

# Reduce CPU Usage with 'SMART' SQL

By Susan Lawson and Daniel Luksetich, YL&A

## TABLE OF CONTENTS

Reduce CPU Usage with SMART SQL.....	3
DB2 Calls .....	3
Coding SMART SQL.....	3
DB2 Version 8 and 9 SQL Features.....	3
Making SQL Fast and Efficient.....	7
Closing Thoughts on SMART SQL.....	11
BMC SQL Performance for DB2 .....	12

## REDUCE CPU USAGE WITH SMART SQL

This paper discusses how and why to design SQL that will reduce the number of times calls are made to DB2. We'll look at how to:

- Write SMART (SQL Made to Accurately Reflect Transactions) applications rather than writing applications that act like generic I/O modules to DB2
- Properly tune SMART SQL to further reduce CPU and execute efficiently
- Exploit SQL and application features in DB2 on z/OS version 8 and 9 to help with these efforts.

## DB2 CALLS

Let's start by talking about the initial problem that causes most organizations to waste a great deal of CPU resources: unnecessary calls to DB2. This is often an innocent mistake, but over time it adds up to be one of the largest undetected performance problems. Why undetected? Simply put, the SQL appears to be 'perfect' and, therefore, no additional tuning efforts are made. The SQL appears to be perfect because each individual statement runs very fast and gets a good buffer pool hit ratio, and the access path looks fine. Common table access is limited to specific methods within the application, and can be called multiple times from various other methods. The actual SQL statements are simple single table access.

The problem with this is that the SQL is not doing anything but calling DB2 like a common I/O module, and not using advanced SQL syntax for performance. This is not a proper usage of the DB2 engine, and it results in the worst possible performance. The response to the high CPU consumption is often to add more resources and just assume that DB2 is rather inefficient.

## CODING SMART SQL

The real answer to this problem is to use the power of DB2 and not treat it like a data access layer. We want to see SMART SQL, not simple SQL. What do we mean by SMART? It's simple: SQL Made to Accurately Reflect Transactions. Yes, we made this up! But our intentions are clear. We need to have SQL that reflects the problems we are trying to solve, not just simply identify the data we would like to access. We need to develop SQL in a manner that not only makes sense for the business, but also for the use of intelligent relational database architecture.

There are many different ways to code a SQL statement to answer a given question. While the answer will be the same, the performance can vary dramatically depending upon the statement construct and data structure. Later in this paper we will look at a few different ways of coding SQL, and when is the proper way to code each technique. We will also look at how to effectively code joins, table expressions, and subqueries for the correct situation.

Before we discuss SQL coding, we will take a quick look at some of the features DB2 offers in the latest releases to help make our SQL more advanced and more efficient.

## DB2 VERSION 8 AND 9 SQL FEATURES

To make SMARTer SQL, we must be aware of all the features DB2 offers. We need to look forward into each release of DB2 and look for ways to most efficiently use DB2. DB2 Version 8 brought us such features as multi-row fetch, multi-row insert, insert within select, scalar fullselect, common table expressions, and recursion to help make our SQL smarter and reduce the number of calls made to the DB2 database. DB2 9 continues the trend with update/delete/merge within a select, and native SQL procedure language. By using these features we have been able to make significant reductions in CPU usage in several instances.

We will take a brief look at some of these features.

### INSERT WITHIN SELECT (V8)

The ability to put an insert within a select was one of the first features we used where we saw a significant reduction in CPU. In one case, we saw an almost 30% reduction because the volume of inserts and select was so high. By combining the statements, we saved CPU by reducing the trips to the database. This feature allows us to add data to the database without making unnecessary calls to DB2 when we view the newly added data and newly generated values such as sequence object values, identity column values, row IDs, defaulted values, etc. These SELECTed values can also be used to easily populate child tables. The example below shows this syntax. As you can see, it is a very easy statement to code. We encourage you to make changes in existing applications to use this and plan to use it in future applications. You can reduce CPU and elapsed time by combining the statements.

---

```
SELECT ACCT_ID INTO :acct-hv
FROM FINAL TABLE (INSERT INTO UID1.ACCOUNT (NAME, TYPE, BALANCE)
VALUES (:name-hv, :type-hv, :balance-hv ))
```

---

### UPDATE WITHIN SELECT (V9)

In DB2 9 we have UPDATE within a SELECT, which is the same concept as INSERT within a SELECT, but with UPDATES. This feature comes with the ability to add a column or variable to the statement to create derived data for retrieval. A single statement can be used to determine the values before or after records are updated (or deleted). Work files are used to materialize the intermediate result tables that contain the modified rows.

The following shows how an UPDATE/SELECT would work prior to DB2 9. We are selecting account balance information and the time it was updated.

---

```
SELECT ACCT_BAL, UPD_TSP
FROM ACCOUNT
WHERE ACCT_ID = :ACCT-ID
```

---

In this next statement we are updating our account to add a payment to our existing balance based on the account ID and the timestamp. There may be an issue with concurrency here, and the update may not even work because another update could have happened after the select was performed.

---

```
UPDATE ACCOUNT SET PAY_AMT
= :PAY-AMT, ACCT_BAL = :ACCT-BAL
, PAY_DATE = CURRENT DATE
, UPD_TSP = CURRENT TIMESTAMP
WHERE ACCT_ID = :ACCT-ID
AND UPD_TSP = :UPD-TSP
```

---

In DB2 9 with the ability to put the UPDATE within the SELECT, we know we are going to update the correct account with the desired timestamp, that no other activity could get in the middle, and that we will return the new balance. It is all done with one statement.

---

```
SELECT ACCT_BAL INTO :ACCT-BAL
FROM FINAL TABLE UPDATE ACCOUNT
SET PAY_AMT = :PAY-AMT
, ACCT_BAL = ACCT_BAL + :PAY-AMT
, PAY_DATE = CURRENT DATE
, UPD_TSP= CURRENT TIMESTAMP
WHERE ACCT_ID = :ACCT-ID
```

---

An example of using INCLUDE in the update is shown below. This powerful construct gives us a lot of flexibility when designing our applications and also gives us the ability to greatly reduce calls to DB2. In this example, we are creating a column to hold our old balance before we do the update. After we do the update, we can select the old balance and the new balance. Prior to DB2 9 this would have taken several statements.

---

```
SELECT ACCT_BAL, OLD_BAL
INTO :NEW-ACCT-BAL, :ACCT-OLD-BAL
FROM FINAL TABLE
UPDATE ACCOUNT INCLUDE (OLD_BAL DEC(9,2))
SET PAY_AMT = :PAY-AMT
, ACCT_BAL = ACCT_BAL + :PAY-AMT
, PAY_DATE = CURRENT DATE
, UPD_TSP= CURRENT TIMESTAMP
```

---

```
,OLD_BAL = ACCT_BAL
WHERE ACCT_ID = :ACCT-ID
```

---

### DELETE WITHIN SELECT (V9)

Another new feature following this theme includes the ability to do a DELETE within a SELECT. In the following example we are deleting our account and would like to report on the balance during this process.

```
SELECT ACCT_BAL, PAY_DATE
  INTO :ACCT-BAL, :PAY-DATE
FROM OLD TABLE
DELETE FROM ACCOUNT
WHERE ACCT_ID = :ACCT-ID
```

---

### MERGE AND SELECT FROM MERGE (V9)

The MERGE statement can reduce the number of calls for applications that synchronize tables. Instead of having the application perform an update and then on failure (not found) do an insert, you can use the MERGE to basically do a conditional update, or if you like to call it so, an 'upsert'. The merge will do an update when the data is matched and will do an insert when it is not. This can dramatically reduce traffic for this type of process.

```
MERGE INTO MAIN ACCOUNT AS M
USING VALUES (:hv_acnum, :hv_amnt)
FOR 4 ROWS AS B(ACCT_NUM, AMOUNT)
ON M.ACCT_NUM = B.ACCT_NUM
WHEN MATCHED THEN
UPDATE SET BALANCE = M.BALANCE + B.AMOUNT
WHEN NOT MATCHED THEN
INSERT (ACCT_NUM, BALANCE) VALUES (B.ACCT_NUM, B.AMOUNT)
NOT ATOMIC CONTINUE ON SQLEXCEPTION
```

---

We can also select from the results of the merge. In the following statement we include a column called DIFF to show the difference the values of the stocks before and after the update, or the value set during the insert if the stock did not exist in our table.

```
SELECT STK_ID, VALUE, DIFF
FROM FINAL_TABLE
(MERGE INTO STOCK_TABLE AS S INCLUDE (DIFF, DECIMAL(6,2))
USING (:hv_stk_id, :hv_value) for :hv_nrows ROWS AS R (STK_ID, VALUE)
ON S.STK_ID = R.STK_ID
WHEN MATCHED THEN UPDATE SET DIFF = R.VALUE - S.VALUE,
                               VALUE = R.VALUE
WHEN NOT MATCHED THEN INSERT(STK_ID, VALUE, DIFF)
VALUES (R.STK_ID, R.VALUE, R.VALUE)
NOT ATOMIC CONTINUE ON SQLEXCEPTION);
```

---

### SCALAR FULLSELECT (V8)

The ability to code a scalar fullselect as an expression was introduced in DB2 z/OS version 8. This allows us to write smarter, more advanced SQL because we can put a select anywhere. We can have a select within a select, a select within a where clause, or a select within a case statement. This opens up several possibilities and options when coding SQL. We have also been able to use it to improve transaction performance.

Below is a fun example using scalar fullselect to answer three completely different questions. In the past this probably would have taken three different statements and subsequently three trips to the database. In this example we have a database called DRINKS, where we store all the ingredients for drinks we can make. Our query is going to provide a list of all ingredients for a martini, and include the number of drinks in the drink database table and the number of ingredients for the drink being processed- all in one statement.

```
SELECT DRINK, INGREDIENT,
       (SELECT COUNT(DISTINCT DRINK)
        FROM DRINKS),
       (SELECT COUNT(*)
        FROM DRINKS B
```

```

        WHERE A.DRINK = B.DRINK)
FROM      DRINKS A
WHERE     DRINK = 'MARTINI';

```

---

### COMMON TABLE EXPRESSIONS AND RECURSION (V8)

Common table expressions allow us to create a sort of inline table that is built upon first reference and then it can be referenced throughout the query. In some cases, this helps reduce the passes we have to make through the data as well as possibly reducing calls to the database. One of the most useful features of a common table expression is the ability to code recursive SQL. We now have the ability to support problems such as bill of materials, navigating networks, traversing organizational charts, or solving problems such as the ability to generate data.

Below is an example of using a common table expression in a recursive manner (the common table expression references itself). This statement simply counts to 10. In the past this would have required several temporary tables and a series of inserts/selects/deletes. Here we have a common table expression called TEMPCNT where we start with a value of 1 and then add to it until we reach 10. Then we will go back and select those values. This is all performed in one statement without the need for temporary tables or several calls to DB2. We are showing a very basic example here, but the use of recursion can be very complex and very powerful. We have seen process run times go from days to hours by using this feature.

```

WITH TEMPCNT(CNT) AS
(SELECT 1
FROM SYSIBM.SYSDUMMY1
UNION ALL
SELECT CNT+1
FROM TEMPCNT
WHERE CNT < 10)
SELECT CNT
FROM TEMPCNT

```

---

### MULTI-ROW FETCH AND INSERT (V8)

Multi-row fetch and insert allow us to fetch or insert up to 32,767 rows at a time in a single call. This can significantly reduce CPU cost. We often found that fetching around 50-100 rows at a time provided our biggest savings. In one case for a sequential reader, we saw around 60% savings. To use multi-row fetch, we do have to make a few changes to our local application code to include the ability to fetch into an array of host variables. But the effort to make this change is worth it when it comes to saving CPU. Below is a simple example of the fetch into a row set.

```

FETCH NEXT ROWSET FROM CUR1
FOR :hv ROWS INTO :values1, :values2;

```

---

For remote applications, multi-row fetch is automatically enabled for applications taking advantage of block fetching.

### NATIVE SQL PROCEDURES (V9)

Stored procedures have always been a way to reduce calls to the DB2 database (if used properly) because we could encapsulate multiple statements into a stored procedure, making only one call to DB2 for execution and then returning the results. The SQL Procedure language had provided an extension to the SQL language in the past, and was executed as a stored procedure as an external process (a compiled C program). However, the overhead for execution was high due to the cost to call the stored procedure and have it execute in a WLM address space. In DB2 9 the SQL Procedure language does not need a C compiler, or a stored procedure address space, can run native in the DB2 engine with no additional calls (overhead), and can be zIIP enabled for remote applications.

## MAKING SQL FAST AND EFFICIENT

So far we have discussed the importance of making our SQL SMARTer and reducing the number of times we call DB2, but that is only solving half of the problem. Once we start taking advantage of the power of the SQL language, we must understand how to code statements efficiently. The flexibility and power of the language allows us to code the same logical SQL statement many different ways. The key is to try different options and find which is best for your particular problem.

You can code an SQL statement many different ways to answer a given question. While the answer will be the same, the performance can vary dramatically depending upon the statement construct and data structure. Here we will look at the different major ways of coding SQL and when to use each technique. How can we code joins, table expressions, and subqueries the correct way for the correct situation?

The contributing factors to making an SQL performance choice include the size of the table, necessary joins, and the predicates. Some of the coding syntax we need to use includes correlated and non-correlated subqueries, nested table expressions, and correlated nested table expressions. We must also decide when to use correlation versus non-correlation when attempting to force an access path. There are also questions regarding merge versus materialization, where the tradeoff is CPU versus I/O. We will address these topics, but these are just a few in a long list of things to consider when make decisions that influence SQL performance.

Before attempting any SQL tuning project, or when constructing new SQL statements for a new application, you must understand the potential performance implications of how you are going to code the statements. To do this, you need an understanding of the tables you are accessing and the process you are attempting to serve. How is what you want to accomplish affected by the size of the tables you access? Consider the indexes available and the impact of adding indexes if it appears necessary, because sometimes the cost of adding the index outweighs the benefit from the SQL. Consider filtering predicates and try to always filter as much as possible, as early as possible.

### IMPORTANCE OF STATISTICS

Before starting to tune SQL you must make sure that the catalog statistics accurately reflect the data stored in the tables. The vast majority of access path problems can be resolved by collecting catalog statistics, or if your data is disorganized, by running a REORG and then collecting catalog statistics. It is very important that table and index statistics are in agreement. Therefore, we strongly advise that you never manually manipulate the catalog statistics or collect index statistics at a separate time from the table statistics. It can be a wasted effort to tune SQL if the statistics are inaccurate. The DB2 optimizer will make the best decision based on the available statistics; if the statistics are not reflective of the data, it is unreasonable to assume the optimizer will make the best decision possible.

To collect catalog statistics, either run RUNSTATS once or all the time. If you are familiar with the data stored in the tables for an application and you are familiar with how the application needs to access that data, then you should be familiar with the proper access paths for the queries. In this case, collect statistics that accurately represent the proper data storage and assure that DB2 is picking the appropriate access path. Once this is achieved, the statistics should not be altered so that your access paths remain consistent. On the other hand, if you do not understand the data organization and how the application uses the data, you are better off putting your faith purely in DB2. In this situation, run RUNSTATS regularly to give DB2 the most up-to-date information to pick what it thinks is the best access path.

The collection of RUNSTATS can be expensive, especially for very large objects. Setting a low sample percentage helps, but you shouldn't set the sample below 10% because you may lose accuracy depending on the utility product you are using.

### KNOWING THE APPLICATION AND THE DATABASE

In addition to having accurate statistics, and knowing what those statistics are, you need a complete understanding of the objects being accessed by the application and how that application is accessing the objects. Talk to application developers to see what their intention was when they coded the SQL statements. Talk to the data administrators to understand the concepts behind the database design.

If you are involved with the database or application design, understand the importance of reducing the number of indexes. Try to keep the number of indexes to a minimum when designing. This can help performance

tremendously and help keep the SQL statements simple. Avoid system-generated keys and stick with natural keys when possible. Do not partition unless there is a reason; size, concurrency, or availability.

### CODING PREDICATES

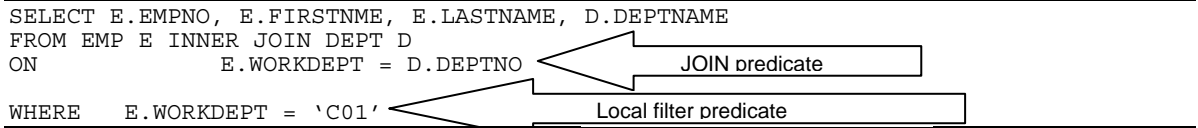
Proper coding of predicates is important to the performance of complex SQL statements. Make sure that you use explicit join syntax when coding joins. Use table specifications if nesting table expressions, and avoid using the same correlation name for a table more than once.

It is important to understand that the predicates in a WHERE clause dictate the filtering for a query. The predicates coded in an ON clause dictate how DB2 performs the join. DB2 can understand when a filtering predicate is placed inside an ON clause, but is not perfect. Coding filtering predicates in the ON clause of outer joins can lead to automated join simplification and unexpected results.

```

SELECT E.EMPNO, E.FIRSTNME, E.LASTNAME, D.DEPTNAME
FROM EMP E INNER JOIN DEPT D
ON          E.WORKDEPT = D.DEPTNO
WHERE      E.WORKDEPT = 'C01'

```



Code your filtering predicates in an order that you think filters the most data first. DB2 does apply predicates in a prescribed order, but for predicates evaluated at the same time they are processed in the sequence coded. The same rule should be applied for subqueries.

### CODING PROPER JOINS

With programming trends centering on object- and service-oriented designs, we are seeing more and more fragmented, or single table, database access. It is important for performance to code joins when you require data from more than one table. The most expensive thing you can do is to call the database at all, especially over a network. To reduce the number of calls, code multi-table access in your statements.

We have found in our testing of a local application that coding a two-table join versus accessing the tables individually saved over 30% CPU. Yes, it does mean coding control breaks in your application, but typically application server CPU costs are lower than the mainframe CPU costs, and coding control breaks really isn't that difficult.

When coding joins, make sure that DB2 is selecting the table that accesses the fewest number of rows first in the join sequence. In the vast majority of cases, DB2 will pick the correct join sequence (provided that the statistics are accurate). However, in some cases we may have to convince it to pick the better join sequence using one of the following techniques:

- Changing an inner join to an outer join
- Forcing the materialization of one of the tables with a DISTINCT or GROUP BY in a nested table expression
- Converting joins to subqueries
- Coding an ORDER BY on the columns of the index of the table to be accessed first
- Ordering the tables in the FROM clause (depending on complexity of query)
- Coding non-transitive closure predicates (i.e. IN, LIKE, subquery) on the table to be accessed first
- Adding a CAST function to join predicate

### SUBQUERY PERFORMANCE

Subqueries are fullselects that are evaluated as part of a predicate. The two major forms of subqueries are correlated and non-correlated. Typically, we refer to the subquery portion of the entire statement as the subquery, and the portion of the statement that references the subquery in a predicate is called the outer query.

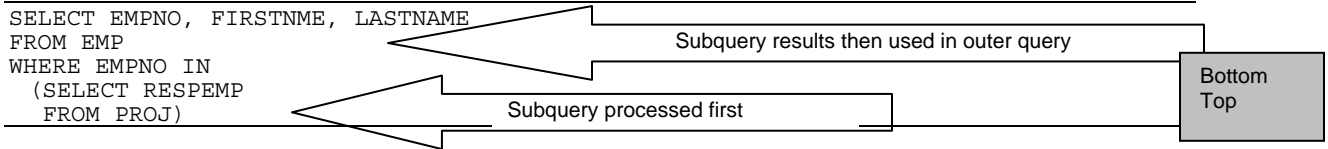
Non-correlated subqueries are processed only once before the outer query is processed. The results of the subquery are used as part of predicate evaluation, and DB2 can use an index for the column or row expression



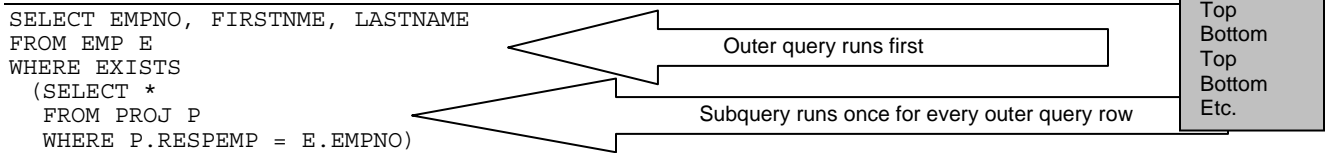
of the outer query. If an index is used in support of the predicate that contains the subquery then the predicate is processed in stage 1 and is index matching. If there is no supporting index or a different index is chosen, then the predicate is stage 2.

Correlated subqueries can be processed multiple times, and after the outer query processing has begun. For the outer query, that means the predicate containing the correlated subquery is always a stage 2 predicate because the correlated values from the outer query are needed to run the subquery. This also means that for every row processed by the outer query, the correlated subquery is called; therefore, the subquery can be called many times, maybe thousands or more.

**Non-Correlated Subquery**



**Correlated Subquery**



Any subquery situation can be coded one of three ways; correlated, non-correlated, or as a join. Which one performs better really depends upon:

- Table sizes
- Query and subquery result quantity
- Available indexes

Knowing table sizes, result sizes, and the indexes are critical to coding the statement properly for performance. Try coding them each way and test them two ways: EXPLAIN, and running a benchmark. Benchmarking can be done via REXX routines to test query performance, or sometimes even SPUFI or DSNTEP2.

Here are some guidelines to help you choose which method is better:

A non-correlated subquery is better if

- The subquery result is relatively small compared to the outer query
- The subquery result is not excessively large
- The outer query utilizes matching index access for the predicate containing the subquery
- There is no supporting index for the subquery

A correlated subquery is best when

- The subquery result is potentially large compared to the outer query
- The outer query has no supporting index for the predicate containing the subquery
- There is a good support index for the correlated predicate in the subquery

A join is usually best when

- There is good matching index potential for outer query and subquery
- The subquery does not introduce duplicate rows

Both non-correlated and correlated subqueries can be transformed into a join by DB2. Beginning with DB2 9, DB2 can actually correlated or de-correlate subqueries automatically.

## MERGE VS MATERIALIZATION

DB2 has the ability to merge table expressions into the outer portion of complex SQL statements. It does this during query transformation (query rewrite). The fullselect inside the table expression is combined with the outer portion of the statement that makes reference to the table expression.

---

```
SELECT A.*
FROM
(SELECT DEPTNO, DEPTNAME, MGRNO
 FROM DEPT
 WHERE DEPTNO = 'A00') AS A
```

---

DB2 merges the table expression above into the following:

---

```
SELECT DEPTNO, DEPTNAME, MGRNO
FROM DEPT
WHERE DEPTNO = 'A00'
```

---

If DB2 does not merge the table expression into the outer portion of the statement, then it would have to materialize the table expression. In that situation, the table expression would run first, its results would be placed in a work file, and the work file would be accessed by the outer portion of the statement.

So which is better? In general, merge is better than materialization because the extra work file processing is eliminated. Most of the time this is true. However, when built-in DB2 functions, complex expressions (such as CASE), and user-defined functions become involved, the performance of a merged table expression may degrade. This is because the functions that are coded in the table expression and referenced in the outer portion of the statement are merged with those outer references. If the reference occurs only once, there is no difference in performance. However, if there are multiple references, the query rewrite process will code the function multiple times, and the function will be executed multiple times - resulting in an increase in CPU usage. This becomes more exaggerated the more times the result of a function in a table expression that is merged is referenced.

When there is repeated execution of merged functions and expressions causing excessive CPU usage, you could force materialization of the nested table expression. For an in-line nested table expression (referenced directly after a FROM as opposed to after a JOIN), adding a non-deterministic function, such as RAND(), will force materialization.

---

```
SELECT
MAX(A.PART) AS MAX, MIN(A.PART) AS MIN, AVG(A.PART) AS AVG
FROM
(SELECT LENGTH(RTRIM(LASTNAME)) AS PART, RAND()
 FROM EMP) AS A
```

---

You can choose between materialization and merge; however, be very careful and benchmark the difference. For five levels of nesting, we have seen 90% CPU savings when forcing materialization of a nested table expression with many functions utilized and re-referenced. When materialization actually occurs depends of version of DB2 (and maintenance level).

Be very aware of the implications of merged table expressions, especially for queries that process large volumes of data with very complex SQL statements. These queries do not typically appear as problem queries because their EXPLAINed access path looks simple, and their execution times are good because they do not do excessive I/O. However, their CPU consumption can be high. Is the CPU usage of your complex merged statement close to the total execution time? You may want to examine the statement for the potential repeated executions of embedded merged functions and expressions.

## TABLE EXPRESSIONS

Nested table expressions can be coded as correlated or non-correlated, and the processing can be similar in some respects to correlated or non-correlated subqueries. Like a correlated subquery, a correlated table expression can execute many times during the statement execution.

In this example, we are retrieving some information about employees that have the job of sales representative in our sample database, as well as aggregated information about the departments that they work in.

### Non-Correlated Table Expression

```

SELECT TAB1.EMPNO, TAB1.SALARY, TAB2.EMPCOUNT
FROM EMP TAB1
LEFT OUTER JOIN
  (SELECT COUNT(*) AS EMPCOUNT, WORKDEPT
   FROM EMP
   GROUP BY WORKDEPT) AS TAB2
ON TAB1.WORKDEPT = TAB2.WORKDEPT
WHERE TAB1.JOB= 'SALESREP'
    
```

### Correlated Table Expression (a.k.a. Sideways Reference)

```

SELECT TAB1.EMPNO, TAB1.SALARY, TAB2.EMPCOUNT
FROM EMP TAB1
LEFT OUTER JOIN TABLE
  ( SELECT COUNT(*) AS EMPCOUNT
    FROM EMP
    WHERE WORKDEPT = TAB1.WORKDEPT) AS TAB2
ON 1=1
WHERE TAB1.JOB= 'SALESREP'
    
```

Like with the subquery examples, the non-correlated table expression will run once and the results will be joined to the outer table in the query. The correlated table expression may run multiple times, depending upon how many outer table rows qualify.

The access path for the correlated table expression is always nested loop, and it is likely that the correlated predicate is used for index access inside the table expression. For the non-correlated table expression, it is less likely that the join predicate will be used for index access inside the table expression, especially if there is aggregation inside the table expression that forces materialization. A correlated nested table expression is also referred to as a sideways reference.

We can write this join two different ways: correlated and non-correlated. Which is better?

A non-correlated join is better when

- All data from the table expression will be used in the result
- Large quantities of rows qualify for outer table access
- There is no supporting index for the join column
- The table expression is sorting

A correlated join is better when

- The join predicate is not pushed into the equivalent non-correlated table expression
- A relatively small quantity of rows qualify for outer table access and if a small percentage of table expression rows are processed

Be very careful with correlated table expression because it executes once for every outer row accessed, the results are placed in a workfile, and it must have a supporting index for inner table access. Always benchmark these queries!

## CLOSING THOUGHTS ON SMART SQL

The bottom line here is that we need to make our SQL work for us. Make it SMARTer and make it efficient. There are many ways to write SQL statements and we have to find the best way for our application and our business. New features continually come out in DB2 to help with our efforts. And as we write these more complex statements, it is important to monitor, explain, and benchmark them to ensure we are getting the best performance. Tools are often a very integral part of this effort, and we need to find the right tool for the job from our vendors.

When you write SQL, you have control over how efficient you can make it. But chances are that you have inherited SQL. Some may have come with ERP packages, and some has probably been in your shop for many years. You often need tools to make this SQL more efficient. One tool that can help is BMC SQL Performance for DB2.

## BMC SQL PERFORMANCE FOR DB2

BMC SQL Performance for DB2 enables you to pinpoint poorly performing SQL quickly and easily, so that you can reduce costs and improve response times. It provides tools to manage SQL performance across the application life cycle. In development, it validates SQL to ensure optimal performance characteristics like valid access paths and best-practice SQL coding techniques. In production, it captures metrics to identify and resolve performance problems. BMC SQL Performance for DB2 recommends optimal index usage, enables workload comparisons, and uses historical data to identify performance trends.

BMC SQL Performance for DB2 consists of BMC APPTUNE for DB2, BMC SQL Explorer for DB2, and Performance Advisor technologies, including Index Analysis, Workload Access Path Compare, and Reorg Advisor.

### BMC SQL Performance for DB2

- Highlights the most expensive SQL statements and makes recommendations to improve performance, resulting in faster problem resolution and more efficient processing
- Enables you to resolve SQL performance problems early in the application life cycle, improving performance and mitigating risk
- Compares workload access paths to identify changes that occur when SQL is migrated across DB2 environments, enabling you to correct problems before they affect performance and availability
- Analyzes index usage and models changes to indexing strategy, reducing costs and improving response times
- Identifies and helps resolve problems with table spaces and index spaces, improving performance
- Gathers and manages historical performance data for trending and analysis
- Identifies which DB2 objects need reorganization, evaluating both space usage and performance metrics, thus reducing reorganizations

### BMC SQL Performance for DB2 exclusive features include

- Integrated Common Explain - models changes in SQL statements and index design
- Reorg Advisor - uses SQL performance metrics and space usage to ensure accurate selection of reorganization candidates
- Workload Access Path Compare - predicts performance impacts as SQL is migrated across DB2 environments.
- Index Analysis component - identifies indexes that are being poorly utilized and SQL statements that are not effectively using DB2 indexes

For more information, visit [www.bmc.com/sqlperformance](http://www.bmc.com/sqlperformance).

## ABOUT THE AUTHORS

*Susan Lawson and Dan Luksetich are internationally recognized DB2 consultants. They specialize in DB2 performance and availability. They have both authored several articles and books on DB2. For more information or to contact them, visit [www.db2expert.com](http://www.db2expert.com).*

## **Business runs on IT. IT runs on BMC Software.**

Business thrives when IT runs smarter, faster, and stronger. That's why the most demanding IT organizations in the world rely on BMC Software across both distributed and mainframe environments. Recognized as the leader in Business Service Management, BMC offers a comprehensive approach and unified platform that helps IT organizations cut cost, reduce risk, and drive business profit. For the four fiscal quarters ended June 30, 2009, BMC revenue was approximately \$1.88 billion. Visit [www.bmc.com](http://www.bmc.com) for more information.

