# The procedural DBA

Until recently, the function of a DBMS was narrowly defined. Its only purpose was to store, manage, and access data. Although these core capabilities are still required by modern DBMS products, additional procedural functionality is no longer just a nice-to-have feature, but a necessity. The ability to define business rules to the DBMS instead of in a separate application program builds upon the concept of sharing data. However, instead of merely sharing data, modern DBMSs will enable applications to share both common data elements and code elements.

All the most popular RDBMS products are adding complex features and components to facilitate procedural logic. Stored procedures, user-defined functions, triggers, and complex encapsulated objects require procedural logic – sometimes very complex procedural logic. Storing logic in the database requires that organizations expand the way they have traditionally handled database management and administration. Typically, as new features are added, administering, designing, and managing these features is assigned to the database administrator (DBA) by default. This approach is not always the best one.

## THE ROLE OF THE DBA

The DBA's role has expanded over the years. In the pre-relational days, database design and data access were complex. Programmers were required to code program logic explicitly to navigate the database and access data. The pre-relational DBA was assigned only the task of designing the database structure. 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 back-up and recovery jobs, little more than space management and reorganizations were required. This is not to belittle these tasks. The pre-relational DBMS products required a complex series of utility programs to be run in order to perform back-up,

recovery, and reorganization. This process 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 modelling specialists. 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.

RDBMSs require additional involvement during the design of data access routines. No longer are programmers navigating the data – now the RDBMS is. Optimizer technology embedded in the RDBMS is responsible for creating the access paths to the data. To make matters more complex, Oracle offers multiple optimization choices – cost-based and rule-based. The DBA must review the optimization methods and choices. Program and SQL design reviews are now a vital component of the DBA's job. Furthermore, the DBA must perform additional monitoring and tuning responsibilities. Back-up, recovery, and reorganization are just a start. In addition, DBAs now use 'explain plan', performance monitors, and SQL analysis tools to administer RDBMS applications proactively.

Often, DBAs aren't adequately trained in these areas. It is a completely 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.

## COMBINING PROCESS AND 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. Oracle supports database-administered procedural logic in the form of stored procedures, triggers, constraints, User-Defined Functions (UDFs) and objects, and data cartridges.

Stored procedures are procedural logic that is maintained, administered, and executed through the database. The primary reason for using stored procedures is to move application code off the client workstation and onto the database server. This set-up typically results in less overhead because one client can invoke a stored procedure and cause the procedure to invoke multiple SQL statements. This approach 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, you can think of stored procedures as 'programs' that 'live' in the database.

Triggers are event-driven specialized procedures that are stored in, and executed by, the RDBMS. Each trigger is attached to a single, specified table. Think of triggers as an advanced form of rule or constraint written using procedural logic. A trigger cannot be directly called or executed; it is automatically executed (or 'fired') by the database as the result of an action – usually a data modification to the associated table. Once you create a trigger, it is non-bypassable. In other words, it is always executed when its 'firing' event occurs.

A constraint is a database-enforced limitation or requirement, coded into the definition of a table, which is non-bypassable. There are three types of constraints – unique constraints, referential constraints, and check 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 two tables that define the permitted specific data values that can be stored in those tables. Although they are both forms of constraints, they are also predefined to the DBMS and cannot be changed. Both unique and referential constraints do, however, require quite a bit of administration and management. The third type of constraint, the check constraint, is used to define the exact requirements for values that can be stored in a specific column. You can define a wide range of rules using check constraints because they are defined using the same search conditions that are used in SQL WHERE clauses. For example, the following constraint ensures that only red, white, and blue are acceptable colours:

```
CHECK (Color IN ('RED', 'WHITE', 'BLUE'))
```

A UDF 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 other built-in function.

Oracle8 uses object types to model real-world entities. An object type is a user-defined composite data type that meets application requirements. An object is an instance of a given object type. Object types build upon Oracle's existing data types (NUMBER, VARCHAR2, DATE, etc) to offer developers a broader range of data modelling capabilities. Each object type has attributes that describe the entity and the methods that act upon it. Object types do not physically store data; they can be thought of as templates that characterize the data being stored. The data is physically stored in the table that references the object type.

Object types stored in column positions are called column objects. Object types that are contained in other object types are called nested objects. All object types have at least one method, called the constructor method. The constructor method is an implicitly-defined method that makes an object of the given type. The constructor method is given the same name as the object type and is executed automatically when SQL statements, or PL/SQL code, create the type values. Oracle8 also allows you to create user-defined methods. A user defined method has complete access to attributes of its associated object and information about its type. Methods are invoked by referring to an object of its associated type.

Oracle8 also supports varrays. Arrays are ordered sets of elements and have been used by application developers since the first program was written! Arrays use indexes of numbers to point to specific elements. All of the elements in an array must be of the same type. The array's size depends on the number of elements it contains. Oracle8 builds upon this concept through varrays. Varrays and nested tables are referred to as collections in Oracle8. Varrays do not allocate storage

space and are variable in size. Varrays are useful for relatively small sets of objects; nested tables should be used for sets containing a large number of elements. A varray can be used as a data type of a relational table, an object-type attribute, or a PL/SQL variable, parameter, or function return type.

Finally, Oracle supports cartridges within its NCA (Network Computing Architecture). A cartridge is a manageable object package. Cartridges provide an Interface Definition Language (IDL) – a language-neutral interface that allows the cartridge to identify itself to other objects in a distributed system. The language-neutral interface allows developers to code cartridges in languages such as Java, Visual BASIC, C/C++, PL/SQL, and other languages. The Inter-Cartridge Exchange will enable cartridges, distributed across a network and plugged into different targets, to communicate with each other as well as clients, servers, and services. Oracle provides the following cartridges (at extra cost) for the Oracle8 server:

- Video Cartridge – delivers video and audio to business applications across the intranet.
- Image Cartridge – stores and retrieves two-dimensional static bitmapped images, including scanned documents.
- Time Series Cartridge – supports a variety of statistical and time-conversion functions to provide temporal analysis.
- Spatial Cartridge – allows the seamless integration of geographic data into business applications.
- ConText Cartridge – offers advanced linguistic capabilities for text searches and for generating themes and theme-summaries from all digital documents.

Indeed, Oracle8 provides a rich environment for integrating business rules and procedural logic into the database. This support impacts the entire application and database life-cycle from planning through development, implementation, testing, administration, management, and support.

## SERVER CODE OBJECTS

Stored procedures, triggers, constraints, and UDFs are just like other database objects such as tables, views, and indexes, in that the DBMS controls them. These types of object are often collectively referred to as Server Code Objects (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 objects may or may not 'physically' reside in the RDBMS. They are, however, always registered to, as well as maintained in conjunction with, the RDBMS.

Why are server code objects so popular? The predominant reason for using SCOs is to try and promote code re-usability. Rather than 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 – which 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 individual users used their own individual and separate code, there would be no assurance that the same business logic was being used by everyone. In fact, it is almost a certainty that inconsistencies would occur.

Additionally, SCOs are useful for reducing the overall code maintenance effort. Because SCOs exist in a single place (the database), you can make changes without having to propagate the change to multiple workstations. Another common reason to use SCOs is 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 communications lines.

Finally, SCOs can be coded to support database integrity constraints, implement security requirements, reduce code maintenance efforts, and support remote data access. With all of these benefits it is certain that SCOs will be utilized. But how does this impact administration?

## A NEW TYPE OF DBA

Although the functionality that is provided by SCOs is useful and desirable, DBAs are presented with a major dilemma. Now that procedural logic is being stored in the database, DBAs must contend with the issues of quality, maintainability, and availability. How and when will these objects be tested? The impact of a failure is not relegated to a single application but can potentially impact the entire organization – causing these objects to become more visible and critical. Who is responsible if they fail? The answer must be a DBA.

With the advent of SCOs, 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 cannot 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 they are not – at least not Data Manipulation Language (DML) experts. Simply because DBAs know the best way to create a physical database design and DDL, it does not mean they will know the best way to access that data.

The role of administering the procedural logic in an RDBMS should fall upon someone skilled in that discipline. We must define a new type of DBA to accommodate SCO and procedural logic administration. This new role can be defined as a procedural DBA.

The procedural DBA should be responsible for those database management activities that require procedural logic support and/or coding. Of course, this job should include having the primary responsibility for SCOs. Whether or not SCOs are actually programmed by the procedural DBA will differ from shop to shop. This decision will depend on the size of the shop, the number of DBAs available, and the scope of SCO implementation. At a minimum, procedural DBAs should participate in and lead the review and administration of SCOs. Additionally, procedural DBAs should be on call for SCO application failures.

If a centralized procedural DBA is not available, consider what happens when a shared stored procedure (for example) fails. A demand deposit application is developed by a large commercial organization that requires customer information. A stored procedure (or UDF) is developed to return customer information on request. The demand deposit application calls this code whenever it requires customer information. The development is completed and the application is migrated to production. A few months go by and another application is being developed (say, a trust accounting system) that requires customer information. The decision is made to use the stored procedure that exists. The application is completed and turned over to the production environment. A few weeks later, the trust application bombs in the customer stored procedure. Who comes in to fix it? The trust folks claim that it is not their code, they don't understand it, and are incapable of fixing it. The demand deposit folks claim that their application is running fine so it's not their problem. The answer is to employ a procedural DBA to review, revise, monitor, fix, and perhaps even code the common SCOs. In that case, the procedural DBA will be 'on call' to correct the problem.

Other procedural administrative functions that should be allocated to the procedural DBA include application code reviews, access path review and analysis (from EXPLAIN PLAN), PL/SQL debugging, complex SQL analysis, rewriting queries for optimal execution, and complex object review and support. 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 approach should result in much better designed databases.

The procedural DBA should still report through the same management unit as the traditional DBA – not through the application programming staff. This set-up enables better skills sharing between the two distinct DBA types. Of course, a greater synergy must exist between the procedural DBA and the application programmer/analyst. In fact, the procedural DBA should move up in the ranks from application programming, building from the existing coding skill-base.

There are, however, potential problems that may be encountered when implementing a procedural DBA role. Firstly, some DBAs will not be content in only one role. Often DBAs are curious and want to know everything about everything. But many currently do not know (or care to know) how to program. Those who know SQL do not want to write C or Visual BASIC (and many of them do not want to know the intricacies of procedural SQL dialects like PL/SQL). Additionally, quite a few DBA staff already have performance analyst/DBAs, who are more programming literate, and design DBAs, who are more database object literate. Implementing a procedural DBA position in this type of organization should be easier than in most. For those few shops that have DBAs who do indeed wish to 'know it all', cross-training DBAs with primary and secondary roles should eliminate the resistance.

Another potential problem might occur if implementing a procedural DBA requires additional staff. Many organizations believe that they cannot afford the DBAs they have now – so how can they afford more DBAs? In fact, most organizations can't afford not to have procedural DBAs. More and more of the organization's business rules are being implemented in SCOs and complex objects, which means the company cannot afford the downtime and inefficiency when performance problems or failures cannot be resolved in a timely manner.

A third problem could arise if your company believes that your DBAs do all this already. Why should they split the tasks into distinct roles when the support already exists? This point could be true. If so, it is wise to delineate the role of each DBA and have them specialize.

Specialization brings 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 neglect can be devastating for UDFs, triggers, and stored procedures, since they are intrinsically tied to data integrity and performance. Many DBA staff are overworked and might not have time to rewrite poorly coded SQL and PL/SQL. This can be a difficult 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.

Finally, there is the heterogeneous argument. It goes something like: 'We use multiple DBMS products, each with a different technique for coding triggers, stored procedures, and functions. No one person can know all of them'. This argument is true, but it actually reinforces the requirement for a procedural DBA. The more diverse and heterogeneous your environment is, the more you need to specialize. It makes sense not only to specialize by task or role (process objects versus traditional database objects), but also to specialize by DBMS product. Just because someone is an Oracle PL/SQL wizard, it doesn't mean they will be equally adept at user-defined functions coded in C for DB2 or Sybase Transact-SQL stored procedures.

## SYNOPSIS

As procedural logic becomes more and more pervasive in your Oracle applications, database administration becomes more complex. The duties performed by a DBA are expanding to the point where no single professional can reasonably be expected to be an expert in all facets of the job. It is high time that the job be explicitly defined into manageable components. We need procedural DBAs.

*Craig S Mullins*
*Vice President Operations*
*Platinum Technology (USA)*