BY DAVID McGOVERAN

More easily remembered and executed, stored procedures are a means of extending and customizing SQL to the needs of an environment and significantly reducing overhead; here's a look at how they work

THE Power OF Stored Procedures

AMONG THE CURrent extensions to standard SQL offered by various vendors, stored procedures are probably the least understood. When a vendor says that they offer procedural extensions to SQL, most of us know they mean the ability to combine IF-THEN, WHILE, BEGIN-END, and similar procedural constructs with SQL statements. Even here, features differentiate the products, such as the ability or inability to declare local variables. But by and large, the concept of "procedural SQL" is universally understood.

Stored procedures, however, can mean quite different things to different vendors. ANSI's SQL committee has been addressing issues such as stored procedures, triggers, and referential integrity, but has yet to issue a standard syntax. In this article, I will try to characterize the variations among vendors, give some examples, and explain some of the more powerful uses of stored procedures. I will also point out pitfalls and problems and look at what the future is likely to hold.

Stored procedures offer a kind of object-oriented approach to relational database management. The stored procedure is used to define the operations (or methods) that can be performed on the object. Since most relational DBMSs do not provide an integrated means of defining objects other than through tables, the object is an abstraction defined by a set of stored procedures that manipulate it. The data of an object instance can be represented by the rows in a single table or multiple tables. By excluding permission to modify the base tables, the object can be encapsulated; that is, the data representing the object can be manipulated only through the stored procedures. If the relational DBMS supports nested procedures, a form of inheritance can be provided.

As an example, consider a set of stored procedures that provide various legal operations on a set of tables representing a general ledger. Regardless of how the data required to support a ledger is distributed among the tables, the user only needs to perform certain

ARTWORK: R. FORRESTER

operations (credit, debit, add, delete, view, and so forth) on the necessary accounts.

These operations are provided via stored procedures. Access to the base tables is never granted the user—no data access is possible except via these procedures. By using a consistent nomenclature, the names of stored procedures can even be made to look like object-oriented messages: the first part of each stored procedure names the object and the second part names the action (method) to be executed.

HOW THEY ARE USED

The uses of stored procedures are many and can be divided into approximately the three categories of performance, management, and development.

Performance. Stored procedures improve performance in a number of ways. Because stored procedures need not be fully interpreted at runtime, some redundant overhead associated with frequently repeated SQL statements is eliminated. Depending on the product, this can include overhead associated with parsing, validity checks, permission checks, and query optimization (Figure 1).

Stored procedures eliminate much SQL redundancy

If the stored procedure is cached in memory, it can be executed rapidly without incurring the I/O associated with looking up the procedure on disk. Even if the stored procedure is not in cache, the disk I/O overhead to retrieve it is often considerably less than the terminal I/O overhead required for a program to make the same request in SQL. The degree of improvement depends on the complexity of corresponding SQL statements.

In a distributed processing system (which may or may not involve distributed database management), network traffic can be greatly reduced by issuing a single procedure request rather than a request per SQL statement. This improves the likelihood that a given request will transmit error-free by eliminating the need to send the entire text of a set of SQL statements across the network. Even in a virtual client-server environment (that is, both on the same physical machine), communications overhead is reduced.

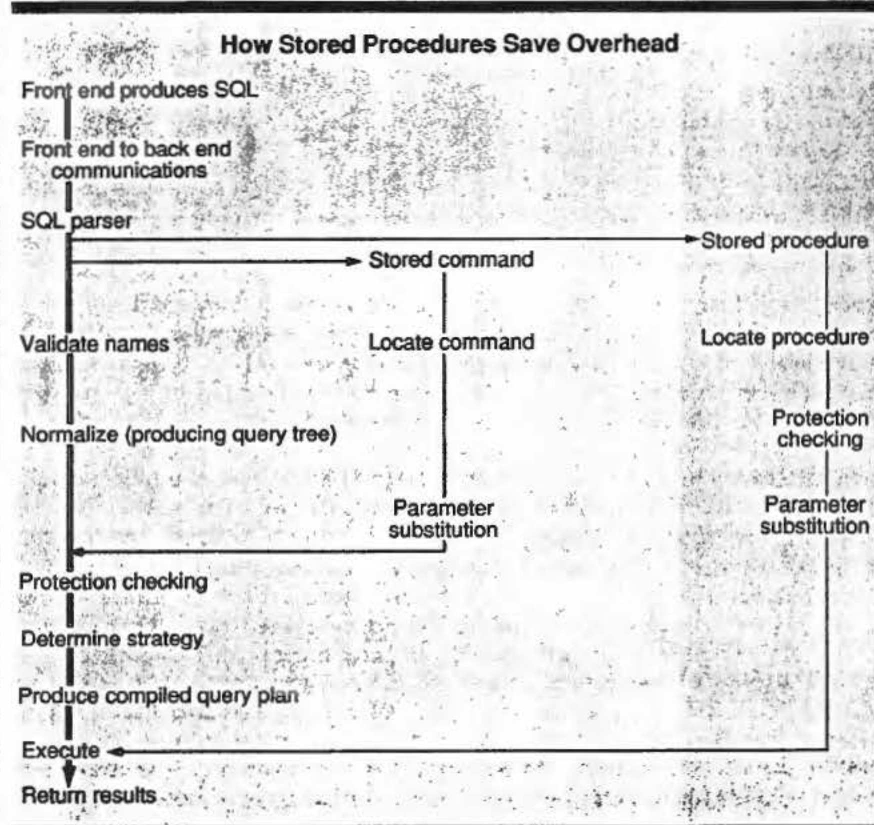

FIGURE 1. *A typical SQL statement is processed through the steps on the left path, while stored commands and procedures take the less costly right-hand paths.*

Remember that distributed processing and/or data management is not the same as distributed DBMS. Remote stored procedure support does not constitute distributed DBMS implementation, although it is very powerful.

The functionality of stored procedures is essential for maintaining site autonomy in a distributed processing environment. By allowing remote modification only through stored procedures, the local DBA can have complete control over a site's data.

Parameter substitution and returning user-defined return codes further decreases communications overhead. Parameter substitution eliminates passing the same data value more than once even when it is used multiple times in perhaps multiple SQL statements. User-defined return codes provide a compact means of informing the application about stored procedure processing. A single, judiciously chosen integer return code can completely alter subsequent front-end processing, including the interpretation of returned results tables.

Often, if the vendor supports procedural extensions to SQL and/or multiple statements per stored procedure, requests to the database can be processed asynchronously with front-end processing. The ability to issue a complex request that provides for conditional execution of SQL statements and error processing eliminates much of the front-end's work.

Suppose that your application requires that you obtain data from one table if a certain condition exists and from another table if a different condition exists. While this can be done with a complete set of relational operations, when many conditions exist on multiple tables the resulting SQL can be so complex that no query optimizer can find the optimal access plan in a reasonable

time. Procedural extensions to SQL used with stored procedures simplify processing, make the third generation language (3GL) easier to read and write, and provide a means for the application to obtain meaningful results regardless of the base tables that must be accessed. This means fewer requests to the database and leads to higher throughput overall (Listing 1a-d).

Finally, some relational DBMSs allow multiple applications to use a stored procedure. Just as most modern operating systems provide a means to create program libraries for use by multiple applications, relational DBMSs using stored procedures provide a means for creating a library of SQL code. This way, total memory and disk space required to cache and store all procedures, respectively, can be reduced. While this is particularly important for managing cache memory, it may not be desirable for too many applications to share a stored procedure. Further cache savings can be obtained if the stored procedures are re-entrant.

Management. As a management tool, stored procedures are a powerful means of exerting access control. Certainly one can GRANT read or write permission on a table-by-table basis to users. However, to do so without strong domain, entity-relationship, and referential integrity controls is too risky in many applications.

Consider, a banking application in which a data entry operator is supposed to update accounts. To GRANT write permission on the table containing account balances involves considerable trust. The value of a debit might not be within the allowed range for the particular customer; also, complex qualifications against the customer's history, current bank policy for that type of account, the data entry operator, and other circumstances make blanket write permission on the table undesirable.

If the rules involved are universal for all applications that update the table, then using an integrity mechanism such as triggers becomes viable. However, most such rules have meaning only on an application-by-application basis. In these circumstances, permission on the base tables can be de-

Listing 1a. Embedded SQL example.

```
main()
{
    int first_idno;
    int last_idno;

:
/*
**  initialize variables through some means  then execute
**  embedded SQL transaction
*/
:
:
:
:
:
for (i = first_idno; i < last_idno; i++)
    {

EXEC SQL INSERT INTO toberesolved
    SELECT * FROM employee
    WHERE idno = :emp.idno;

EXEC SQL DELETE FROM employee
    WHERE idno = :emp.idno;

EXEC SQL COMMIT WORK
    }
:
:
}
```

Listing 1b. Embedded database procedure. *By creating a database procedure as in Listing 1c, the program is reduced to:*

```
main()
{
    int first_idno;
    int last_idno;

:
/*
**  initialize variables through some means  then execute
**  procedure
*/
:
:
EXEC SQL EXECUTE PROCEDURE build_resolve_list
    (first_idno = :first_idno, last_idno = :last_idno);
}
```

Listing 1c. Database procedure definition *(defined outside the program). Courtesy of Relational Technology Inc.*

```
CREATE PROCEDURE build_resolve_list
    (first_idno INTEGER, last_idno INTEGER) AS
DECLARE
    i INTEGER;
    BEGIN
        i = :first_idno;
        WHILE ( i < :last_idno) DO
            INSERT INTO toberesolved
                SELECT * FROM employee
                WHERE idno = i;

            DELETE FROM employee
                WHERE idno = :emp.idno;

            COMMIT WORK
            i = i + 1;
        ENDWHILE;
END;
```

LISTING 1a-c. *Stored procedure example.*

nied and permission to execute a stored procedure given instead. This precludes modifying the table in any way other than in the prescribed manner.

Using stored procedures in this way leads to the possibility of asserting not only application-specific business rules but general ones as well. Referential integrity can be maintained with sufficient effort, although other mechanisms may be preferred. For example, triggers can be used to cause cascade deletes and updates by specifying them once, whereas each stored procedure would have to be written to execute all the appropriate modifications on all tables that the procedure modifies. Analyzing the requirements can be difficult enough in complex database applications without trying to maintain all the necessary SQL.

In a sense, stored procedures provide a means to extend and customize the SQL language to the needs of an environment. Each stored procedure name becomes a high-level verb that users can remember and execute more easily.

Procedures can be used to update the base tables of views

The complexities of typing correct-as-intended SQL interactively in all but the most straightforward applications make SQL an undesirable end-user language. Even if one is careful not to commit changes resulting from unintended SQL and roll them back, the cost of doing so can be overwhelming. Not only might locks have been held unnecessarily (reducing concurrency), but the consumed CPU time could be great.

Indeed, there is no way to eliminate retroactively the cost of having processed an unintended SELECT statement. Consider the following (possibly too transparent) example with two SQL statements. The only difference existing between them is one character, but the cost is tremendous if MY_TABLE is very large:

```
SELECT * FROM MY_TABLE
WHERE KEY_FIELD1 = KEY_FIELD2

SELECT * FROM MY_TABLE
WHERE KEY_FIELD1 = KEY_FIELD1
```

Stored procedures can be used to greatly diminish the likelihood of such typographical errors, as well as the potential resource costs, through the selection of meaningful procedure names. These steps make SQL a more customized, user-friendly language.

Stored procedures can be used to update the base tables of views, although one must be careful not to treat this capability as though it were view update support. Rather, it provides a means of implementing functions that are equivalent to views and of defining the inverse functions.

Using stored procedures in this way is quite powerful and may be more desirable than view update, considering that views are used primarily to assert security and as a convenience for simple tabular reports. However, because the general view update problem is unsolvable, views are not tenable for controlling updates or inserts except in simple cases. Stored procedures offer a means of modifying base tables as well as performing the function of a select on a view at the expense of using a different syntax.

Experience with relational applications has shown that as the number of embedded SQL modules increases, so does the cost of making changes to the database schema or to the SQL for performance optimization, because these modules may well have to be recompiled and relinked. Although changes to the database schema are easily effected, the impact on applications is costly. Eventually, the cost becomes so great that the

```
CREATE PROCEDURE sp_addlogin
@loginame VARCHAR(30),
@passwd   VARCHAR(30) = NULL,
@defdb    VARCHAR(30) = "master"
AS
DECLARE (@msg varchar(250)
IF ssuser_id() != 1
BEGIN
     PRINT "Only the System Administrator may EXECUTE this procedure".
     RETURN(1)
END
/* Use nested procedure to check that @loginame is valid. */
DECLARE @returncode INT
EXECUTE @returncode = sp_validname @loginame
IF @returncode != 0
BEGIN
     SELECT @msg = "'" + @loginame + "' is not a valid name."
     PRINT msg
     RETURN @returncode
END
IF EXISTS(SELECT * FROM syslogins WHERE name = @loginame)
BEGIN
     PRINT "A user with the specified login name already exists."
     RETURN(1)
END
IF NOT EXISTS(SELECT * FROM sysdatabases WHERE name = @defdb)
BEGIN
     PRINT "Database name not valid -- login not added."
     RETURN(1)
END
INSERT INTO syslogins(suid, status, accdate, totcpu, totio,
     spacelimit, timelimit, resultlimit, dbname, name, password}
SELECT MAX(suid)+1, 0, GETDATE(), 0, 0, 0, 0, 0, @defdb,
     @loginame, @passwd FROM syslogins
PRINT "New login created."
RETURN(0)
GO
```

LISTING 1d. *A more complicated nested procedure (courtesy Sybase).*

advertised flexibility of relational databases is lost.

Development. Perhaps the most significant use of stored procedures is that they provide a means of enforcing schema transparency (see "Twelve Rules for Stored Procedures," Rule 1). For a 3GL application, executing operations within the DBMS can be viewed as a (possibly complex) series of function or subroutine calls that happen to manipulate data in a data store. Good program design dictates that functions should hide particular file access methods, the particular file structure, and any local data elements. It should be possible to treat the function as a black box, minimizing the degree of coupling between modules.

Stored procedures allow the database programmer to extend this design philosophy into the DBMS. The stored procedure "function" is specified by a set of input and output parameters and the work to be done. By uncoupling the application from the database schema, the structure of the database can evolve without invalidating applications. As well, the SQL code can be altered to accommodate changes or for performance optimization.

The 3GL programmer need only think in terms of, and communicate to, SQL coders in standard (and very familiar) black box functional specification: the "goes into's," "goes out-of's," and the "gotta do's." If these change, then, by definition, the application requirements have changed. All other changes are internal to the relational DBMS "function." Source code control, impact analysis, debugging, and release management are all significantly improved over the embedded SQL approach.

With the 3GL programmer uncoupled from the details of the SQL and the database schema, productivity and staffing improve (see "Twelve Rules for Stored Procedures," Rule 2). The 3GL programmer can code a series of stubs supplied by the SQL specialist. The programmer need not even learn SQL; nor would the SQL specialist need proficiency in the particular 3GL, thus making large projects easier to staff and manage.

If the 3GL programmer has written general purpose routines that use stored procedures, considerable flexibility can be transferred from the relational database to the application. In effect, applications become not only table-driven, but SQL-driven. Significant changes to the details of the application can be made by changing the SQL that a stored procedure executes without modifying, recompiling, or relinking 3GL code. At most the user has to shut down the application and reinvoke it after a brief pause. In many cases, automatic recompilation is sufficient to keep things going without the application's user ever knowing the difference.

The vendor can supply stored procedures that update or report on systems tables without subjecting the DBA to the details of the SQL involved. Once learned, these stored procedures are a special and efficient DBA command-line language that need not change. Even when significant changes are made to the systems tables they can be increasingly hidden from the user with new releases of the relational DBMS software.

For example, such procedures can be used to create exam-

Twelve Rules for Stored Procedures

Stored procedure implementations that meet the following rules provide a means for developing flexible RDBMS applications:

Rule 1: Schema transparency. Information-preserving changes to the DBMS schema have no effect on the invocation or execution of the procedure when such changes theoretically permit unimpairment.

Rule 2: DML and DDL transparency. The particular data manipulation language (DML) or data definition language (DDL) used in defining the procedure has no affect on invocation or execution of the procedure, including changes as severe as selecting SQL versus QUEL.

Rule 3: DBMS location transparency. The location of tables on which the procedure depends does not affect invocation or execution of the procedure.

Rule 4: Procedure transparency. The procedure is treated like any other database object, is maintained in the database system catalog, can be executed in a manner consistent with the syntax of the DML, and can be shared by all users.

Rule 5: Domain transparency. Changes to column domains does not affect invocation or execution, or parameter definitions.

Rule 6: Syntax transparency. Changes to definition syntax do not change the invocation or execution method where changes theoretically permit unimpairment.

Rule 7: Complexity independence. Invocation and execution is independent of the procedure definition's complexity.

Rule 8: Detailed diagnostics. Detailed error information is available.

Rule 9: DML and DDL completeness. All DML and DDL can be used within a procedure.

Rule 10: No DBMS-imposed restrictions. There are no practical limits on the number of parameters, statements, or definition size.

Rule 11: Security completeness. A means of controlling execution permission is provided, consistent with the DML and DDL.

Rule 12: Transaction scope independence. Transactions can span and be embedded in procedures.

ple databases for training purposes. As the example database is improved, there is no need for the DBA to create new tables explicitly since these may be hidden in a stored procedure. Similarly, if database statistics encoding changes (from text to float, for example) or if information such as the number of disk I/Os is maintained in a new table, a stored procedure can supply the information in humanly readable form so that the user is never aware of the changes.

Very often, entire applications can be written using stored procedures. At least two vendors use stored procedures to enter, track, and report on customer support. The application consists of a set of stored procedures, which become a highly customized command-line language.

Even if a command-line driven application is not desirable, it can be a means of rapidly prototyping required functionality. At the same time, a menu- or windows-driven front end that uses stored procedures need not contain all the database functionality during prototyping. This allows the developer to prototype an relational DBMS application even before the database schema is completed. The developer can use a test database that need not resemble the final schema.

Over time, SQL redefinition of the stored procedure can be made to access the proper tables, become more complex, and be optimized until a fully functional production system is obtained. Thus, the development of database access and manipulation can proceed in parallel with development of the user interface and be merged transparently.

THE FUTURE

We can expect stored procedures to become even more powerful than they are today. They figure prominently in several vendors' distribution strategies for several reasons:

- They minimize network traffic.
- They provide site autonomy in a way SQL cannot.
- They can help improve performance.

Stored procedures can hide sins that should not exist

To the programmer, the syntax to invoke a stored procedure will be the same regardless of where the procedure is stored. The relational DBMS will be able to migrate the procedure intelligently to the client or server, depending on where the procedure is most often invoked or will be processed most efficiently.

Load distribution of this sort will be very important in complex distributed systems. Dynamic load-leveling is not the only way in which this will work. Where the network is complex and distributed query usage patterns are well-established, dynamic optimization may be too costly. Instead, comprehensive optimization algorithms may be used to precompile the best distribution and access plans globally (for all stored procedures) rather than locally (on a procedure by procedure basis).

Stored procedures will also be able to work with user-defined access methods, including user-defined gateways. Used along with nested and remote procedure capability, an evolutionary path towards integrating corporate data becomes possible.

Even more useful would be a variation on stored procedures, which would allow the DBA to define precisely what is meant by update, insert, or delete operations on a view. For example:

```
CREATE {UPDATE | INSERT | DELETE} OF VIEW
view_name
    AS
    BEGIN
    SQL_statements
    END;
```

Whenever the user issued a statement like:

```
DELETE VIEW view_name WHERE column_name
= value;
```

any WHERE clause conditions within the definition of the view delete would have to be met in conjunction with those specified at runtime by the user. In this way, the SQL user could use standard SQL syntax to further manipulate the view table.

Of course, such a proposal leads to some interesting problems. The rules by which such resolutions are to be accomplished would ideally be prescribed by a standard rather than by the vendor. Nonetheless, I suspect that most DBAs would rather be able to update views in a controlled and vendor-prescribed manner than not at all.

THE DARK SIDE

All is not rosy in the world of stored procedures. Though they provide flexibility, stored procedures can hide a lot of sins that should not exist. If a stored procedure is improperly defined, the 3GL designer may not know the expected functionality is not being achieved. Similarly, stored procedures make it easier to lose control of the interaction between SQL and 3GL programs in terms of transaction management and recovery.

As noted earlier, the failure mode of a particular statement in a stored procedure can determine whether it is necessary to rollback an entire transaction. Multiple procedures may be invoked within a transaction, and transactions can either begin or end within a procedure. Shifting what work is accomplished by a procedure (within a sequence of procedures) can inadvertently move critical work outside the transaction boundary. As a result, all the work will not rollback on an abort. Avoiding such possibilities requires good communication between the 3GL and SQL developer—or profound understanding if they are one and the same person.

The potential complexity of stored procedures makes it clear that the scope (in terms of the number and duration of locks acquired) of a stored procedure in a transaction needs to be examined and controlled. This is, of course, no different than the concern one should have over any transaction.

However, modifying a stored procedure indiscriminantly can result in unexpected deadlocks and excessive lock wait times for other

applications, especially if transaction management is handled outside the stored procedure definition. For this reason, it is wiser to write stored procedures as standalone transactions whenever possible, or for the statements in them to be outside a transaction altogether.

Stored procedures can be extremely powerful. For example, invoking a single stored procedure can cause wholesale update or deletion of rows from many tables. Parameters can control the number of rows affected in such a procedure. In the proper hands, such power is clearly an advantage when the parameters are properly chosen. Because permission to execute stored procedures is separate from that for base tables, a stored procedure in the wrong hands can be even more dangerous than (and can circumvent denial of) write permission on a critical table since it can affect many critical tables.

Stored procedures do not alleviate the need to understand and control security in a system. This need is easier to meet if user security groups are classified according to the "need to execute" groups of stored procedures. A policy such as "applications users cannot modify any base tables" should then be implemented. Security concerns emphasize the need for user-friendly security management with relational DBMSs. The ability to GRANT or REVOKE execute permission on entire groups of stored procedures is highly desirable.

Problems with stored procedures will probably be solved as relational DBMS products and applications development methodologies mature. All in all, stored commands, database procedures, and stored procedures are very powerful. They promote exactly the kind of programming discipline that has long been espoused and make it possible in a relational, nonprocedural environment. ■

The author would like to thank Ed Horst of Relational Technology Inc. and Howard Torf of Sybase for their cooperation.

David McGoveran is president of Alternative Technologies in Santa Cruz, Calif., a consulting firm specializing in relational database applications.

The Implementation To Fit Your Needs

VENDORS REFER TO stored procedures and their variants by many names: precompiled queries, stored queries, stored commands, database procedures, preprocessed procedure blocks, stored front-end procedures, and database request module. Depending on what the vendor had in mind, stored procedure implementations are equally different.

The following are questions you might want to use in evaluating whether a vendor's implementation is appropriate to your needs. After this section, we will look at four implementations of stored procedures and see how vendors have answered many of these questions.

1. **How many SQL statements are allowed in a single procedure?** If the implementation allows only a single SQL statement, little user-defined functional benefit can be achieved with the procedures. Instead, the procedures become a kind of shorthand for specific user-defined SQL statements.

2. **What kinds of SQL statements are not allowed?** If SELECTs are not fully supported, procedures cannot be written to generate tabular reports, such as to view daily balances for a chart of accounts. If CREATE TABLE is not supported, then the DBA can't use procedures to manage the schema.

3. **What restrictions exist on the amount of select data or other information that can be returned?** If the implementation returns only a single row or a single column value, extracting virtual tables from the database is impossible.

4. **Can the procedure return error codes?** Without support for error codes, it's impossible to differentiate between the errors that might cause a procedure to end without its desired result.

5. **Are error conditions handled within the procedure?** If nested procedures are supported, it's impossible to maintain the integrity of the sequence of (possibly conditionally executed) events the procedure defines. Without support of nested procedures, the ability to detect and handle errors in the procedure eliminates unnecessary handling by applications and standardizes error processing across the database.

6. **If the vendor supports procedural extensions to SQL, which ones are allowed within a stored procedure?** For example, if it fails to support loop constructs, the programmer is forced to code statements explicitly for each iteration. This makes the procedure unnecessarily complex, and it may then exceed size limitations.

7. **Is the procedure definition compiled:**

☐ **Down to optimized machine language?** This approach removes the most overhead under normal conditions. The resulting code is highly efficient, but a degree of flexibility may be forfeited. It is less likely that such code is shareable among users or applications and the cost of recompilation is higher.

☐ **To an access or query plans (along with procedural extensions as necessary)?** This eliminates the overhead incurred in parsing the statements and selecting an access strategy, leaving only security checks and parameter substitution prior to execution.

☐ **To a query tree?** This technique eliminates parsing overhead, access plan selection (optimization), security checks, and parameter substitution that occur prior to execution. If parameter values can affect the optimizer, this may be better for producing a precompiled access plan.

8. **Is the procedure definition interpreted (SQL stored as text)?** Without precompilation and with only text stored, no overhead is saved; the user must repeatedly process the procedure.

9. **Is the procedure definition stored:**

☐ **In the database?** This allows for automatic recompilation if the schema or permissions change, and for integrated maintenance source control of procedure definitions.

☐ **In a host file?** This method makes it difficult to support automatic recompilation and may lead to such problems as multiple versions of the same procedure.

☐ **In the 3GL or 4GL program?** When associating a procedure with such a program, it is unlikely that procedures are shared across applications. It defeats the benefit of treating procedures like language extensions to SQL.

10. **Can applications share procedure definitions?** If not, different applications can manipulate database data inconsistently. The proliferation of procedures represents a code control and impact analysis problem.

11. **Can procedures be invoked interactively?** If not, having to debug procedures is time-consuming. Furthermore, their benefits can't be passed on to the interactive user.

12. **Does the database *detect* dependency problems:**

☐ **By time and date stamp?** This method compares the time and date of the last change of a stored procedure to that of the objects on which it depends. This is a powerful method, which can consume some overhead each time the procedure is run.

☐ **By dependency list?** This is a brute force method that forces recompilation whenever an object on which the procedure is defined is changed.

☐ **By validity flag?** Whenever a database object is modified, all procedures on which it depends are marked as invalid. These procedures are then either automatically recompiled on the next invocation or else fail and must be manually recompiled. This method makes the overhead at runtime small.

☐ **By failure?** Letting the stored procedure fail (perhaps after successfully executing a statement) puts database integrity at risk. Recovery in this way is left to the application or user.

☐ **Not at all?** If no checking is done and failures are not reported, procedures can be used only in noncritical or static databases. Integrity can't be guaranteed.

13. **How does the database *resolve* dependency problems:**

☐ **Automatically at execution time?** This technique frequently eliminates manual intervention in maintaining stored procedure definitions. But, resolution automatically at execution time could introduce semantic errors in processing. Thus, the first invocation after a change could be delayed.

☐ **Automatically when the dependency is changed?** This technique eliminates overhead at first invocation, but may inadvertently make concurrently running applications that use the procedure unrunnable. The solution is to use a retry mechanism within the application or to take dependent applications offline for a short time while redefinition takes place.

LISTING 2. *Syntax used by Sharebase.*

```
STORE procedure_name
     SQL_statements
END STORE

[START] procedure_name[:owner] [(
  [parameter_name =]
   value[, [parameter_name =] value]
  )];

GRANT START ON procedure_name TO
 { user_list | PUBLIC};

DROP procedure_name;
```

☐ **Manually?** Manual redefinition and resolution of dependency problems require using all the usual techniques for source code control and new version release. Runtime overhead is eliminated. If the system manager is clever, tools can be developed that aid the process.

14. **When invoked, can these procedures take formal parameters:**

☐ **Ordered list?** An ordered list eliminates semantic coupling between the definition of procedure variables and application variables. This is both a blessing and a curse since errors are harder to detect, but higher modularity is achieved.

☐ **Named list?** This eliminates the possibility of passing the wrong value to a particular parameter, but semantic coupling of this sort decreases modularity. If parameter names are changed, applications must be edited and recompiled to reflect the change.

☐ **With automatic data type conversion?** If this is is not supported, the parameter passing mechanism is

THE RELATIONAL INSTITUTE PRESENTS

DISTRIBUTED DATABASE '89

October 24-26, 1989—Fairmont Hotel— San Jose, California

Join DR. E.F. CODD, CHRIS DATE, and COLIN WHITE in the essentials of Distributed Databases, today and tomorrow. Learn from the vendors of products which cross all current platforms...from **PC—MINI—to MAINFRAME.**

Event will cover issues on

- 12 Rules of Distributed Database
- Applying the Technology
- Issues and Applications
- Nonlocking Concurrency
- Heterogeneous DDBMS(s)
- The Marketplace
- Distribution Independence
- What the Future Holds

Vendors

- Cincom Systems, Inc.
- Computer Associates, Int'l.
- Digital Equipment Corporation (DEC)
- Informix Software, Inc.
- Interbase Software, Inc.
- Oracle Corporation
- Relational Technology Inc. (RTI)
- Tandem Computers, Inc.
- Sybase, Inc. • Via Information Systems, Inc. • Microrim

WHO SHOULD ATTEND?

Data processing or computer science professionals, system planners, database administrators, system architects, and managers.

For registration or information call **THE RELATIONAL INSTITUTE** (408) 268-8821 Suite 106, 6489 Camden Avenue, San Jose, CA 95120, FAX (408) 997-6641 or our **east coast number** (201) 906-7979, FAX (201) 906-7993 — 24 Hour Registration.

CIRCLE 19 ON READER SERVICE CARD

strongly typed. While aiding debugging, this forces the application to perform the type conversions.

15. **What is the maximum number of parameters allowed?** If too few parameters can be passed, multiple applications can't be supported. It isn't uncommon in complex database applications to encounter wide tables; thus, an update in a procedure requires at least as many parameters as there can be columns in the table.

16. **Is the optimizer syntax-sensitive for a stored procedure?** Optimizer syntax sensitivity is unfortunate in any case. However, a stored procedure should hide the definition from the user as much as possible. In this way, changes to the schema can be relatively transparent to users and applications ("schema transparency"). While such transparency can never be perfect, it is a goal optimizer syntax sensitivity defeats.

17. **What diagnostic information is returned and how can it be processed:**

☐ **Upon failure, is a definitive error code returned?** It is essential to know whether or not the error is due to a failure of the procedure definition or a missing procedure definition.

☐ **Is a single success or failure code returned for the entire procedure?** If the success or failure of the procedure as a whole is all that is reported, it becomes difficult to write error-processing code. Alternatively, if success or failure is returned as a first order error code with more detail available via a special function call or access to a special error structure, it is possible to write more efficient error processing code than if only detail codes are returned.

☐ **Upon failure, can the statement number and type be identified?** When attempting to debug a failed procedure, it is important to know which statement failed. At runtime, much modularity can be achieved in processing the code if the type of statement can be identified. In particular, it is important to know whether a table will be returned by the next statement. Similarly, knowing how many rows are affected by each statement can be helpful, although this should be a general characteristic of the 3GL or 4GL interface, whether embedded or runtime.

☐ **Can exception handlers be declared to respond to particular errors?** This kind of error processing is extremely efficient and clearly necessary for a proper interface to 3GLs.

18. **Are dependencies identified and maintained automatically?** This lets the vendor or user design a database schema impact analysis tool. If the user is able to associate procedures with applications that use them and associate between the program variables and parameters, select lists, error, or return parameters, it is possible to develop full impact analysis tools. Without impact analysis, maintenance becomes laborious.

Here's how IBM's DB2, Sharebase's Sharebase, Sybase's Sybase, and Relational Technology Inc.'s Ingres answer many of these questions.

DB2. When a DB2 3GL program is precompiled, the SQL statements are stripped out and replaced by host language CALL statements. The SQL statements are converted by the SQL compilation step (bind) into an optimized machine code called a Database Request Module, which is then stored in the system catalog or dictionary for access by the runtime supervisor.

If a database object is altered, DB2 checks all application plans and marks them as invalid if they depend on the object. When such an invalid plan is retrieved for execution, it is then recompiled. It causes a slight delay for the user, but is otherwise transparent unless an object necessary to the execution of the SQL is no longer appropriate and no replacement object is available.

Database Request Modules are a very primitive kind of stored procedure. They are compiled, thus eliminating some of the overhead associated with parsing and generating access plans. The modules are stored in the database under the system catalog. They can consist of any valid DB2 SQL. However, they may not be shared by applications and do not support procedural extensions to SQL. Recompilation is automatic.

Sharebase. Previously known as Britton Lee, Sharebase has provided what they call "stored commands." The relational DBMS is an interpretive system. Stored commands consist of one or more SQL (or IDL, a variant on QUEL) statements. Since Sharebase does not offer procedural extensions to the query language, non-SQL or non-IDL statements may not be included. They may be defined interactively or from a 3GL program (using precompilers or the call interface).

Stored commands can take either a named or an ordered list of formal parameters. There are no limitations on the amount of select data that

LISTING 3. *Syntax used by Sybase.*

```
CREATE PROCEDURE [owner.]procedure_name[;number]
    [[(]@parameter_name datatype [= default] [OUTput]
    [, @parameter_name datatype [= default] [OUTput]]...[)]]
    [WITH RECOMPILE}
    AS SQL_statements

[EXECute] [@return_status = ]
    [[[server.]database.]owner.]procedure_name[;number]
    [[@parameter_name =] value |
        [@parameter_name =] @variable [OUTput]
    [,[@parameter_name =] value |
        [@parameter_name =] @variable [OUTput]...]]
    [WITH RECOMPILE]

GRANT EXECUTE ON procedure_name TO {PUBLIC | name_list}

DROP PROCedure [[database.]owner.]procedure_name
    [, [[database.]owner.]procedure_name] ...
```

NOTES

1. The ";number" option allows grouped procedures, which can be dropped by a single DROP PROCEDURE statement. Procedures within the group can then not be dropped individually.

2. The "default" option allows the specification of a default value for parameters not supplied at runtime. They must be a constant, but can include wildcards if the parameter is used with LIKE.

3. The "OUTput" option shows that the parameter is a return parameter. This provides a call-by-reference capability so variables can be modified by the procedure.

4. The "WITH RECOMPILE" option on CREATE PROCEDURE means that SQL Server will never save a plan for this procedure, but will create a new one each time the procedure is invoked. When used with EXECUTE, "WITH RECOMPILE" forces compilation of a new plan for use during that invocation only.

can be returned; however, the statements in a stored command either process normally or the stored command fails. Data type conversion is automatic, although some restrictions apply.

When a stored command is defined, a parameterized query tree is produced and stored in the dictionary. On subsequent invocations, no parsing is required and all objects referenced by the stored command are able to be solved into internal name-independent references.

A query plan (and other processing options) may be stored with the command, further reducing overhead accrued due to selection of a query plan. A dependency list is maintained showing what stored commands and views depend on what database objects. If an attempt is made to drop an object, all dependent objects must be dropped first and then subsequently recompiled. Stored commands may be shared by multiple applications.

Any statement that creates an object such as a table, view, or stored command is not allowed. Any other statement is allowed. The number of parameters allowed and the maximum amount of text that can be used to define a stored command are determined by settable parameters. The syntax used by Sharebase is shown in Listing 2.

Sybase. Sybase provides what they call stored procedures. Sybase's stored procedures can contain any number of "Transact-SQL" statements except for those that create certain objects (described in a moment). Since Transact-SQL provides for procedural extensions to SQL, this is an appropriate name. Stored procedures may be defined interactively or through the 3GL call interface. They can take formal parameters at execution time, either as an ordered or a named list.

When a stored procedure is first executed, it is compiled down to a query plan, which is stored in the database. Automatic recompilation will occur at invocation if any object on which the stored procedure depends has been altered since its definition. The query processor parses the statements and produces optimized query plans. These are stored in the database and may be shared by all users. Execution of stored procedures can be controlled by an extension to the SQL GRANT command.

By issuing an UPDATE STATISTICS command, access plans for all stored procedures are automatically reevaluated. The Sybase query optimizer is sensitive to statistics.

Stored procedures can reference objects in multiple databases and on remote servers. The procedures can also be nested. SQL statements allowed in stored procedures include control of flow language such as DECLARE, WHILE, BREAK, and GOTO; CREATE VIEW, DEFAULT, RULE, TRIGGER, and USE or CREATE PROCEDURE are not allowed. Multiple conditional returns from the procedure are supported.

The maximum parameters allowed in a Sybase procedure is 255. Since variables used as stored procedure parameters are not part of the database, rolling back a transaction will not roll back these variables. This means that transaction management in the 3GL and 4GL code must be written so as to rollback these variables as well. While this problem is not specific to stored procedures, it is a common source of problems in relational applications.

The maximum text in a stored procedure is 65,280 characters. You can rename a procedure. The procedure definition is stored in the system catalog and can be displayed at any time. The query plan for a three to four statement procedure will use two to four kilobytes (kB) of cache memory. The syntax Sybase uses is shown in Listing 3.

Ingres. Ingres v. 6.1 supports "database procedures" (to emphasize that they are stored in the database). Like Sybase, Ingres database procedures support procedural extensions to SQL. When a database procedure is first invoked, the Ingres database engine looks up the definition in the catalog, compiles it, and produces a compiled query plan. Database procedures take a named list of formal parameters. Ingres automatically recompiles the database procedure if objects on which the procedure depends change or if an UPDATE STATISTICS is executed.

Database procedures may be shared by applications. They can be invoked interactively, through Applications by Forms (ABF) or embedded SQL. Execute permission is distinct from permission on the objects on which the database procedure depends.

The Ingres optimizer is sensitive to statistics like the selectivity of indexes and to distribution of data values. Thus, the UPDATE STATISTICS command can affect the query plan.

Ingres database procedures look identical to procedures defined in the ABF 4GL product. This lets ABF procedures to be replaced by database procedures transparently to the application. They are a natural extension to an idea long available in Ingres, that of a "repeated" query. Specifying a query as repeated informs the optimizer to save the query plan for repeated execution during the life of the application. Database procedures add procedural language enhancements, parameters, error returns, and longer term storage of the query plan.

Database procedures can handle a maximum of 127 parameters in the current release and will be able to handle 300 parameters in the next release. There is no practical limit on the amount of text that defines a procedure and which is stored formatted in the database. Any datatype and legal SQL statement can be used. Each procedure uses about one kB of fixed overhead and typically one to two kB for each SQL statement. Thus, a three- to four-statement procedure might require five to nine kB when compiled into an access plan. The maximum size access plan supported is determined by a user-adjustable parameter.

Presently, database procedures can't declare a cursor. Hence, multiple row select results cannot be returned from a database procedure. In keeping with the syntax of embedded SQL, Ingres will solve this problem by allowing a database procedure to declare a cursor on behalf of the front end. ■

— David McGoveran

LISTING 4. *Syntax used by Ingres.*

```
[CREATE] PROCEDURE procedure_name
    [(parameter_name[=]datatype{,parameter_name[=]datatype})]
    =AS
    [DECLARE
        variable{,variable}[=]datatype;
        {variable{,variable}[=]datatype;}
BEGIN
    SQL_statements
END;
GRANT EXECUTE ON PROCEDURE procedure_name TO user_list;
EXEC SQL EXECUTE PROCEDURE procedure_name
    [(parameter_name=value{, parameter_name=value})][INTO return_status];
DROP PROCEDURE proc_name
```