Craig S. Mullins

December 1995

As originally published in *Database Programming & Design* magazine:

The Physical Side of Business Rules and the Procedural DBA

by Craig S. Mullins

Until recently, the domain of a database management system was, appropriately enough, to store, manage, and access data. Although these core capabilities are still required of modern DBMS products, additionally procedural functionality is slowly becoming not just a nice feature to have, but a necessity. The ability to define business rules to the DBMS instead of in a separate, application program build upon the concept of data sharing. However, instead of merely sharing data, modern DBMSes will enable applications to share both common data elements and code elements.

Specifically, all of the most popular RDBMS products are adding more and more complex features and components to facilitate procedural logic. This occurrence requires the expansion of the way that organizations have traditionally handled database management and administration. Typically, as new features are added, the administrative, design, and management of these features is assigned to the database administrator (DBA) by default. This is not always the best approach.

This article will discuss the different physical implementations of business rule support and their impact on the role of the DBA.

The Classic Role of the DBA

Just about every database language programmer has their favorite curmudgeon DBA story. You know, those famous anecdotes that begin with "I have a problem..." and end with "...and then he told me to stop bothering him and read the manual." DBAs simply do not have a "warm and fuzzy" image. This probably has more to do with the nature and scope

of the job than anything else. The DBMS spans the enterprise, effectively placing the DBA on call for the applications of the entire organization.

To make matters worse, the role of the DBA has expanded over the years. In the pre-relational days, both database design and data access was complex. Programmers were required to explicitly code program logic to navigate through the database and access data. Typically, the pre-relational DBA was assigned the task of designing the hierarchic or network database design. This process usually consisted of both logical and physical database design, although it was not always recognized as such at the time.

Once the database was planned, designed, and generated, and the DBA created backup and recovery jobs, little more than space management and reorganizations were required. This is not to belittle these tasks. The pre-relational DBMS products such as IMS and IDMS required a complex series of utility programs to be run in order to perform backup, recovery, and reorganization. This consumed a large amount of time, energy, and effort.

As RDBMS products gained popularity, the role of the DBA expanded. Of course, DBAs still designed databases, but increasingly these were generated from logical data models created by data administration staffs. The up-front effort in designing the physical database was reduced, but not eliminated. Relational design still required physical implementation decisions such as indexing, denormalization, and partitioning schemes. But, instead of merely concerning themselves with physical implementation and administration issues, DBAs found that they were becoming more intimately involved with procedural data access.

The nature of the RDBMS requires additional involvement during the design of data access routines. No longer were programmers navigating through the data; the RDBMS was. Optimizer technology embedded into the RDBMS was responsible for creating the access paths to the data. The optimization choices had to be reviewed by the DBA. Program and SQL design reviews became a vital component of the DBA's job. Furthermore, the DBA took on additional monitoring and tuning responsibilities. Backup, recover, and REORG were just a start. Now, DBAs used EXPLAIN, performance monitors, and SQL analysis tools to proactively administer RDBMS applications.

Oftentimes, DBAs were not adequately trained in these areas. It is a distinctly different skill to program than it is to create well-designed relational databases. DBAs, more often than not, found that they had to be able to understand application logic and programming techniques to succeed.

The Trend of Storing Process With Data

RDBMS products are maturing and gaining more functionality. The clear trend is that more and more procedural logic is being stored in the database. One of the most common forms of logic stored in the database is the exit routine. An exit

routine, such as an EDITPROC or VALIDPROC in DB2 for MVS, is usually coded in Assembler language (sometimes a 3GL like COBOL is permitted). This code is then attached to a specific database object and is executed at a specified time, such as when data is inserted or modified.

Exit routines have been available in DBMS products for many years, and are typically the responsibility of the DBA to code and maintain. But exit routines are merely the tip of the iceberg. The most popular and robust RDBMSes also support additional forms of database-administered procedural logic known as stored procedures, triggers, constraints, assertions, and user-defined functions (UDFs).

Stored procedures are procedural logic that is maintained, administered, and executed through the RDBMS. The primary reason for using stored procedures is to move application code off of a client workstation and on to the database server. This typically results in less overhead because one client can invoke a stored procedure and cause the procedure to invoke multiple SQL statements. This is preferable to the client executing multiple SQL statements directly because it minimizes network traffic which can enhance overall application performance. A stored procedure is not "physically" associated with any other object in the database. It can access and/or modify data in one or more tables. Basically, stored procedures can be thought of as "programs" that "live" in the RDBMS.

Triggers are event-driven specialized procedures that are stored in, and executed by, the RDBMS. Each trigger is attached to a single, specified table. Triggers can be thought of as an advanced form of "rule" or "constraint" written using procedural logic. A trigger can not be directly called or executed; it is automatically executed (or "fired") by the RDBMS as the result of an action—usually a data modification to the associated table. Once a trigger is created it is always executed when its "firing" event occurs (update, insert, delete, time, etc.). Refer to Figure 1 for a depiction of the difference between stored procedures and triggers.

A **constraint** is a database-enforced limitation or requirement, coded into the definition of a table and that is non-bypassable. Most experienced RDBMS users are familiar with unique constraints and referential constraints. A unique constraint forbids duplicate values to be stored in a column or group of columns. Referential constraints define primary and foreign keys within a two tables that define the permitted, specific data values that can be stored in those tables. Although both of these are forms of constraints, they are also pre-defined to the DBMS and can not be changed. They do, however, require quite a bit of administration and management. A newer type of constraint, known as a **check constraint**, is gaining acceptance in RDBMS products. Check constraints are used to defined the exact requirements for values that can be stored in a specific table. A wide range of rule can be defined using check constraints because they are defined using the same search conditions used in SQL WHERE clauses. Some sample check constraints follow:

    CHECK (REGISTERED IN ("T", "F"))

    CHECK (MONTH BETWEEN 1 AND 12)

```
CHECK (SALARY < 75000)
```

An **assertion** is basically a free-standing check constraint. Whereas check constraints are explicitly defined within the DDL of a single table, an assertion is defined outside the scope of any table. This limits the flexibility of a check constraint. Assertions are usually created to enforce restrictions that span more than one table. An assertion, once defined, operates basically the same as a check constraint. The following is an example of an assertion:

```
CREATE ASSERTION vehicles_in_stock
    CHECK   (    (SELECT COUNT(*) FROM TRUCK_TABLE)  +
                 (SELECT COUNT(*) FROM CAR_TABLE)
                 < 1500
            )
```

This assertion will enforce the business rule that a total of no more than 1500 trucks and cars can be kept in stock at any one time.

A **UDF**, or **user-defined function**, provides a result based upon a set of input values. UDFs are programs that can be executed in place of standard, built-in SQL scalar or column functions. A scalar function transforms data for each row of a result set; a column function evaluates each value for a particular column in each row of the results set and returns a single value. Once written, and defined to the RDBMS, a UDF becomes available just like any others built-in function.

Stored procedures, triggers, constraints, assertions, and UDFs are just like other database objects such as tables, views, and indexes, in that they are controlled by the DBMS. These objects are often collectively referred to as **server code objects, or SCOs,** because they are actually program code that is stored and maintained by a database server as a database object. Depending upon the particular RDBMS implementation, these object may or may not "physically" reside in the RDBMS. They are, however, always registered to, and maintained in conjunction with, the RDBMS.

Intelligent Agent Technology

Though not strictly a server code object, many products are incorporating **intelligent agents.** This technology provides interoperable, compatible programs that operate in a fault-tolerant, secure memory space to automatically perform a specific task or tasks. Agents are sometimes referred to as good viruses. An example of a primitive agent is a word processor that automatically corrects misspellings as you type (without requiring a "spell check" to be explicitly requested). None of the currently popular RDBMS products support intelligent agent technology, but many supporting products such as performance monitors do. Look for the incorporation of intelligent agents into RDBMS products soon.

Why Are Server Code Objects So Popular?

The predominant reason for using SCOs is to promote code reusability. Instead of replicating code on multiple servers or within multiple application programs, SCOs enable code to reside in a single place: the database server. SCOs can be automatically executed based upon context and activity or can be called from multiple client programs as required. This is preferable to cannibalizing sections of program code for each new application that must be developed. SCOs enable logic to be invoked from multiple processes instead of being re-coded into each new process every time the code is required.

An additional benefit of SCOs is increased consistency. If every user and every database activity (with the same requirements) is assured of using the SCO instead of multiple, replicated code segments, then the organization can be assured that everyone is running the same, consistent code. If each individual user used his or her own individual and separate code, no assurance could be given that the same business logic was being used by everyone. In fact, it is almost a certainty that inconsistencies will occur.

Additionally, SCOs are useful for reducing the overall code maintenance effort. Because SCOs exist in a single place (the RDBMS), changes can be made quickly without requiring propagation of the change to multiple workstations.

Another common reason to employ SCOs to enhance performance. A stored procedure, for example, may result in enhanced performance because it is typically stored in parsed (or compiled) format thereby eliminating parser overhead. Additionally, in a client/server environment, stored procedures will reduce network traffic because multiple SQL statements can be invoked with a single execution of a procedure instead of sending multiple requests across the communication lines. Refer to Figures 2 and 3.

Finally, SCOs can be coded to support database integrity constraints, implement security requirements, reduce code maintenance efforts, and support remote data access.

Database Server Support for Server Code Objects

All of the most popular RDBMS products provide some level of support for SCOs. Of course, the manner in which they are supported differs from product to product because there no universal standard for SCO implementation. Consult the grid in Table 1 for a listing of which RDBMS supports which server code objects.

It is important to understand the differences in the way that these RDBMSes support SCOs. Organizations that must support and administer more than one RDBMS products will need to comprehend the differences, some of them subtle, and implement accordingly without confusing features and functionality from product to product. Additionally, organizations that are evaluating which RDBMS to implement should delve into the product implementation and usage

details to uncover the subtleties of each product. Rarely is a simple checklist of features sufficient for decision-making purposes.

Table 1. Server Code Object Support in the Major RDBMS Products.

DBMS	Triggers	Stored Procedures	Check Constraints	Assertions	UDFs
Oracle7	YES	YES	YES	NO	YES^1
Sybase System 10	YES	YES^2	YES	PARTIAL^3	NO^2
INFORMIX V6	YES	YES^2	YES	NO	NO^2
DB2 for MVS V4	NO^4	YES	YES	NO	NO
DB2 for MVS V3	NO^4	NO	NO	NO	NO
DB2 for Common Servers V2	YES	YES	YES	NO	YES

1   Oracle7 supports the creation of functions, which are actually procedures that return a value. They differ from UDFs, which **must** be used in the context of a SQL statement.

2   Sybase and INFORMIX stored procedures can return values (like Oracle functions). But they can not be used in the context of a SELECT statement.

3   Sybase assertions are called rules. Although created as free-standing objects, rules must be bound to particular tables and columns before they are enforced.

4   DB2 for MVS supports the VALIDPROC, EDITPROC, and FIELDPROC exit routines that are similar to triggers, but do not provide the same functionality.

For example, if it is important that the RDBMS you are considering support stored procedures, there would appear to be little difference between the major players after examining Table 1. All of the major players support stored procedures. However, each product provides stored procedure support in entirely different ways and with differing functionality. DB2

stored procedures are written in a traditional programming language and then registered to the DBMS. Oracle, Sybase, and Informix, however, have extended versions of SQL that are used for stored procedures. And each of these extensions provides different syntax and different functionality.

To further clarify this point, consider the support for triggers in the RDBMS products that provide trigger support. Each uses a different language syntax and each furnishes a different set of features. For example, Oracle and Informix triggers are much more flexible than Sybase's. Using Oracle or Informix, a developer can specify whether the trigger is to be executed before the firing activity or after. Sybase triggers always fire after the firing activity occurs. This can greatly impact database and application design. With Sybase it is impossible to perform any activity in a trigger if it must occur before the event that invokes the trigger. Furthermore, Oracle and Informix triggers can be fired once per firing activity or once per row impacted by the firing activity. Remember, a single SQL update statement can modify multiple rows. It can be beneficial to run trigger code for each row impacted, instead of one global trigger for the entire activity. Sybase only provides a single execution per firing activity.

Server Code Object Programming Languages

Being application logic, most server code objects must be created using some form of programming language. Check constraints and assertions do not require procedural logic as they can typically be coded with a single predicate. Although different RDBMS products provide different approaches for SCO development, there are three basic tactics employed:

- Use a traditional programming language (either a 3GL or a 4GL)

- Use a proprietary dialect of SQL extended to include procedural constructs

- Use a code generator to create SCOs

DB2 for MVS takes the approach of employing traditional programming languages for the development of stored procedures (the only procedural SCO currently supported). Any LE/370-supported language can be used to code stored procedures. The current list of supported languages is Assembler, C/370, COBOL, COBOL II, and PL/I. DB2 stored procedures can issue both static and dynamic statement SQL statements with only a few specific exceptions.

The second approach is to use a procedural SQL dialect. One of the biggest benefits derived from moving to a RDBMS is the ability to operate on sets of data with a single line of code. Using a single SQL statement, multiple rows can be retrieved, modified, or removed. But this very capability limits the viability of using SQL to create server code objects. Sybase, Oracle, and Informix support procedural dialects of SQL that add looping, branching, and flow of control

statements. The Sybase language is known as Transact-SQL, Oracle provides PL/SQL, and the Informix dialect is called SPL (stored procedure language). Procedural SQL has major implications on database design.

Procedural SQL will look familiar to anyone who has ever written any type of SQL or coded using any type of programming language. Typically, procedural SQL dialects contain constructs to support looping (while), exiting (return), branching (goto), conditional processing (if...then...else), blocking (begin...end), and variable definition and usage. Of course, SPL, Transact-SQL, and PL/SQL are incompatible and can not interoperate with one another.

A final approach is to use a tool to generate the logic for the server code object. Code generators can be used for any of RDBMS that supports SCOs, as long as the code generator supports the language required by the RDBMS product being used. This approach is touted by IBM DB2 for MVS stored procedures and the VisualGen code generator. Of course, code generators can be created for any programming language, including procedural SQL dialects such as Transact SQL, PL/SQL, and SPL.

Which is the best approach? Of course, the answer is "It depends!" Each approach has its strengths and weaknesses. Traditional programming languages are more difficult to use but provide standards and efficiency. Procedural SQL is easier to use and more likely to be embraced by non-programmers, but is non-standard from product to product and can result in sub-optimal performance.

It would be nice if the developer had an implementation choice, but the truth of the matter is that he must live with the approach implemented by the RDBMS vendor.

The Duality of the DBA

Once server code objects are coded and made available to the RDBMS, applications and developers will begin to rely upon them. Although the functionality provided by SCOs is unquestionably useful and desirable, DBAs are presented with a major dilemma. Now that procedural logic is being stored in the DBMS, DBAs must grapple with the issues of quality, maintainability, and availability. How and when will these objects be tested? The impact of a failure is enterprise-wide, not relegated to a single application. This increases the visibility and criticality of these objects. Who is responsible if they fail? The answer must be—a DBA.

With the advent of server code objects, the role of the DBA is expanding to encompass too many responsibilities for a single person to perform the job capably. The solution is to split the DBA's job into two separate parts based upon the database object to be supported: data objects or server code objects.

Administering and managing data objects is more in line with the traditional role of the DBA, and is well-defined. But DDL and database utility experts can not be expected to debug procedures and functions written in C, COBOL, or even procedural SQL. Furthermore, even though many organizations rely upon DBAs to be the SQL experts in the company, often times they are not--at least not DML experts. Simply because the DBA knows the best way to create a physical database design and DDL, does not mean he will know the best way to access that data.

The role of administering the procedural logic in a RDBMS should fall upon someone skilled in that discipline. A new type of DBA must be defined to accommodate server code object and procedural logic administration. This new role can be defined as a procedural DBA.

The Role of the Procedural DBA

The procedural DBA should be responsible for those database management activities that require procedural logic support and/or coding. Of course, this should include primary responsibility for server code objects. Whether server code objects are actually programmed by the procedural DBA will differ from shop-to-shop. This will depend on the size of the shop, the number of DBAs available, and the scope of server code object implementation. At a minimum, the procedural DBA should participate in and lead the review and administration of SCOs. Additionally, he should be on call for SCO abends.

Other procedural administrative functions that should be allocated to the procedural DBA include application code reviews, access path review and analysis (from EXPLAIN or show plan), SQL debugging, complex SQL analysis, and re-writing queries for optimal execution. Off-loading these tasks to the procedural DBA will enable the traditional, data-oriented DBAs to concentrate on the actual physical design and implementation of databases. This should result in much better designed databases.

The procedural DBA should still report through the same management unit as the traditional DBA and not through the application programming staff. This enables better skills sharing between the two distinct DBA types. Of course, there will need to be a greater synergy between the procedural DBA and the application programmer/analyst. In fact, the typical job path for the procedural DBA should come from the application programming ranks because this is where the coding skill-base exists.

Problem Points

Of course, with every new idea come resistance. The following section anticipates some of the barriers to acceptance for a procedural DBA and a possible solution for overcoming the resistance.

Potential Problem #1: Some DBAs will not be content in only one role. Often DBAs are a curious lot who want to know it all. My company can not afford to alienate our highly skilled DBA staff by changing their job descriptions.

Potential Solution #1: Many times this will be a phantom problem. Many current DBAs do not know (or care to know) how to program. Those who do know SQL, do not want to write COBOL or C (and many of them do not want to know the intricacies of procedural SQL dialects (such as SYBASE Transact SQL or Oracle PL/SQL). Additionally, quite a few DBA staffs already have performance analyst/DBAs who are more programming literate and design DBAs who are more DBMS object literate. Implementing a procedural DBA in this type of organization should be easier than most. For those few shops where there are DBAs who do indeed wish to "know it all" cross training DBAs with primary and secondary roles should eliminate the resistance.

Potential Problem #2: We can not afford the DBAs we have now, how can we afford more DBAs?

Potential Solution #2: In actuality, most organizations can't afford not to have procedural DBAs. More and more of most company's business rules are being implemented in server code objects. This means the company can not afford the down-time and inefficiency when performance problems or abends can not be resolved in a timely manner.

Potential Problem #3: Our DBAs do all of this now. Why should we split the tasks into distinct roles when we get this support already?

Potential Solution #3: This could be true. If so, it is wise to delineate the role of each DBA and have them specialize. We specialize in all other areas from pediatricians to interior decorators. Specialization bring efficiency and rapid response.

However, this assertion could also be false. Investigate this argument and find out just what the DBA staff is doing. They might not have the time to review every piece of code that goes into production. This can be devastating for UDFs, triggers, and stored procedures, as they are intrinsically tied to data integrity and performance. Many DBA staffs are overworked and might not have time to re-write sub-par SQL. This can be a huge problem for procedural database objects because they don't affect just one program--they are reused and potentially can impact every program that accesses the database. It can also be problematic for strategic application programs that have not been thoroughly benchmarked and performance tested.

Potential Problem #4: We use multiple DBMS products, each with a different technique for coding triggers, stored procedures and functions.

Potential Solution #4: This argument reinforces the requirement for a procedural DBA. The more diverse and heterogeneous your environment is, the more you need to specialize. It makes sense to not only specialize by task or role (process objects vs. traditional database objects), but also to specialize by DBMS product. Just because someone is a SYBASE Transact-SQL wizard, that doesn't mean they will be equally adept at user-defined functions coded in C for DB2 for OS/2 (or even Oracle PL/SQL for that matter).

Potential Problem #5: No one in my company can do this type of job.

Potential Solution #5: If no one can do this type of job, it just means that your organization is not prepared for the database management trials and tribulations of the 1990s. That is tantamount to admitting that you are poised for failure. The answer is "Training!" Seek out skilled training organizations that can assist you immediately. And consider training a skilled staff of procedural DBAs to specialize in SCO and procedural database administration.

Synopsis

As vendors race to provide server code objects to support business rules implementation in their RDBMS products, database administration become more complex. The role of the DBA is rapidly expanding to the point where no single professional can be reasonably expected to be an expert in all facets of the job. It is high time that the job be explicitly defined into manageable components.

From Database Programming & Design, December 1995.

[Home](). Phone: 281-494-6153 Fax: 281-491-0637