THE CASE OF:
THE OVERLOOKED INDEX

One of the most frustrating aspects of tuning a database to run SQL statements quickly is choosing which indexes to create. A very simple approach used by many IT shops is to either run their queries with debug turned on or by using the PRTSQLINF command on a program or SQL package. This information may surface messages from the DB2 query optimizer that recommend a particular index be built to improve performance. The database or application administrator can take those recommendations literally and create the index under the assumption the optimizer will use it next time the query is run. Much to their surprise, the optimizer may not use the index and instead choose the same approach as before – seemingly overlooking the index.

The natural question is "if the optimizer itself recommended the index, why won't it use it?"

To answer that question, you have to understand a little about why the query optimizer chooses an index. Essentially, the optimizer looks at many different methods to find and read data from your tables and chooses what it believes to be the least expensive approach. The order and priority of which method to use is different between the Classic Query Engine (CQE – the only optimizer available before V5R2) and the SQL Query Engine (SQE – the second optimizer added in V5R2). Both query engines use cost formulas to calculate the estimated time to use (or not use) one or more indexes. If those cost formulas return an estimate that shows that a newly created index is more expensive than either a table scan or using a different index, then the recommended index will not be used.

So why would the optimizer recommend the index in the first place? The answer to this question boils down to the fact that the optimizer is **guessing** the new index would be cheaper than the method it ultimately used. Once the index is built, information is now available to the optimizer that gives it **actual metrics** about the cost of the index. These actual costs include the number of rows expected to be returned by reading the index, the number of disk I/O's required to navigate the index structure, the number of disk I/O's required to retrieve the actual data, and the CPU needed to do all of that work. If the estimated number of rows is high (i.e. more than 20% of the table) or the size of the index was larger than expected, the cost to use the new index could easily outweigh other alternatives.

Should you immediately delete the new index if it isn't used? The answer is maybe and maybe not. If you do not notice improved performance in your queries the answer is yes, delete the index. However, you may see improved performance even though debug messages, PRTSQLINF messages, or insure/ANALYSIS reports indicate the index is not being used. The reason for this is that the optimizer may have used the index to make better decisions because it could more accurately estimate the number of rows to be returned by the SQL statement. The optimizer can do this by peeking in the index itself and getting an estimate of the rows expected to be selected. This new information may provide the key information to help the optimizer choose the most efficient method.

FORMER UMBRELLA	FORMER PRODUCT NAME	CURRENT UMBRELLA	CURRENT PRODUCT NAME	IBM SERVER PROVEN®
DATABASE ESSENTIALS	INDEX ADVISOR	insure/SQL	insure/ INDEX	YES
	ANALYSIS	insure/SQL	insure/ ANALYSIS	YES
	CLIENT SERVER MONITOR	insure/SQL	insure/ MONITOR	YES
		insure/SQL	insure/ MONITOR for OneWorld®	YES
	CLIENT SERVER SECURITY	insure/SQL	insure/ SECURITY	YES
	QUERY PATROLLER	insure/SQL	insure/ RESOURCES	YES
		insure/ TOOLS	disk/ HUNTER	YES
		insure/ TOOLS	delta/ TRACKER	YES
		insure/ TOOLS	iSeries database/ ASSESSMENT	YES

What's in a Name?
HISTORY OF CTI'S TOOLS' NAMES

When the company was incorporated in 1997, its original name was Bradenmark. Under the Bradenmark name, its first product was created for the ISV market and called CD-ROM Studio for the AS/400 (later to become CD-ROM Studio for the iSeries). In late 1998, it was decided to change the name from Bradenmark because it was difficult to remember and spell correctly. In 1998 the name of the company officially changed to Centerfield Technology, Inc.

Centerfield TECHNOLOGY

strategic tools for strategic applications *teams*

A Beautiful View

If the average end-user were to look at an application's database, could they determine what data to query if they wanted to write their own report? *Of course not!* There are significant barriers for anyone trying to understand the information in a production database. The most noteworthy challenges include:

♦ The names of the columns are short and cryptic. Even using nice graphical tools, the end-user may not be able to determine what a name means because it follows a convention used by the application rather than an intuitive label.

♦ There are thousands of tables and columns to search through. Complex applications have complex databases. As a result, it is not uncommon to have a very large number of files in production. Furthermore, it is very common to have information in the tables so applications operate properly but not be of interest to users.

♦ The data is encoded. For example, a column that contains information about what state a customer lives in may contain numbers between 1 and 50 instead of using the state abbreviations.

♦ A single business entity or event is distributed across multiple tables. Unfortunately, the table relationships are rarely documented to the level they need to be in order for users to understand the data.

♦ The data may be very granular, but for reporting purposes it needs to be coalesced to be properly formatted. For example, a report may need to have a customer's first and last name in the form "last name, first name." If the table contains a column for first and last name, the report will require the columns be concatenated properly.

♦ The tables may contain private or sensitive information that needs to be protected from all but authorized users. For example, if a table contains credit card numbers, it is important to prevent unauthorized table downloads containing that column.

Beginning with this newsletter, we'll start a series of articles on how to solve these problems in a very elegant way through the clever use of SQL views. We'll also provide some practical examples of each technique. In this issue we'll talk about solving the problem of poor names and the large volume of tables and columns.

New paint job

So what is the best way to give those difficult-to-understand names a new look? First of all, lets discuss where a user can get descriptive information about a column. For an iSeries table, this 'metadata' can come from the column name itself, column headings, and column text.

Lets say for example you have a physical file that has the following fields:

Field Name
CID
CFNAM
CLNAM
CADDR1
CADDR2
CCITY
CST
CZIP
CTYP

Now lets create an SQL view over the top of this physical file (or SQL table, it doesn't matter) in order to clarify what the columns mean. The following SQL statement could be used to do that:

```
CREATE VIEW CVIEW/Customer_Master
("Customer ID" FOR COLUMN CUSTOMERID,
"First Name" FOR COLUMN FIRSTNAME,
LAST_NAME FOR COLUMN LASTNAME,
ADDRESS_1 FOR COLUMN ADDRESS1,
ADDRESS_2 FOR COLUMN ADDRESS2,
CITY FOR COLUMN CUSTCITY,
STATE FOR COLUMN CUSTSTATE,
ZIPCODE FOR COLUMN CUSTZIP,
CUSTOMER_TYPE FOR COLUMN CUSTTYPE)

AS
SELECT CID, CFNAM, CLNAM, CADDR1, CADDR2, CCITY,
CST, CZIP, CTYP FROM cview/cstmst
```

This view creates two sets of names. The first names become the column headings and the second names (the ones specified after FOR COLUMN) become the field names. If you run a query like SELECT * FROM CVIEW/Customer_Master in interactive SQL you will get a screen similar to the following:

asktheexpert@centerfieldtechnology.com

Lifecycle of the QZDASOINIT Job

I participate in couple of iSeries technical forums on occasion, and even though IBM offers quite a bit of information on the way iSeries handles ODBC & JDBC requests, questions on these jobs always resurface. Just recently, I've seen the following two questions and thought I'd comment on QZDASOINIT life cycle for those unfamiliar with the answers.

Q1: How can I adjust the time a QZDASOINIT prestart job stays in TIMW status?

Q2: Recently we realized that there are QZDASOINIT prestart jobs remaining on the system for days and weeks at a time in a DEQW state. Shouldn't they leave the system when the job is done?

QZDASOINIT jobs serve the purpose of connection pooling for database host server connections on the iSeries. There are usually a number of them started on the iSeries in the subsystem QUSRWRK as PJ (prestarted jobs), waiting for the JDBC/ODBC requests to come in. Once there is a new connection request these jobs change to RUN state then to DEQW state, waiting for the next SQL request from that JDBC/ODBC connection.

Dequeue wait (DEQW) is normal state for the ODBC database server jobs (QZDASOINIT) when the iSeries thinks that the network connection is still valid. It's basically waiting for new SQL requests to serve. Once an application (i.e. Websphere app running on Linux), is done with the JDBC/ODBC connection, it is supposed to properly close the connection. This would cause the QZDASOINIT job to go back to prestarted wait state (PSRW state in WRKACTJOB SBS(QUSRWRK) -> F14).

Timeout wait (TIMW) status will be seen in jobs while they wait to end. These jobs had clients connected that

are no longer communicating with the iSeries. This status usually suggests that the client (i.e. PC) has terminated the connection on its end without sending the close request to the iSeries. The operating system will send the packet requiring a response to the client and also start a timer that varies depending on the TCP setup. If it doesn't receive a response in that time period, the job will end.

Here is an illustration of the job status changes I have observed:
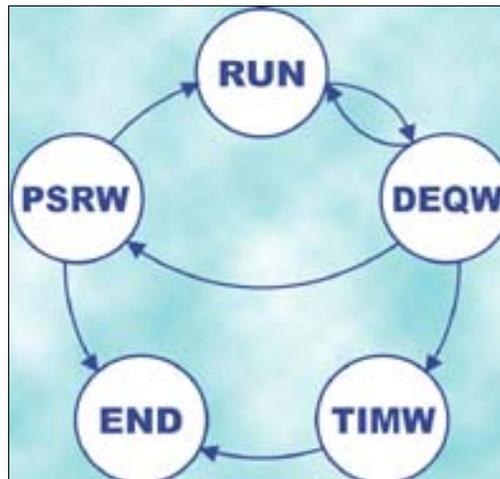
With this information in mind, let's tackle the original questions.

Answer 1:

TIMW status normally indicates an application problem. In most cases, the client application is closing the connection prematurely, at least from the iSeries perspective. Accordingly, I recommend some client side debugging should be performed.

In this particular instance, it turned out problems were due to the fact that they were using the new .NET data provider supplied with V5R3 Client Access. Their developers switched the code to use the traditional OLEDB connector all started working as expected.

Answer 2:

Extended DEQW status (days) indicates that application is not terminating the connection properly. If app was closing the connections, active QZDASOINIT jobs would go back to prestart-ed job state (PSRW) and wouldn't be seen in the WRKACTJOB SBS(QUSRWRK) display unless F14 was hit.

Alternatively, these jobs may end once their reuse count is exceeded (200 by default - new jobs would be started) or when there are too many prestarted jobs started (i.e. demand for them has fallen off).

A ccordingly, I recommend some client side debugging for this scenario as well, paying particular attention if "close JDBC/ODBC connection" method is being invoked in all circumstances.

Fun Reading!

Elvis Budimlic
Development Director

Centerfield
TECHNOLOGY

ROUTE SPECIFIC ODBC/JDBC USERS TO AN ALTERNATE SUBSYSTEM

On occasion, customers have a need to route specific users or specific application servers to a subsystem other than QUSRWRK or in older releases QSERVER. Most often this is for real world reasons like an application that requires jobs to be multithreaded, run specific users under lower priority, allocate more memory to the power users running ODBC/JDBC requests. I also find it a useful trick for diagnosing specific user's or specific server's issues.

Here are the steps to route specific IP addresses to a user defined database jobs subsystem.

♦ Perform WRKSHRPOOL to find a suitable memory pool (note memory size and activity levels) for the new subsystem. You will use that shared pool name in the instructions below.
♦ CRTSBSD SBSD(QGPL/ODBC) POOLS((1 *SHRPOOLx)) TEXT('Alternate ODBC subsystem')
♦ CRTDUPOBJ OBJ(QPWFSERVER) FROMLIB(QSYS) OBJTYPE(*CLS) TOLIB(QSYS) NEWOBJ (QPWFODBC)
♦ CHGCLS CLS(QSYS/QPWFODBC) RUNPTY(35) **Or whatever RUNPTY you need**
♦ ADDRTGE SBSD(QGPL/ODBC) SEQNBR(1) CMPVAL (*ANY) PGM(QCMD) CLS(QSYS/QPWFODBC) POOLID(1)
♦ ADDPJE SBSD(QGPL/ODBC) PGM(QZDASOINIT) INLJOBS(21) THRESHOLD(20) ADLJOBS(5) CLS (QSYS/QPWFODBC)
♦ STRSBS SBSD(QGPL/ODBC) ** you'll need to add this step to the system startup program defined in system value QSTRUPPGM; use RTVCLSRC to retrieve the existing program's source, append this command to it and recompile the cource **
♦ Start iSeries Navigator GUI client
♦ Click on: systemName->Network->Servers->iSeries Access
♦ Right-Click on Database and select Properties
♦ Click on Subsystems tab
♦ Click Specific Clients and click Add Client
♦ Fill in the Add Client screen specifying either IP address range or specific IP addresses
♦ Click OK ** changes take effect immediately **

Any user connecting from the IP addresses you have specified in the step above will be routed to the newly created ODBC subsystem. I have used this setup to route only my jobs to the new subsystem, so I can easily experiment with the resource allocations and other job settings during early phases of application development.

If you need to back out of this configuration, just remove the

(A Beautiful View—Continued from page 2)

"Customer ID"	"First Name"	LAST_NAME
1	John	Doe
2	Jane	Anderson

If you wish to use a different name for the column heading you can use the following SQL statement to replace the existing text:

LABEL ON COLUMN CVIEW/CUSTOMER_MASTER
("Customer ID" IS 'Customer Identifier')

Customer Identifier	"First Name"	LAST_NAME
1	John	Doe
2	Jane	Anderson

Once this statement is run, the column has three "names" – the column name itself, the column heading, and an alternate name ("Customer ID"). Whatever the tool, descriptive text can be assigned to column names to give them meaning to users.

Narrow the view

As mentioned earlier, complex applications often have thousands of files and columns. Along with providing more usable names, SQL views can be used to limit a user's view of the database to make it less overwhelming and focused for their particular needs.

One approach is to create a library for a user or group of users with similar requirements. This library would be their entry point into the database. Within this library, simply create views over the interesting files and useful columns. Like logical files, views can be constructed to include a subset of the columns and eliminate fields used just by application processing.

An additional advantage of this technique is for security. We'll discuss this aspect in a later column.

Once the library and views have been created, simply provide information and access to the library to your users. By eliminating the clutter and providing useable names, end-users will be much more effective using query tools.

In our next column we'll talk about using views to further simplify a production database for users by showing you how to build in join criteria and turning cryptic data values into human readable values.

above listed IPs and end the ODBC subsystem.

This particular configuration nicely demonstrates Usage of time proven iSeries work management with the new functionality available in iSeries Navigator.

Reader Input

In our April newsletter article "DDS or SQL … which is better" we focused primarily on functional differences between DDS an SQL defined files. We received a note from DB2 UDB for iSeries expert Kent Milligan asking us to address some performance implications of SQL defined files.

SQL Table

We mentioned that data validation in SQL defined tables (physical files) is done when the record is written (insert, update). We did not, however, emphasize that since data validation is done on the writes, reads of SQL defined tables don't have to do it, hence the added performance benefit on the read requests. All of our experience with end user application tells us that there is a great discrepancy in favor of reads in most user applications, strengthening the case for using SQL defined tables.

SQL Index

SQL defined index default logical page size of 64KB. In DDS, default size is 4KB, but it could be 8KB or even 32KB. This gives obvious benefit to SQL defined indexes where most of the index entries will be accessed by the request (i.e. query), as storage management is able to bring in larger chunks of index in a single request.

Seasoned iSeries veteran Dan Cruikshank wrote a very informative article on this particular topic. Article can be found at the following link:
http://www.iseriesnetwork.com/artarchive/index.cfm?fuseaction=viewarticle&CO_ContentID=20067&channel=art&subart=auth&authid=94

asktheexpert@centerfieldtechnology.