

Out in Left Field

strategic tools for strategic teams Volume Two, Issue Six



I/O, I/O, its off to the disk we go

How to avoid the Dopey I/O

When Walt Disney envisioned the Seven Dwarfs singing as they sauntered off to work, it is not likely that disk drives crossed his mind – but if you are in IT they may be on your mind often as you go to work.

Today, we need to think a lot about disks and the associated input/output because it is often the biggest performance bottleneck we have in our applications. Because processor performance continues to grow at an incredible pace while the speed of disk drives has improved at a much slower rate, eliminating unnecessary disk I/O has become essential.

Disk I/Os come in two basic flavors – reads from the disk and writes to the disk. Each of these types can be done synchronously (done while you wait) or asynchronously (done while you get something else done). The purpose of this article is to expose cases where unnecessary disk I/O is being done, how to find it, and how to avoid it.

Dopey I/O

There are several types of disk I/O that are completely unnecessary and can be totally eliminated.

The first of these is the use of the FRCRATIO (physical files) and the FRCACPTH (physical and logical) values. The FRCRATIO value should be *NONE for physical files and FRCACPTH should be set to *NO for both types of files. These file settings require that DB2 sends changed records or access paths to disk as soon as possible. Because these I/Os are synchronous they can have a disastrous effect on response times.

The use of journaling is a much better alternative than the FRCRATIO for physical files. It not only performs better but also provides a much more robust way to keep your database safe from system outages. The use of the FRCACPTH parameter for keyed access paths performs very poorly and in general the system automatically ensures the access path is protected if it is very large.



Grumpy I/O

One of the most common sources of application performance issues continues to be file open and close processing. In SQL-based applications the cost of a “full open” is especially high. When a file is opened, new OS/400 objects are created. When a file is closed, those objects are destroyed. Both the creation and destruction of objects causes disk I/O to occur. If files are constantly being opened and closed, it hurts not only the speed of the running program but also the whole system because everyone is sharing the same disk arms.



Another common reason disk I/Os occurs when an application reads many records from a logical file based on data from another file. If the key is built using data from another file it often results in a random key probe into the logical file. This may lead to at least two disk I/Os – one or more on the index used by the logical file and another for the record itself. These I/Os cannot be completely eliminated but here are a couple ways to reduce them:

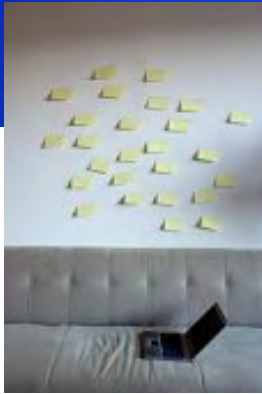
- Sort the data in the primary file by the key used to probe the logical file. This will allow the index pages to be processed sequentially instead of in a random fashion and allow the system to keep going back to memory-resident storage.
- Reorganize the physical file referenced by the logical file and specify that the order of the records should be the same as the key order of the logical. If the keys are then processed sequentially, the data in the file will also be processed sequentially.

Inside this issue:

COVER STORY:	
I/O, I/O, its off to the disk we go	1
Reorg Revival	2
The Smart DBA's tool	3
Reader Input	3
Some Index Usage No-Nos	5
Leverage and Tuning	7
Anatomy of a joined UPDATE	9

(Continued on page 8)

performance
is
everything



Reorg Revival

The RGZPFM (Reorganize Physical File Member) command has never been a favorite among iSeries customers. While necessary to reclaim space used by deleted records, the fact that applications could not be used during the reorganization has been a showstopper for its use. As the demand for 7x24 systems grows, IT shops have found it harder and harder to run the command even though the need for reorganizing files is increasing.

Recognizing the need to use this command while keeping applications available, IBM introduced an improved reorganize function in V5R3. With the Allow Cancel (ALWCANCEL) keyword, it is now possible to do a partial reorganization that fits into a smaller window or even while users read and update the same file. This has suddenly

made reorganization a realistic technique to use on systems that require high availability. Unfortunately, the command has yet to shed its bad reputation and is too seldom used to improve the performance of the many databases that need it.

The RGZPFM command is routinely used by most shops to simply shrink the size of files. A more subtle, but equally important, purpose is the command's ability to speed up performance. One good example is the case where there are many deleted records in the file. If the data is typically accessed through a keyed logical file, only the active records are visible to the application. But any deleted records that sit on the same physical disk page (4096 bytes) *will also be brought into memory with*

the active record. So for a file that has 150 byte records, the pages in memory for this file could be wasting up to 97% of that valuable space.

A second way to improve performance with RGZPFM is to reorganize the file in the key order. The sorted order can either come from the physical file itself if it is keyed or from a logical file. If your applications or batch jobs have a favorite logical file that is used to retrieve data sequentially you have an extra incentive to use RGZPFM. By putting the physical data in the same order as the index, random I/O can be dramatically reduced which will improve interactive response time and batch elapsed times.

(Continued on page 4)

the smart DBA's tool

Performance tuning is difficult. SQL performance tuning is even more difficult. Do I have a problem? Where do I start? What to do once I locate the problem? Good tools can make these difficult tasks easy.

If you're searching for a tool to help you analyze SQL access to your database, search no more. Centerfield's insure/ANALYSIS is the best tool for the job, period!

It works on the principle of *drilling into* data and correlating categories based on dynamically configured filters. In other words, you define interest points (filters) as you traverse the data. The more filters you define, the narrower the focus of your problem determination. With a few clicks you can zero in on the worst performing SQL query, program, stored procedure, job or even user session.

Once located, the inefficient object can be pulled into our Visual Explain tool (included) and dissected to learn why it's performing poorly. Changing the statement itself or the query runtime criteria can play out what-if scenarios. Indexes can be built to help query optimizer pick a better-optimized plan, and index use can be verified.

- What was the statement that caused insure/INDEX to recommend an index that will supposedly save 9 ½ hours in runtime?
- What program is running that nonsensical statement (SELECT * FROM WHERE 1=0) ten gazillion times?
- What are the top 5 worst performing queries on my system?
- Which user is driving the most SQL on my system?

Can you answer these questions today?

- Would you like to leverage the DB2 expertise of guys that live and breathe the System i database?
- Would you like to spend time doing things you enjoy rather than deciphering 'hieroglyphics' located in the system's database monitor output files?

Smart DBAs use tools to manage SQL access to their database. insure/ANALYSIS is that tool on the System i platform. It's a no-brainer really.

Elvis

Reader Input

[Re: Tabula Rasa April 2006](#)

Thank you!!!! This was exactly what I was looking for. Very good help.

Only thing that was different for me was step #3:

> select option 5 to "Install Licensed Internal Code and Initialize System"

Mine was option 2, but still had the text, "Install Licensed Internal Code and Initialize System".

Again, thank you!

B.C.—Utah

I enjoy your newsletter...great job !

J.D.—Missouri

I just read your news letter (i.e. No Brainer article). I would like more information about your database/ASSESSMENT tool.

T.H.—Michigan



Improve Your Performance

High Speed Tuning for High Speed SQL

Improve your productivity
Just sit there...
...and learn

[download](#) 

<http://www.centerfieldtechnology.com/pdf/High%20Speed%20Tuning%20for%20High%20Speed%20SQL.pdf>

What are the best options on the RGZPFM command? It depends on what you want to accomplish and how much time you have to get it done. The following chart shows how the parameters affect space, time and the final layout of the file. It is an abbreviated version of IBM's documentation (the full table can be found using the link at the end of the article).

Let's say you want to reorganize your files and reclaim disk space as well as reorder the data by the most popular logical file – all while allowing users to access the data. Here is an example of how that could be done:

```
RGZPFM FILE(PRODUCTION/ORDERDTL) KEYFILE(PRODUCTION/ORDERDTLL1 ORDERDTLL1) RBDACCPH(*NO)
ALWCANCEL(*YES) LOCK(*SHRUPD)
```

This command will reorganize the ORDERDTL file using the ORDERDTLL1 logical file. While it does that work, users will be

	ALWCANCEL(*NO)		ALWCANCEL(*YES)		
	KEYFILE(*NONE)	KEYFILE(*FILE or keyfile)	KEYFILE(*RPLDLTRCD)	KEYFILE(*NONE)	KEYFILE(*FILE or keyfile)
Concurrent access	No	No	Yes	Yes	Yes
Performance	Very fast	Fast	Very fast	Slower	Slowest
Temporary storage	Double data storage	Double data storage	Journal receiver storage	Journal receiver storage	Journal receiver storage
Index processing (non-KEYFILE)	Synchronous or asynchronous rebuilds	Synchronous or asynchronous rebuilds	Maintain indexes or synchronous or asynchronous rebuilds	Maintain indexes or synchronous or asynchronous rebuilds	Maintain indexes or synchronous or asynchronous rebuilds
Final row position exact	Yes	Yes	Only if LOCK (*EXCL) and not restarted	Only if LOCK (*EXCL) and not restarted	Only if LOCK (*EXCL) and not restarted
Amount of CPU and I/O used	Smallest	Next smallest	Smallest	More	Most

able to read and update the ORDERDTL file. If backups need to be performed at 1:00AM and the command is still running it can be canceled and the work done so far will be kept. We also chose to tell DB2 that all of the logical files should be maintained as the reorganization was being done rather than rebuilt from scratch. This ensures that if other logical files were needed over the file, they would be available to applications. One requirement to use this version of the command, however, is that the ORDERDTL file would have to be journaled.

Hopefully you can see the real benefit of using the new and improved RGZPFM command in your own shop. Now you can have your cake and eat it too!

For further information on this command here are several web links.

<http://publib.boulder.ibm.com/infocenter/iserics/v5r3/topic/dbp/rbafomngprgz.htm?resultof=%22%52%47%5a%50%46%4d%22%20%22%72%67%7a%70%66%6d%22%20>

<http://publib.boulder.ibm.com/infocenter/iserics/v5r3/topic/dbp/rbafodldbr.htm?resultof=%22%52%47%5a%50%46%4d%22%20%22%72%67%7a%70%66%6d%22%20>

HomeRun

6 tools + 1 Price = SQL Performance

SOME USAGE

ינהדזק הו-הוס

Have you ever seen “cost of this index is too expensive” in Query Optimizer Debug Trace or Visual Explain, and asked yourself, “Why-oh-why won’t you just use my ‘beautiful’ index?”

The answer? Only Query Optimizer knows! (Subconsciously, I might add) Figuring out all of the reasons why Query Optimizer rejects an index is a difficult task due to the complex algorithms Query Optimizer uses in its costing.

I will list some common scenarios in hopes you can avoid them in your development. Let’s start the countdown.

Numeric Conversion

I was taking the query optimization class from IBM and noticed that I wasn’t quite progressing in a manner outlined in the Visual Explain lab courseware. Namely, an index that I had just proudly and specifically built for the sample query *was not being used*. Frustration set in and I decided to dig in and figure out why.

The query had a WHERE clause like: ... **WHERE SMALLINT_FIELD > 3** ... and the index I had built was over SMALLINT_FIELD. There was no reason, in my mind at least, why the index shouldn’t be used. Only after looking at the query as rewritten by the Query Optimizer did I finally hit upon a solution.

Literal value 3 was implicitly cast to INTEGER data type (4 bytes in length) and the Query Optimizer was casting my SMALLINT_FIELD (2 bytes in length) to INTEGER in order to make it match the right-side integer value and perform arithmetic comparison.

“Geez,” I thought, “that just seems counterintuitive.” However, when you think about it, Query Optimizer is really only artificial intelligence and if you consider the complexity of, and all the possible ways to write an SQL query, it’s easy to see how “obviously” simple things like this one can work in what a human would consider non-intuitive fashion. Since I can’t change Query Optimizer’s implicit cast behavior for literal values, I decided to explicitly cast my literal to a SMALLINT in hopes that DB2 would only perform that cast once, match it correctly to my SMALLINT_FIELD data type and use my index.

Lo and behold, that’s exactly what happened.

No-No #1: Thou shall not perform numeric conversions (implicit or explicit)

Arithmetic Expressions

Have you heard of a saying “When the only tool you have is a hammer, everything starts to look like nail head”? Likewise, as one gains experience with SQL and learns of all the power embedded within, it’s easy to fall into the trap of thinking that SQL is THE tool for EVERY task.

Pretty soon you start seeing all kinds of complex calculations being performed right in the SQL statement itself. There’s nothing wrong with that, UNLESS it prevents the Query Optimizer from using an index to implement your query.

Say you’re coding embedded SQL in some HLL (Higher Level Language). Being a hotshot SQL developer you put as much code as possible into the SQL itself. So instead of doing something like:

```
int x = 1000 * 1.2;  
SELECT * FROM T1 WHERE COLUMN1 = :x;
```

You do:

```
SELECT * FROM T1 WHERE COLUMN1 = 1000*1.2;
```

Notice that in the first case COLUMN1 is compared against a static value of 1200 (value of x variable). In the 2nd case, value is calculated dynamically.

The bad news with the 2nd case is that it will prevent Query Optimizer from using an index for statistics to gauge best estimates in the optimization process, and that in turn **may** result in not using an index for scan-key positioning.

No-No #2: Thou shall not use arithmetic expression as an operand to be compared to a column

Character String Padding

The little known fact that DB2 for System i does not use an index if the constant value or host variable is longer than the column length is an easy thing to overlook in embedded SQL. For example, if you know all of the fields in your file are a character data type, and in your program you make sure your variable is long enough to fit the longest column length, then you decide to reuse that variable for comparison against other, shorter fields as well. Right there you shot any chance of using an index for implementation of your query.

(Continued on page 6)

A less likely example in interactive SQL is if you type something like:

```
.... WHERE LIBRARY = 'LIB_10_LEN ' ....
```

Notice that there is an extra space following the letter 'N'. We all know library name lengths are 10 on the System i, and the extra space is probably just a typo. However, it amply illustrates just how easy it is to make this type of error, which will result in a poor performing query.

No-No #3: Thou shall not use character buffers that are longer than column length

Leading Wildcards

RPG, C and COBOL programmers on the System i are familiar with use of partial keys. If you have an index on LASTNAME and want to look at all folks whose last name begins with BUD, you just specify your key length as 3. In SQL the same thing can be accomplished by using the LIKE clause:

```
SELECT * FROM T1 WHERE LASTNAME LIKE 'BUD%'
```

The good news is that DB2 will use an existing index and return your result set quickly. The same thing is true for 'BU%D%': the index will still be used for index-key positioning.

The problem comes when a leading wildcard is used, i.e.:

```
.... WHERE LASTNAME LIKE '%UD%' .....
```

A leading wildcard prevents DB2 from using any index-key positioning and causes non-use of this excellent performance aid. In fact, in this case, DB2 might not use an index at all, even though index-key selection can still be used.

Ideally, you should avoid usage of the LIKE clause whenever possible since it restricts you from using the SQE engine on releases prior to V5R4. Where you can't avoid it, given the fact that the number of wildcards you can use in your string is not restricted, you should always try to provide as much information as possible in your LIKE clause. In our example, if I really only wanted BUDIM* last names, I should have specified that as LIKE 'BUDIM%'.

No-No #4: Thou shall not use leading wildcards in a LIKE clause

FOR UPDATE OF Clause

In embedded SQL programming, it is possible to declare a cursor and append a FOR UPDATE OF clause to it, instructing DB2 that you intend to update some of the columns that are returned in the result set. This is perfectly fine, but it also means that DB2 will not use an index for these fields if they are being used in your WHERE clause.

No-No #5: Thou shall not unnecessarily include columns in the FOR UPDATE OF clause, unless they're truly updated

Comparison of Two Columns from the Same Row

Every once in a while you may be tempted to compare two columns in the same row, i.e.:

```
SELECT * FROM T1 WHERE PREVIOUS_SALARY = CURRENT_SALARY
```

Take note that in this case, usage of an index is not possible. There is just no way around it.

No-No #6: Thou shall not compare two columns of the same row unless it's absolutely unavoidable

So far, I have provided a list of common scenarios that can preclude DB2 from using an index to implement your query.

Sometimes though, you'll use Visual Explain or Query Optimizer debug messages to build an index that Query Optimizer itself recommends and your index will still not be used. What's this, Query Optimizer lying to itself? Not quite.

I can think of a few valid reasons why a recommended index doesn't end up being used, but this list is by no means exhaustive:

- 1) Up to V5R4, Query Optimizer recommended indexes focus on selection criteria only. There are three additional criteria that play a role in determining if an index is used or not: joining, grouping and ordering. So, once you build the recommended index, newly available statistics become available to Query Optimizer, which then realizes that the new index is not a "perfect" index (an index satisfying more than one query criteria) and ends up not using it. BTW, Centerfield tools recommend perfect indices on releases prior to V5R4 as well.
- 2) A common problem for novice System i DBAs is that they don't realize Query Optimizer is likely to opt for a table scan, rather than index implementation if selection criteria costing ends up resulting in an estimate of 30% of rows or more being returned. In fact, a table scan can be run in parallel and return your result set quicker.
- 3) The index you built has a different sort sequence from the job's sort sequence where the query is being run.

Hope this review helps you avoid some of the common issues preventing index implementation of your SQL query.

Good luck and have fun!



Elvis Budimlic
Development Director

wishing you and your families a very safe and happy holiday season....



David C. Ehlert
Lee R. Hasler
Mark L. Helm
Dean Lynn
J.P.
Leo
Brenda Schelella
Chris Bustrick

Effective performance tuning can only happen with two key skills – the ability to accurately determine why work is not finishing quickly enough and the ability to do something about it.

Accurately identifying the problem is obviously very important. Tuning an application, a database, or an SQL statement is a waste of time if the tuning efforts are not contributing in a significant way to the underlying problem. In today's complex, multi-tier environments it is getting more and more difficult to isolate the root cause of a performance problem. For example, even when the issue has been generally isolated to database access through SQL, it remains very difficult to target specific queries causing a slowdown. As a result, hundreds of man-hours can be wasted working on "problems" that don't lead to **measurable results**.

If the performance problem can be accurately isolated, the next challenge is to remove the bottleneck. In the case of in-house applications, the source code can be changed to improve performance. On the other hand, if you have purchased applications those changes may not be possible. In the case of SQL-tuning, you essentially have a third-party package that is determining how well the query runs – the software called the DB2 Query Optimizer. In the end, you may know what the problem is but have no control over the solution.

THE POWER OF THE LEVER

The ancient Greek mathematician Archimedes once said "Give me a lever long enough and I will move the world."

Wouldn't it be nice if we had more levers in our efforts to performance tune our database and the SQL used in our applications? The good news is that in the last several releases of the iSeries and IBM System i, the basic tools used to analyze database and SQL performance have gotten much better. The most widespread tool for SQL performance analysis is the database monitor (started using the STRDBMON command). In the past, the database monitor was like a 200 pound lever trying to move a 25 pound rock –

it was very heavy and collected too much data to make it useable in all but the most drastic situations.



More recently, however, options have been placed in the operating system to reduce the overhead of the database monitor thus making it realistic to use in more situations. Now there are ways to collect data for specific files or for individual SQL statements that are estimated to take longer than a certain amount of time. While not perfect, these advances are essential if the process of tuning is to be automated and made useable for people without deep knowledge of the DB2 optimizer. With these improvements, IBM has made it possible for any shop armed

with the proper tools to leverage the information collected by DB2 to do basic tuning. Furthermore, with the advent of V5R4, there are options to allow a database administrator to examine database access and SQL statements by looking at the optimizer's plan cache. This allows analysis to be done without having to collect database monitor data and for the administrator to have a 'rear view mirror' look at the efficiency of the SQL running on his system.

Secondly, IBM has also continued to provide methods to take action once a bottleneck has been uncovered. These methods vary from the obvious (building the right index) to the obscure (using the QAQQINI file). Along with basic performance improvements in the query optimizer, these techniques provide the needed knobs to remove bottlenecks when necessary to improve response time or reduce resource utilization.

SUMMARY

There are two trends working against each other in our pursuit of Archimedes' lever. The first is the increased complexity of our applications and the environments in which they run. The second trend, a positive one, is the constantly improving tools and techniques available to address performance bottlenecks. You may never have a lever long enough to move the world, but soon you will have one that will help you move your database access to the next level.

Bashful I/O

Sometimes I/O that occurs on an iSeries is hidden and difficult to find, as in the recovery ratio for journaled database files. When data is put into a journal receiver as a result of an insert, update or delete operation, the system looks to see if there would be a long recovery time if the system crashed. To shorten the potential recovery time, the files with the oldest entries on the journaled are forced to disk. This allows the journal component to have a newer “synch point” for that object. Obviously this means a lot of potential disk I/O.



The way to make these synch points occur less often is to increase the journal recovery ratio. A journal recovery ratio can be changed but is done differently depending on the release of the system. On V5R2 or earlier it is changed like this:

```
CALL QSYS/QJOCHRVC PARM(200000)
```

On releases V5R3 or higher an extra parameter is required:

```
CALL QSYS/QJOCHRVC PARM(X'00030D40' X'00000000')
```

The first input parameter on V5R3 and higher is an integer value passed in hexadecimal. In this example these two commands have the same effect.

Sleepy I/O

Another application mistake that can result in more disk I/O than necessary happens when a file is processed by a file that has been opened for input and output. When DB2 sees that there is a possibility the file will be updated, it does two things: (1) when the record is found it is locked; and (2) the database assumes that only that record is needed in memory. If, however, the file is being processed in arrival sequence because most of the rows need to be changed, the system will not do a good job of getting the file into memory before it is needed.

To solve this problem, an OVRDBF command is needed to tell DB2 that it is ok to bring a block of the file into memory instead of just the amount that holds a single record.

```
OVRDBF FILE(PAYROLL) TOFILE(PRODUCTION/  
PAYROLL) MBR(*FIRST) NBRRCDS(100)
```

Sneezy I/O

Sneezy was probably allergic to most I/O. One kind of I/O that is definitely something to sneeze at happens when files are extended to hold more rows. When extra space is allocated to hold more data, the system has to manage its bookkeeping to

keep track of where that space is and to which object it belongs. This requires that information be written to disk.

For example if you have applications that insert records into temporary files, this type of I/O is happening and slowing down those inserts. The way to avoid this is to pre-allocate the storage using the ALLOCATE parameter on the physical file. If you typically put 30,000 records into a work file, then that file could be changed, before the inserts start, *with the this command:*

```
CHGPF FILE(qtemp/workfile) SIZE(30000 1000 50)  
ALLOCATE(*YES)
```

When the first row is inserted into the file, DB2 automatically allocates enough room for 30,000 records. This eliminates the need to extend the file later and avoids the unnecessary disk traffic.

Happy I/O

Of all the dwarfs, Happy would probably be OK with just about any I/O. But even Happy might frown if an SQL statement created a large temporary table and used up a bunch of space in a busy memory pool. This can happen occasionally if join queries use a hashing algorithm to do the join when there are many join records.



The simplest way to avoid this problem is to create access paths (indexes) over the columns used to join one table to another. This allows the query optimizer to better estimate how many records in one table will join to records in another – and avoids using up all of your memory and causing all of the

other jobs to perform a lot of disk operations.

Doc I/O

Doc's job is to figure out why disk I/O is happening. On the iSeries there are multiple ways to diagnose disk issues.

The most commonly used is the WRKSYSSTS that allows you to see the paging activity in each of your memory pools and see how much page faulting and disk I/O is being done. From this you can go to the WRKACTJOB command and look at the open files of jobs running in a particular pool. When doing a DSPJOB (option 10) in WRKACTJOB, you can see the files that are open and how much each has been accessed. This can help identify which files are being accessed the most and possibly learn if there is an opportunity to improve the application.

For more difficult-to-solve problems, the most powerful tool to diagnose disk issues is Performance Explorer (PEX). You can see these tools by doing a GO CMDPEX command and reading IBM's documentation on how to collect data and print reports.

Hopefully you have a few new ideas on how to avoid disk I/O that may be happening on your iSeries. Good luck finding your own Dopey I/Os!

Anatomy of a joined UPDATE



Every time I answer a question about a searched or joined SQL UPDATE or DELETE, I think, "That's the last time, now EVERYBODY knows how to do it!"

And then I get the question again.

The problem is that the syntax of joined SQL UPDATE and DELETE is simply not intuitive and doesn't correlate directly to the syntax of the SELECT statement. Everybody uses a SELECT statement first to test what rows will be updated by their query and then they get a rude awakening when that same syntax doesn't work in their UPDATE statement.

Normally when you run a joined SELECT statement you do something like:

```
SELECT * FROM TableA a INNER JOIN TableB b  
ON a.key_field = b.key_field
```

It is then assumed that the UPDATE syntax *should* be similar:

```
UPDATE a SET a.update_field = b.update_field  
FROM TableA a INNER JOIN TableB b  
ON a.key_field = b.key_field
```

Wrong! This syntax, while intuitive, is not standard and although it may work in SQL Server's T-SQL language, it won't work with most other database platforms. Here is a rewrite that will work on all databases supporting SQL standards, DB2 included:

```
UPDATE TableA a SET a.update_field =  
(SELECT b.update_field FROM TableB b  
WHERE a.key_field = b.key_field)
```

Examine our WHERE clause and you'll notice that the join takes place, even though it's executed in a subquery and we're not using an explicit JOIN syntax.

One thing to remember with this joined UPDATE is that it assumes 1-1 correlation between keys in TableA and TableB. If there is a single instance of 1-N correlation, UPDATE will fail with the error **SQL0811** "Result of SELECT more than one row", or something to that effect. What this means is that DB2 found 2 or more matching rows in TableB for the key field in TableA and doesn't know which one to use to update the value in TableA, so it correctly aborts the operation.

One legitimate way to remedy this issue is to further enhance your join criteria by adding additional WHERE criteria, i.e. **AND a.key_field2 = b.key_field2**, in order to ensure uniqueness of your natural key. Alternatively, introduce a unique surrogate key in both tables i.e. via IDENTITY column attribute and like. Another, not so legitimate way, that may work to remedy this

issue is to use one of the aggregate SQL functions that return single values like MIN, MAX or AVG and simulate uniqueness that way.

Wait, we're not done yet. What if the 1-1 match is in place, but there's no guarantee that for every row in TableA there is a matching row in TableB?

In that case you'll get a **SQL0407** "Null values not allowed in column or variable UPDATE_FIELD" error. This is a perfectly legitimate error as the UPDATE statement so far is only performing a LEFT OUTER JOIN type of UPDATE, meaning that there is *no value* (the meaning of the keyword NULL) that matches the key value in TableA. The scary thing is that the UPDATE is only aborted if the TableA column that is being updated does not allow NULL values. If it did, it would simply be updated with NULL values, even though that is not what we intended here.

To remedy this issue, let's further enhance the UPDATE statement:

```
UPDATE TableA a SET a.update_field =  
(SELECT b.update_field FROM TableB b  
WHERE a.key_field = b.key_field)  
WHERE EXISTS  
(SELECT 1 FROM TableB b  
WHERE a.key_field = b.key_field)
```

What we're doing here is ensuring that only matching values in TableA are updated while non-matching values are left alone. You may be wondering about the purpose of projecting 1 in the second SELECT. It turns out that it doesn't really matter what is projected in the 2nd sub-select because it's only an existence check (EXISTS clause). It could be any column, '*' or 'Mickey Mouse'...it doesn't really matter.

An alternative method to address the NULL value issue is to use COALESCE or IFNULL functions and replace the target field with some default value. I dislike this solution though, as I believe in most scenarios you're better off leaving the existing value in there. However, if you have corrupted or invalid data in a target column and you want to update it unconditionally, this approach would work:

```
UPDATE TableA a SET a.update_field =  
COALESCE(  
SELECT b.update_field FROM TableB b  
WHERE a.key_field = b.key_field), '')
```

Similar rules apply to DELETE statements.

The issues I addressed in this article are common to all new users of the SQL UPDATE syntax. Hopefully this article eases your transition to SQL.