

MTI vs IDX



CQE temporary indexes

If you've ever signed on to green-screen command line session, chances are you've probably used the WRKACTJOB command. While using it, have you ever seen "IDX" in the Function column?

If so, that was probably your first introduction to CQE (the Classic Query Engine) building a temporary index. You probably also noticed that CPU usage jumped when this happened. If you've ever built a keyed LF you may have seen the same behavior (IDX & CPU jump).

What's going on is that DB2 is building a data space index object for you. The bad news about CQE temporary indexes is that they're not reusable across different jobs and on occasions not even within the same job. If they're built over a large result set, they consume large amount of temporary disk space and end user's query has to wait to run until that build is done. A more frequent case is that CQE temporary indexes are small but their cumulative cost piles up (they may be built every time the query runs). There's no other way to say it: CQE temporary indexes are bad news!

So why does CQE build temporary indexes? In my opinion, DB2 for i5/OS is too forgiving. It'll run any query you throw at it. If it can make it perform well through use of indexes and temporary structures, then all is well. If indexes are not available and temporary structures (i.e. hash tables) are too expensive to build, it'll weigh the cost of one or more table scans versus a temporary index build. With CQE, temporary indexes frequently win out.

The good news is that there is an easy workaround for this issue. Negative implications of CQE temporary indexes can be remedied by simply building a permanent index instead. You can figure out the keys needed for a CREATE INDEX statement through use of IBM database monitor, iSeries Navigator's SQL Performance Monitors, Visual Explain, and System Index Advice table (V5R4)... or if these options sound like too much of a learning curve consider letting Centerfield Technology's HomeRun toolset take care of it

MTI

So far, I've tried to make the case that CQE temporary indexes are bad news. I assure you the IBM lab was aware of the negative implications CQE temporary indexes had well before we discovered them. In their long standing tradition, they did something about it. With V5R4 the smart folks at the blue buildings in freezing Rochester MN introduced MTIs – Maintained Temporary Indexes.

As I've mentioned a number of times before, IBM's strategic query engine is SQE. On V5R2 & V5R3 SQE made use of improved temporary structures (lists, sorts, hash groups...) and improved algorithms to provide fantastic performance enhancements, but it did not leverage temporary index support. This was an obvious hole as index implementation often outperforms all other available implementations. The great news about MTIs is that they're built only after careful consideration and are reusable across all jobs on the system! Once it's built, the MTI is most likely going to stick around until you IPL. The query optimizer can use it for both statistics and implementation. It's not directly tied to a file in a cross-reference context hence it does not cause functional issues to your applications (i.e. RMVM will not fail due to a dependency on the physical). Any query can leverage it. Ahhh.... all's well with the world now.

Well... you're not totally off the hook yet. While this fantastic enhancement is fully autonomic and inline with long standing System i tradition of "taking care of things for you", there are couple of things to consider with MTIs. They are not built over large tables so you still need to be proactive about building indexes using the sources I mentioned in the previous section (including SQE Plan Cache on V5R4). As I mentioned previously, MTIs are deleted at IPL time so you may experience a post-IPL warm-up effect for the queries that leverage MTIs. The obvious workaround for this is to build them as permanent indexes, as you've done with CQE temporary indexes. A tools that can help you with this is iNav's Index Evaluator.

(Continued on page 3)

Inside this issue:

COVER STORY: MTI vs IDX	1
System i Database Modernization to SQL Now Automated	3
What Was That Name?	4
Page Up Page Down	7



**HAPPY
HOLIDAYS**

*from the
FIT Team*

I live in New England. My very first house was a post-and-beam colonial built in the 1790s, a testament to the fact that houses from that era were built to last. They're also packed with charming little quirks that their owners quickly come to accept and sometimes even love. We got so used to our house's little quirks that we didn't even see them anymore. But the quirks were still there. And, over time, as our needs changed, some of the quirks became problematic. We had to modernize.

In many ways, my first house was like our beloved System i and its native database.

Most databases running on the System i today were built in the traditional way, using Data Description Specifications (DDS). The DDS approach to creating databases is solid and quite serviceable for applications written in RPG or COBOL. But the fact is, it's quirky. No other applications in the IT world speak DDS. Everyone else uses SQL to define and control access to databases. And IBM has officially stated that moving to SQL is part of the System i Developer Roadmap, and that its investments in improving database access will be through SQL, not DDS.

SQL has become the industry standard for good reason. In most cases, SQL databases offer significant advantages over

DDS databases. Some of those advantages are improved application response times (roughly 20% on average according to experts); greater data integrity; compatibility with modernized applications, particularly those using Service Oriented Architecture (SOA); and support for advanced database functions and data types. You can learn more about the advantages of modernizing your DDS-generated database in the IBM Redbook [Modernizing iSeries Application Data Access](#).

According to Kent Milligan, Senior DB2 for i5/OS Specialist at IBM, his team has worked with many System i developers over the past several years to modernize their database definitions and access methods to SQL as they navigate the [System i Developer Road Atlas](#). "SQL brings so many real benefits to System i applications that it should be a no-brainer to update your database structure to benefit from the additional data types and performance advantages. However, it does take time for clients to migrate all their database definitions to SQL -- as well as build an understanding of DB2 and SQL -- and those have proven to be hurdles for some companies," Milligan says.

(Continued on page 6)

(MTI vs IDX — Continued from page 1)

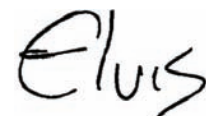
Alternatively, if you'd like to prioritize your index advice in a database and application sensitive manner and consider indexing from a systemic perspective, consider Centerfield's AutoDBA feature.

Conclusion

Why am I bringing temporary indexes and proactive index management up now?

I've seen real excitement about IBM's DB2 Web Query product in the user community. I suspect that this will cause a number of traditional Query/400 users to switch and start using the new product. You may want to convert your existing Query/400 queries that are using CQE for implementation and use SQL for newly created query objects (most likely using SQE for implementation). Report writers often forget little fact that they need to build indexes proactively if they're to keep their end users happy. I mean, what's the value of those nice web reports if an end user has to swallow a performance pain pill when they run it?

Be proactive. Replace temporary indexes with permanent ones and you will provide benefit to any query workload hitting your critical database files.



What was that name?

Users often have the need to search databases even when they don't have the key data to locate information. For example, a customer support representative may need to find a person's records even though they don't know how to spell their name. Programmers or database administrators have a couple of ways to provide "fuzzy searches" and allow their users to find information with partial data.

Method One

One common method is to allow users to guess at the spelling of a name or word. This guess is then put into an SQL statement used to search the table. Let's say for example that the user want to find a customer whose name they think starts with the letters 'McC'. The programmer builds an SQL statement similar to the following:

```
SELECT custLastName, custFirstName, custZIP FROM CUSTOMER WHERE custLastName LIKE '%McC%'
```

The application would then be presented with data similar to:

custLastName	custFirstName	custAddress
McCabe	George	55901
McCallen	Jane	92902
McCallen	Dean	22190
McCree	Lynn	82208

This approach can work well if the user of the application can make a good guess at the spelling of the name. If they are inaccurate with even one letter none of the returned rows would contain the right customer. Furthermore, if the user had the correct letters but didn't have the correct case they would not find the correct information. To avoid searches not finding data because of case issues the SQL can be changed to the following:

```
SELECT custLastName, custFirstName, custZIP FROM CUSTOMER WHERE UPPER(custLastName) LIKE '%MCC%'
```

In this case both the input string and the column containing the data are upper-cased. This eliminates the chance the data in the table does not match the input string and the situation where the end user simply guessed wrong about capitalization.

There are several disadvantages to this technique related to performance. When a column in a table is transformed by a function, an index will typically not be used to make the search faster. Because the data in the index is in the same form as the data in the column, it is not possible for the query optimizer to use the keys in the index to compare to transformed information. Even if the column was not upper cased, the use of the LIKE comparison with an argument with a leading wildcard can not use an index. Index searches require that a search argument be compared to the leading part of the key. In the case of leading wildcards, the search pattern precludes the use of the leading part of the key. A final performance issue with LIKE relates to the two query optimizers. Prior to V5R4, the Classic Query Engine (CQE) will be used for all SQL statements that contain a LIKE predicate. Because the CQE optimizer is typically slower than the SQL Query Engine (SQE), SQL statements that contain LIKEs will run slower on average.

(Continued on page 5)

Method Two

A second approach to doing “fuzzy” string searches is to use the SOUNDEX and DIFFERENCE functions in SQL.

The SOUNDEX function takes a string and generates a four byte code that represents the sound of the string. This code can be used to compare the sound of one word to another. As you can imagine, users often type in search values that may sound like the word, but may not be spelled anything close to the actual word. For example, here is the output of a sample query with several different words in the table and their associated SOUNDEX values.

<u>WORD</u>	<u>SOUNDEX (WORD)</u>
Mark	M620
Marc	M620
Marcus	M622
Jon	J500
John	J500

You can see that the SOUNDEX function produces a value for each word that can then be used to search for similar sounding words similar to the following query:

```
SELECT word FROM sound WHERE SOUNDEX('Marq') = SOUNDEX(word) .
```

The output of this query using the data from above would yield the following rows:

<u>WORD</u>
Mark
Marc

This example would require that the user type in a spelling that

was very similar to the word they were looking for. What if you wanted to be a bit more lenient about the spelling? The answer is to use the DIFFERENCE function. This function returns a value from 0 to 4 indicating how close the words are phonetically. A value of four is the best possible match, so if you want a wider range you can select a lower difference value. For example, the following SELECT statement would select more data than the previous one:

```
SELECT word FROM sound WHERE DIFFERENCE ('Marq',word) >= 3
```

Returns:

<u>WORD</u>
Mark
Marc
Marcus

Performance considerations

When using either the LIKE, SOUNDEX, or DIFFERENCE function, you must be careful about the performance of your SQL statement. As mentioned earlier, if you use a function on a column in your table, an index over that column will typically not be used which can have a disastrous impact on the speed of the query.

One way around this problem is to actually store an alternate form of the data in the table. For example, if SOUNDEX searches are performed often, then it may be a good idea to keep the actual SOUNDEX value in the table so that an index can be created over that column to be used to speed up searches. To implement this with minimal impact to existing applications you can use a database trigger to generate the SOUNDEX value and then have the trigger fill in the value. In most situations you would need both an insert and update trigger to accomplish this, but they could both use common code to reduce the implementation cost.



CHH Consulting is a UK IT consultancy and software company based in Manchester, England. It focuses on practical and affordable solutions in the areas of Applications, Security, Availability and Virtualization for the iSeries/System i. Its solutions cover the main iSeries/System i Server together with all Wintel LAN/WAN security solution requirements re: the Internet.

Centerfield welcomes CHH Consulting as our newest Partner!

+44 (0)1925-758995

**Email: sales@chhconsulting.com
www.chhconsulting.com**

(System i Database Modernization to
SQL Now Automated - Continued from page 3)

Enter [Resolution Software Ltd.](#), a 20-year-old veteran in database design and modeling best practices, and [Xcase for System i](#). Xcase the first database modeling tool to automatically upgrade traditional System i databases so they are fully compatible with SQL – without impacting existing applications. It allows companies with data on the System i to quickly and cost-effectively modernize their databases so they can fully implement any SQL function across new and existing applications as needed.

Until now, developers that wanted to modernize their databases had two options: they could either modify all existing applications that relied on the modernized database table, or use a complex manual process to modernize the table to SQL in a way that preserves compatibility with existing programs. In both cases, the process was long, intricate, and error-prone. If the application impacted data across multiple tables, all tables or related programs required updates.

Milligan says that the time and expertise required for manual conversion are no longer relevant. "Xcase removes those hurdles by providing a solution that simplifies management of the DB2 objects involved in the database modernization process and speeds up the process by employing IBM-recommended best practices that allow existing applications to keep running with no modifications, as if no change occurred to the database definitions."

Xcase for System i consists of two components: one to address the database migration process and one to address ongoing modifications to the SQL database. The Modernization Module prepares tables within a DB2 for i5/OS database to accept I/O from any application using SQL functions, while it preserves their ability to communicate with existing applications that use native I/O methods. To accomplish this, it follows the IBM-recommended process laid out in the Redbook titled *Modernizing iSeries Application Data Access*. The Xcase tool requires no special knowledge of SQL.

Xcase also includes an Evolution Module that helps developers modify their modernized database quickly, in a controlled way, as needed. It reverse engineers your database structure into graphical data models so you can see relationships at a glance. The models show both SQL and DDS-generated objects. This can be particularly useful in the System i market, where only an estimated 7% of legacy applications have documented data models. Xcase's Evolution module also performs impact analyses to show the implications of any given change to SQL database objects; meticulously tracks impacted views, triggers and stored procedures; and automatically generates the code needed to implement some or all of the changes. For reporting, Xcase offers an easy way to define the information you require and then generates the required view using SQL.

Resolution has refined the Xcase technology according to database design and modeling best practices for nearly 20 years. This mature technology is now available with a Modernization Module and other enhancements to meet the unique requirements of the System i platform.

We're excited about bringing Xcase to the System i market," says Elie Moyal, Resolution's CEO. "So many developers are discovering the untapped potential of SQL. Xcase will help them leverage SQL's power faster and with less effort, and still preserve the option of using native I/O methods whenever they want. Modernizing to 100% SQL compatibility dramatically increases the System i's profile as a database server. When you pair the System i's reliability and security with the power, speed, flexibility and data integrity of SQL, you create compelling business reasons for management to look at consolidating all of its important data onto the System i."

Xcase for System i will be generally available in January 2008. Resolution will also offer a Modernization Service beginning December 2007.



Historically, there are number of ways System i developers have implemented subfile pagination. The same goes for web developers adding *page back* or *page forward* support, except their challenge is even greater due to the stateless nature of web development.

If you're lucky, there is a primary (unique) field in the table that you can use to specify a range of rows. For example, getting a result set *BETWEEN thisValue AND thatValue* where the column values are unique is straightforward. If there is no primary key field (and management does not allow you to introduce a surrogate key field into the table), scrollable cursors are another option. Scrollable cursors provide *fetch next* & *fetch relative* features which make paging up & down relatively straightforward. This type of implementation is trickier to implement in a stateless environment since the web connection cannot maintain a persistent database connection required for scrollable cursors.

There are other ways to implement the page up & down requirement, including temporary work files or, on the web side of things, using stateful session beans or HttpSession. Let's ignore these alternatives for now and talk about a DB2 method to address this common requirement.

ROWNUMBER() built-in

The iSeries V5R4 release introduced several new SQL OLAP functions. My favorite is the *ROWNUMBER()* built-in. Here is another solution to pagination that will leverage this cool built-in.

Let's say your query is:

```
SELECT FirstName, LastName, Address, Salary, Rating
FROM myFile
WHERE City = 'Rochester'
ORDER BY Salary DESC, Rating ASC
```

You want to display people with top salary and run on the first page, followed by people with lower salary and rank. You're taking advantage of dynamic ordering SQL offers to sort the result set just the way you need it. Let's say this query returns 1000 or more records, but you only want to show 20 per page (humans quit paying attention when faced with an overwhelming amount of data). If you're in a stateless session, your first call is relatively easy to make by appending *FETCH FIRST 20 ROWS ONLY* to the statement. However, since you're not using scrollable cursors, how will you fetch next 20 rows, and next after that and so on? Enter the *ROWNUMBER()* built-in.

Let's modify the original statement to work with your requirement:

```
WITH cte1 AS (
    SELECT FirstName, LastName, Address, Salary, Rating,
           ROWNUMBER() OVER (ORDER BY Salary DESC,
                             Rating ASC) AS NextRow
    FROM myFile
    WHERE City = 'Rochester')
SELECT * FROM cte1 WHERE NextRow BETWEEN ? AND ?
```

Your code only needs to provide the range arguments i.e. between 20 & 40, then 40 & 60 etc. That's it, you're done.

Conclusion

This example illustrates how even a stateless connection can achieve result set pagination by leveraging the new *ROWNUMBER()* built-in -- and essentially introducing a surrogate key for your result set. Your web page can now make calls until it exhausts the entire result set as the user hits the Page Up & Down keys or clicks on Back & Forward buttons on the webpage.

Elvis