need at this very moment. You get a full set of CQE & SQE index advice, including EVI recommendations! If you don't want to wait to build that critical index, turn us on and we'll build it for you, autonomically!

Obviously, the focus of the first release of AutoDBA is balanced index management, but the underlying autonomics infrastructure is going to carry it forward and bring hundreds of other advice quietly sitting in thousands of pages of IBM redbooks and minds of our DB2 experts.

Don't fear DBAs, help is here and its name is AutoDBA! Way I see it, it's a no brainer.

Elvis

**www.inn-online.de / iNN-eNews.pdf**

http://www.inn-online.de

The first autonomic DB2 tuning tool is finally here...

- ✓ Expert advice to manage unused logical files and SQL indexes
- ✓ Optimize select/omit and logical files in existing databases to improve performance
- ✓ Advanced prioritization methods to expose changes with the highest “bang for the buck”
- ✓ Reduce database disk I/O for batch jobs so they finish quicker
- ✓ Right-click UNDO support for all actions taken

**For a 45-day trial, please contact trial@centerfieldtechnology.com**

# SUCCESSFUL BUSINESS INTELLIGENCE

ALAN JORDAN. ROCHESTER, MN www.coglinmill.com

The recent IBM announcement that Query/400 is being retired and replaced by the new DB2 Web Query for System i product powered by technology from Information Builders has caused quite a buzz.

Feedback from IBMers and some of our customers makes it clear that there is a renewed interest in business intelligence on System i – and this new product announcement is very timely. While this new product with its web based interface and WYSIWYG report builder is certainly a big step up from Query/400, you need to look beyond the marketing hype if you plan to get serious about business intelligence (BI).

A good query or OLAP tool is certainly an important part of a successful BI initiative. It's the piece that most business users are familiar with - however it is just the tip of the iceberg. Unlike icebergs though, it's the absence of anything below the waterline that will sink your BI initiative.

Talk to any organization that has implemented a successful Business Intelligence initiative. Without exception they will all tell you that you need to first build a well architected data warehouse or data mart infrastructure. Read a book. Listen to the industry gurus. They all say the same thing.

But one of the major causes for long term failure of Business Intelligence projects is the failure to build a solid **architectural foundation**. So why is it that many organizations get it wrong? Part of the reason for this must fall squarely on the shoulder of the BI tool vendors. Almost universally they tout that their product is all you need. The vendor wants a quick sale and downplays the need for a real data warehouse or ETL tool. Equally, the customer is eager to keep the cost down, so is easily persuaded.

In the beginning the new query tool starts delivering on its promise: you've created a few reports or dashboards that pull data directly from your operational databases. Pretty soon, those few reports have grown to a few dozen. Business analysts are relying on them extensively and ask for additional reports. At some point this house of cards comes crashing down, usually for a number of reasons:

***Data quality:*** Let's face it, there are *known* data quality issues in your operational data. What about the issues you *don't* yet know about? Bad or missing data can lead to bad business decisions. You know where that can lead to.

***Complexity:*** Those first few reports were fairly simple. The new requests are not and may be impossible to produce. Data is stored in different databases, on different systems and in different formats.

***Security:*** How can I give Mary from accounting access to summary payroll data without her seeing the detail?

***Consistency:*** Why are my numbers different from yours?

***Compliance:*** That GL report is going to be used for financial reporting. How does that fit in with Sarbanes-Oxley? Should I risk my neck?

***Documentation:*** What reports are available? What do they mean? What data sources were used and how were calculations done?

***On-going*** maintenance becomes a huge resource and cost burden.

***Performance:*** All those end-user queries running against the production tables are killing the system.

The problem though, is that it doesn't happen overnight. The signs are there from early on, but a few band-aid fixes keep things humming along nicely: build a summary file here, write a quick RPG program there. Get Jim to conjure up the complex SQL to join those 7 tables together, hire someone new to help with the report backlog.

Suddenly, this is a large, mission critical application that stops delivering value. There are so many requirements that just can't be satisfied and it's a nightmare to maintain. You realize that you *have* built a data warehouse of sorts, or at least some data marts. But you did this without a plan and consequently it has no coherent architecture, no controls and very little documentation. You hope Jim doesn't quit!

So, now you know that you need to build a data warehouse, you need to decide how to go about it. First, you need to recognize that a data warehouse (or data mart) is much more than a collection of database tables.

*(Continued on page 11)*

*(Successful Business Intelligence — Continued from page 10)*

It needs some careful planning: Identify the business requirements. Design the database. Identify and describe the data sources, transformations, validations and business rules. When you've finished this task, you'll probably find it took at least twice the resource you expected (time, people, and of course cost), so you're behind schedule. At least you now have a set of requirements, an architecture and a plan. Now you need to implement it - quickly!

Building the database tables and indexes from the logical model isn't too hard. However, writing the Extract, Transform and Load (ETL) logic/code can be very difficult, especially if you expect to achieve all the requirements via the SQL language. SQL is great at pulling data OUT of a data warehouse. It's not so hot when it comes to putting data IN to a data warehouse. There are several reasons for this, but let's just look at just two of them:

**Validation**

To maintain data quality, you need to check for errors during the ETL process and somehow identify and set aside the 'bad' data, while loading the 'good' data. However, an SQL statement can only generate one result set. The only solution is to process the source data twice – once selecting the bad data and once selecting the good data. Very inefficient!

**Inserts vs updates**

You will undoubtedly need to handle both (hopefully automatically). SQL can perform one or the other, but not both in the same step. The common solution to this involves a multi-stage process, with intermediate staging tables. More inefficiency!

These and a myriad of other similar issues are the reasons why almost all Fortune 500 companies have built enterprise data warehouses using state of the art ETL and metadata tools. These tools significantly reduce development costs, reduce time to market and greatly simplify long term maintenance of the data warehouse. In fact, they are a key component of any successful BI initiative: whether you are a Fortune 500 company or a much smaller organization. Any organization will save time and money using an ETL tool, regardless of project size and budget.

Do yourself a favor and look beyond the desktop when scoping out your BI project. The data warehouse and ETL processes are the essential plumbing that will ensure trouble-free data flows to your end-user BI tools.

*Alan Jordan is a Senior Vice President with Coglin Mill (the developers of RODIN), based in Rochester, Minnesota. He has 12+ years of AS/400-System i data warehousing experience. You may reach him at 1-866-RODIN-DW or ajordan@coglinmill.com*

## training

www.centerfieldtechnology.com/training.asp

Join indexology guru Mark Holm as he presents important topics for database administration using Centerfield's tools. All training sessions are provided at no charge and open to the public unless otherwise specified.