Craig S. Mullins

Return to Home Page

October 1998

From IDUG Solutions Journal...

The Future of SQL

By Craig S. Mullins

I believe the future of SQL is bright; it is the present of SQL that I am worried about — but first, some background.

Structured Query Language, better known as SQL (and pronounced "sequel" or "ess-cue-el"), is the *de facto* standard query language for relational database management systems (RDBMSs). SQL is used not just by DB2, but also by the other leading RDBMS products such as Oracle and SQL Server. Indeed, every relational database management system — and many non-relational DBMS products — support SQL as the method for accessing data.

Why is SQL So Successful?

Why is SQL pervasive within the realm of relational data access? What benefits are accrued by using SQL rather than another, more computationally complete language?

There are many reasons for SQL's success. Foremost is that SQL is a high-level language that provides a greater degree of abstraction than do procedural languages. Third-generation languages (3GLs), such as COBOL and C, and even fourth-generation languages

(4GLs), require that the programmer navigate data structures. Program logic must be coded to proceed record-by-record through the data stores in an order determined by the application programmer or systems analyst. This information is encoded in the high-level language and is difficult to change after it has been programmed.

SQL, on the other hand, is designed to allow the programmer to specify what data is needed. It does not, indeed it cannot, specify how to retrieve it. SQL is coded without embedded data-navigational instructions. The DBMS analyzes the SQL and formulates data-navigational instructions "behind the scenes." These data-navigational instructions are called access paths. By forcing the DBMS to determine the optimal access path to the data, a heavy burden is removed from the programmer. In addition, the database can have a better understanding of the state of the data it stores, and thereby can produce a more efficient and dynamic access path to the data. The result is that SQL, used properly, provides a quicker application development and prototyping environment than is available with corresponding high-level languages.

Another feature of SQL is that it is not merely a query language. The same language used to query data is used also to define data structures, control access to the data, and insert, modify, and delete occurrences of the data. This consolidation of functions into a single language eases communication between different types of users. DBAs, systems programmers, application programmers, systems analysts, systems designers, and end users all speak a common language — namely, SQL. When all the participants in a project are speaking the same language, a synergy is created that can reduce overall system-development time.

Arguably, though, the single most important feature of SQL that has solidified its success is its capability to retrieve data easily using English-like syntax. It is much easier to understand a query such as:

```
SELECT    LASTNAME
FROM      EMP
WHERE     EMPNO = '000010';
```

than it is to understand pages and pages of COBOL, C, or PL/I source code, let alone the archaic instructions of Assembler. Because SQL programming instructions are easier to understand, they are easier also to learn and maintain — thereby making users and programmers more productive in a shorter period of time.

SQL is, by nature, a flexible creature. It uses a free-form structure that gives the user the ability to develop SQL statements in a way best suited to the given user. Each SQL request is parsed by the DBMS before execution to check for proper syntax and to optimize the request. Therefore, SQL statements do not need to start in any given column and can be strung together on one line or broken apart on several lines. For example, the following SQL statement:

```
SELECT LASTNAME FROM EMP WHERE EMPNO = '000010';
```

is equivalent to the SQL statement previously depicted. Another flexible feature of SQL is that a single request can be formulated in a number of different and functionally equivalent ways. Of course, this feature can also be very confusing to SQL novices. Furthermore, the flexibility of SQL is not always desirable because different but equivalent SQL formulations can result in extremely differing performance results.

Finally, one of the greatest benefits derived from using SQL is its ability to operate on sets of data with a single line of code. Multiple rows can be retrieved, modified, or removed in one fell swoop using a single SQL statement. This provides the SQL developer with great power, but this very feature also limits the overall functionality of SQL. Without the ability to loop or step through multiple rows one at a time certain tasks are impossible to accomplish using only SQL. Of course, as more and more functionality is added to SQL, the number of tasks that can be coded using SQL alone is increasing.

The Origins of SQL

The original version of SQL was called SEQUEL (Structured English QUery Language) and it was designed by IBM research in San Jose, California in the early 1970s. The first commercial implementation of SQL was in 1979 by Oracle Corporation (then known as Relational Software, Inc.).

In October 1986, ANSI approved a basic version of SQL as the official standard. Most SQL implementations since have included many non-standard extensions to the ANSI standard. ANSI updated the standard in 1989 to include data integrity enhancements and again, more substantially in 1992 to a new standard commonly known as SQL2. Further enhancements have been made to the SQL standard to include support for a Call Level Interface (CLI) and stored procedures.

The next iteration of the SQL standard, commonly known as SQL3, is in the works and will extend SQL to provide object/relational capabilities. There is no clear consensus as to when SQL3 will be fully agreed upon and delivered.

The Threat of the Present

If you believe some industry pundits reliance on SQL for relational data access is on the wane. New Internet technologies such as Java and XML are being touted as the "next big thing" for accessing databases.

The promise of these new technologies is intriguing. Take XML for example. If you believe everything you read, then XML is going to solve all of our interoperability problems, completely replace SQL, and possibly even deliver peace on Earth. Okay, that last one is an exaggeration, but you get the point. In actuality, XML stands for eXtensible Markup Language. The need for extensibility, structure, and validation is the basis for the evolution of the web towards XML. XML, like HTML, is based upon SGML (Standard Generalized Markup Language) which allows documents to be self-describing, through the specification of tag sets and the structural relationships between the tags. HTML is a small, specifically-defined set of tags and attributes, enabling users to bypass the self-describing aspect for a document. XML, on the other hand, retains the key SGML advantage of self-description, while avoiding the complexity of full-blown SGML.

But XML does not do what SQL does and hence, cannot replace it. And the same can be said for Java. Willie and I discussed Java in this column in the last issue of *IDUG Solutions Journal*, so I will not rehash the subject here again. Suffice it to say, Java cannot and will not replace SQL either. The API to relational databases remains SQL and any other data access or presentation technology necessarily must communicate with relational data by means of SQL. XML and Java can provide benefit to organizations by extending the capability of the Internet to include data access and modification through SQL, and this is goodness. But don't be fooled into believing they (or anything else) will be able to completely

replace SQL any time soon.

Even so, object-oriented programmers tend to resist using SQL. The set-based nature of SQL is not simple to master and is anathema to the OO techniques practiced by Java developers. All too often the manner in which data is accessed is not planned out and designed in a thoughtful manner. In fact, sometimes it is not thought of at all until performance suffers.

Object orientation is indeed a political nightmare for those schooled in relational tenets. Proponents of OO are almost always the enemy of those who practice sound data management policy. All too often organizations are experiencing political struggles between the OO programming team and the data resource management group. The OO crowd espouses programs and components as the center of the universe; the data crowd adheres to the tenets of normalized, shared data with the RDBMS as the center of the universe.

Thanks to all of the hype, the OO crowd tends to win many of these battles, but the war will eventually be won by data-centered thinking. The notion of data normalization and shared databases to reduce redundancy provides too many benefits in the long run to be abandoned. As the focus blurs away from data management and sound relational practices, data quality will deteriorate and productivity will decline. And then a new era of SQL vitality will arise.

The Glow of the Future

SQL is still a growing and very adaptable language. Depending on the version and flavor of DB2 you are using, new functionality such as CASE statements, outer joins, and nested table expressions have increased the number of tasks we can perform using SQL alone. But these features are not the end of how SQL will adapt to

change.

Recursive SQL, available in DB2 Universal Database Server in V5, and soon to be ported to other members of the DB2 family, further expands the functionality of SQL. Recursion enables a table expression to refer to itself. With recursive SQL even more tasks become possible using only SQL without embedding it in a 3GL or 4GL. To help you understand recursion better, think of the following metaphor. Consider what happens when a camera films someone watching himself on live TV. You can see the person multiple times in the picture because it is recursive. It is possible to traverse hierarchies such as a bill of materials or an organization chart using a single recursive query.

Traditionally, DBMS products stored data and nothing else. But all of the major RDBMS products of today support (or are moving to support) procedural logic in the form of triggers, functions, and stored procedures, sometimes referred to as Server Code Objects (or SCOs for short). The ability to store procedural logic in the database is increasingly common because it enhances performance of client/server applications, eases security implementation, promotes reusability, and makes databases active.

DB2 for OS/390 V6 will add features such as triggers, user-defined functions, and more built-in functions that will enable active databases with increased data integrity. An active database can take actions automatically and implicitly by virtue of an event occurring. For example, a database trigger can be specified on a table that automatically calculates derived data whenever any value it is derived from changes. The trigger can even INSERT the derived data into another column automatically, thereby preserving data quality and integrity. And all of this happens just because a single

SQL statement was issued to change data. So even more tasks will be able to be accomplished using only SQL.

DB2 Universal Database, and recently DB2 for OS/390, provide SQL CLI support. The SQL CLI (Call-Level Interface) is an alternative binding style for executing SQL statements. Instead of embedding SQL in an application program, you use routines that allocate and deallocate resources, control connections to the database, execute SQL statements using similar mechanisms to dynamic SQL, obtain diagnostic information, control transaction termination, and obtain information about the implementation. Basically, the CLI issues SQL statements against the database using procedure calls instead of via direct embedded SQL statements. A SQL CLI is useful for enabling SQL access to more languages and programmers than before. This is goodness.

The Long Term Future is Even Brighter

And the long-term future of SQL is even brighter. More and more features will be available using just SQL. Large object support (available in UDB today, in OS/390 in V6) enables SQL statements to access very large text and multimedia objects. As these features become available more and more applications will use them in novel ways to gain a competitive advantage.

Additionally, SQL will eventually be extended to incorporate new and exciting technology such as fuzzy logic and temporal data. These features are further in the future, but promise to provide exciting new capabilities.

The addition of temporal qualities to relational databases, and thus SQL, will make your databases time-sensitive. Temporal extensions will enable you to query the database not just on its current state,

but on its past state as well. For example, temporal capabilities could help you to answer questions when data changes over time, such as:

- The actual price of a particular product on a given date
- The commission rate assigned to a sales rep on a given date
- Monthly revenue changes
- The status of a project at a particular moment in the past
- Inventory at any given point in time
- Sales on the day before Christmas over the past ten years

And these are just a few of the examples. I'm sure you can think of more using your own systems and needs. There are many possible ways to extend SQL to support temporal databases. Here is one possible example of a simple temporal SQL query:

```
SELECT    PROD_ID
FROM      PROD
WHERE     PRICE > 100,00
FROMDATE  '1997-03-01'
TODATE    '1998-09-30';
```

Of course, you will have to assign a time granularity to the PROD table before such a query could be issued. And there are many different time granularities that could be assigned: YEAR, MONTH, WEEK, HOUR, etc. You get the idea!

Fuzzy logic is also a discipline that will eventually find its way into relational database technology and SQL. Fuzzy logic is a discipline that relies on Eastern philosophy more than Western. Its simple theory is that everything is true, just to different degrees. In the Western world we are used to absolutes: YES and NO; TRUE and FALSE; ON and OFF; 1 and 0. Eastern philosophy is less rigid,

teaching that all things are shades of gray, instead of black or white.

Why is fuzzy logic useful? Consider the following:

- When eating an apple, when does it stop being an apple and become an apple core instead?
- If you wish to control room temperature to be 70 degrees F, do you really want to tell the air conditioner to shut off immediately when the temperature is 60.9 and turn back on at 70.1?
- When querying your data looking for the products earning highest amount of revenue, what is the cut off point inclusion? If it is $6 million, do you really want to ignore the products that earned $5.99 million?

Fuzzy logic helps to alleviate these problems by imposing degrees of truth on everything. A bank might want to identify bank accounts that are a drag on earnings. Someone who makes many transactions but maintains a low balance is probably not earning the bank a lot of revenue, so it might wish to levy fees on these accounts. A fuzzy SQL query to help identify these accounts might look something like this:

```
SELECT    NAME, ACCT_NO
FROM      ACCT
WHERE     BALANCE IS LOW
AND       ACTIVITY IS HIGH;
```

Now wouldn't that be simpler than trying to set an arbitrary cut off for what constitutes a low BALANCE or high ACTIVITY? Of course, you will need to set up the fuzzy vocabulary up front. This usually involves defining terms such as LOW and HIGH, FEW and MANY, or YOUNG and OLD. Additional second order terms can be defined

to augment the first order fuzzy terms, such as USUALLY, ALWAYS, and VERY. So, we could augment our query above to add second order terms such as:

```
SELECT    NAME, ACCT_NO
FROM      ACCT
WHERE     BALANCE IS USUALLY LOW
AND       ACTIVITY IS VERY HIGH;
```

In general, fuzzy logic can help to improve the results of SQL and will enable us to do even more with a single SQL statement.

Synopsis

The future of SQL is indeed bright, but challenges lurk around every corner. What should you do as OO, Java, and XML encroach on the world of SQL and threaten data integrity? Well, use common sense. Understand any new technology before implementing it at your company. Know what it can and can't do by practicing with the technology — not by listening to the hype. Be sure to maintain and develop expertise in SQL in your organization. Be sure to know how the new technologies interact with SQL (JDBC, JSQL) and implement them appropriately. And keep on promoting SQL and data resource management. I mean, deep down inside we know these must be the basis for our IT infrastructure or that infrastructure will crumble around us.

From IDUG Solutions Journal, October 1998.

© 1999 Mullins Consulting, Inc. All rights reserved.
Home. Phone: 281-494-6153 Fax: 281-491-0637