Distributed database for SAA

by R. Reinsch

This paper describes, in general terms, distributed database and its relationship to Systems Application Architecture (SAA). It shows the importance to effective distribution of IBM's Structured Query Language (SQL), the database element of the Systems Application Architecture Common Programming Interface (SAA CPI). The paper defines five levels of distribution, showing how each fits real-world application requirements. Finally, it outlines the magnitude of the task.

The success of IBM's entry into distributed database depends heavily on the success of Structured Query Language (SQL) in providing the database element of the Common Programming Interface for Systems Application Architecture (SAA). In addition to being a programming interface, SQL is used directly by many end users through query products such as the Query Management Facility (QMF). As workload and database data are distributed to multiple large mainframe computers, departmental systems, and workstations, it is essential that users of these systems be protected from the culture shock such changes could cause. Consistency of SQL language statements and the results they produce at execution is crucial.

This paper describes, in general terms, what a distributed database is. It shows how a distributed database is different from distributed files and general distributed processing. Five distinct levels of distributed database handling are defined, with a discussion of the classes of applications that can be handled at each level. The argument is presented that there are natural states of technology which also suggest these divisions.

This paper does not present theorems, proofs, rules, or criteria for truly distributed Database Management Systems (DBMSs). Rather, general concepts are presented to attempt to bridge the gap between the tasks at hand and the technologies available to accomplish them.

The reader will see that in the distributed database world, change is both the greatest strength and the greatest challenge to be faced. To succeed, change must be allowed, even encouraged. But control must be maintained in the process. Most companies which choose to invest in DBMSs do so to gain better control over their corporate data resource. They need better security, better concurrency, better recoverability, and so on. Control must be maintained while new and powerful workstations and departmental systems are being incorporated into a company's overall computing strategy. At the same time, the data processing community within most companies must grow more flexible in forming itself to the natural organization of the company it serves. Distributed databases are a natural result. It's almost impossible to avoid them. They certainly provide a better fit in most cases.

Most companies are not in the database management business or even in the data processing business. They design things, build things, sell things,

arrange things, or transport things. They *use* computers and databases as tools to do their real job. Distributed databases with SAA SQL can provide a powerful means for making tasks easier, for making the Database Management System (DBMS) a better *silent* partner, while at the same time supporting those changes which *are* required to provide constantly improving service and productivity for the real job.

Traditional DBMSs operate in a single environment.

What is a distributed DBMS?

Let's break down this question. What is a database? A database is a collection of information (data) which is stored and organized so that people can add things, look at things, and change things in an efficient manner. In general, the more data available and the more people who have to work with it at the same time, the more attention is given as to how it is stored and organized. This is particularly important when updates are involved.

Once the volume of data and/or the number of users reaches a certain point, things start to go out of control. Data are lost or destroyed, or people can't get their jobs done. Enter the DBMS. The Database Management System adds support to the raw database. DBMSs typically provide concurrency (locking) controls to prevent users from trampling one another's changes, security controls to prevent unauthorized users from accessing or changing the wrong information, recovery procedures to protect against "accidents," etc. DBMSs also provide transaction scheduling support, accounting information, diagnostic and servicing information when needed. Relational DBMSs also take control of determining where the data physically reside and deciding the best method of transferring data to the user.

Traditional DBMSs operate in a single environment comprising the *computer* with all the programs and data, and the *users* who access the *computer* to do their jobs. The DBMSs have been centralized to reduce the cost of sharing corporate data. Statements such as "Computers are expensive," "You can't waste a second of computer time," and "We have to fill all our disks" are simply no longer true. Users and data processing managers alike are seeing the changes in the economics of computers. They no longer are so expensive that the financial side of the company demands major justification. If it fits on the desk, buy it. Often the end result is a lot of little databases which are clearly distributed but not part of a cohesive distributed DBMS.

For the purposes of this paper, then, let's agree that for a DBMS to be considered distributed, it must still be a DBMS (with the emphasis on management system), with some of the data managed located on a different computer, or the application on another computer, or both. How the user got into this distributed situation and how permanent he or she expects the condition to remain are the factors that really dictate the level of distributed DBMS support needed to do the job.

General distributed processing would allow an arbitrary division of the parts of the application into different machines. With distributed DBMS, the line is drawn *exactly at the Application Programming Interface (API) between the application and the DBMS.*

The remainder of this paper describes five different levels of distributed DBMS support, each of which is suitable for a different application environment.

User-assisted distribution

In the first stage of distributed systems, the user is completely aware of the distribution process. In fact, for each distribution activity, the users are involved twice.

For this stage, the user interacts with one system to extract the needed data. He or she then physically takes the data to the system which is to receive the information. The user then initiates a process on the receiving system to load the extracted data.

This does not sound very elegant. However, if this is a relatively rare event, it may be completely adequate. Across some organizational boundaries, this may be the best way to get the job done. In fact, this is exactly the process we follow when we install

software on our systems. One user creates an unloaded version of some information we want. We buy it and load it into our system.

One of the interesting questions to consider in this situation is why the data are being moved in the first place. Is this a one-shot installation process? Did a user move between systems? Did he or she bring some programs along for use at the new site? Did he or she bring data too? Did the user just need a frozen copy of data at a particular point in time?

In the world of remote requests, life is easier for the end user.

If both systems are of the same type and are running the same operating system and DBMS, moving the programs, canned queries, etc. to the new system and putting them to work should be relatively simple. This is usually the case for packages you buy (or you'd buy something else).

However, if the systems are different, many changes to the program may be needed just to get it to run. And if you want to get exactly the same results that you got on the first machine, considerable reworking of the queries and surrounding logic might be required to get the desired (consistent) answer.

What is fundamental at this stage is that the receiver must know how to handle what has been sent. A special program could be used to handle each different set of information. In many cases, however, it will be better to send a description of the information along as part of the information sent.

With the Systems Application Architecture database language SQL, the queries should run without modification and produce consistent results.[1] When these SQL statements are part of a program written in one of the SAA languages, such as COBOL, the whole program should move easily.

By minimizing the effort to convert programs, this level of distribution can be put to use more often to solve data processing or information center problems. At this level, the fact that the SQLs and COBOLs match is a convenience and a cost saver. The user is aware of the process and can intervene where necessary.

Whether files, programs, or database data are being distributed at this stage, the general processing flow is the same. In the next degree of distributed processing, the user is a little more isolated from what is actually going on and has less opportunity to adjust the process to account for differences between environments.

Remote requests

In the world of remote requests, life is easier for the end user. Instead of interacting with two different systems at two different times to get data uploaded or downloaded, he or she interacts with one system once. At this stage, communications technologies are used directly by the systems to accomplish a user's task. Here's what happens. Generally the user establishes a connection between his or her system and the system which has the information he or she needs. This connection may be established automatically, but it is more likely that the user will invoke a procedure to log on to the remote system and prepare it to receive requests from his or her system. The user does this once, regardless of the number of upload and download requests he or she will make.

When the user needs data, he or she interacts with an application running on the local machine to start the required operation. (This high-level request, such as an extract query, can be edited immediately before execution.) The application at the user's machine composes the message which represents the user's request. That application then sends the message to the other machine, along with any data required to satisfy the request (e.g., for upload). The application at the DBMS's system receives the request and begins the processing necessary to do the job.

For a database extract, the application on the remote system will perform dynamic SQL operations to read all the rows of the answer and buffer them up. When the whole answer set has been collected, the application closes the cursor and terminates processing with the DBMS. Finally, the buffered answer and status are transmitted back to the user's system. The user's application will then put the answer where the user requested.

The Virtual SQL (VSQL) processor in the Enhanced Connectivity Facility (ECF) product[2,3] performs SQL operations against DBMSs using this level of distributed database. The application on the user's system is called the VSQL requestor and the application on the remote DBMS system is the VSQL server.

In this environment, the DBMS is really unaware that distribution is happening, just as it was in the previous case. The general rule is this: *NO* communication goes on while *any* DBMS resources are held. This allows use of communications facilities which might not notify the DBMS when failures occur without jeopardizing availability of the data managed by the DBMS. All the failures which matter to the DBMS are local and the operating system ensures that the DBMS is informed.

It is important that the SQL statement behave consistently, regardless of the target system.

In this environment, SQL statements (in VSQL queries) are sent to the remote system to which the *user* is connected. There is very little opportunity for a system administrator to intervene in this process; therefore, it is important that the SQL statement behave consistently, regardless of the target system. The user knows which system contains the data he needs. He can also have duplicate (but slightly different) queries for each system to accommodate language differences, though this is not desirable from his point of view.

In all the cases discussed so far, the distributed data are copies of originals. Both the original version and the copy have lives of their own, and will diverge over time as updates are made. When this situation becomes intolerable, the next level of distributed database handling is required.

Remote unit of work

With remote unit of work, an application program executes on one system and uses the remote API (Application Program Interface) provided by another system. For distributed database, this means that any DBMS facilities available to local applications are available to applications running on remote systems.

With the SQL language, all requests which change the database are tentative until committed by the application program. While the application is executing, database resources, i.e., data, are protected with locks from interference by other users. While update locks are held, no other program is allowed to access the data. While read locks are held, no other program is allowed to change the data. An application in a workstation could gain read locks over large quantities of data by executing simple queries. These locks could interfere with those required by others, and they would not be released until the *application* released them. The SQL application program controls boundaries of the active unit of work or transaction through the use of COMMIT and ROLLBACK requests.

This is essentially sharing in real time, as it was before any distribution was introduced; local users and remote users share the data as if they were all local. Application programs must do their job and then release the locks, which identifies a key requirement at this level, timely and reliable failure notifications.

Users have all encountered programs that do not execute properly. In particular, programs fail in ways which prevent them from doing a complete job, including releasing locks. For local applications, the operating systems provide notification of such application failures. The DBMS then rolls back the incomplete work and releases the locks held by the failed application.

In the distributed case, the operating system which sees the failure is remote to the DBMS. A mechanism for timely failure notification is a major new requirement as we approach remote API. SNA's Advanced Program-to-Program Communication (APPC) and Logical Unit Type 6.2 provide a connection architecture which gives the timely notification required to allow remote applications to hold locks on database resources between requests without jeopardizing availability of the data for the rest of the users.

From the SAA perspective, remote unit of work means that a program written, compiled, and executing in one environment will be using an API provided by another system. Operating System/2™ Extended Edition (OS/2™ EE) applications running on

Personal System/2® (PS/2®) machines will be accessing IBM Database 2 (DB2) data using DB2's SQL API on MVS on a System/370. So long as the application accesses only one DBMS at a time, it can adjust its logic to accommodate differences between DBMS environments. However, it is likely that the very same program may access local OS/2 data on its next execution, SQL/DS data on the one after that, and AS/400™ data on the next. For this application to be successful, the APIs presented by all of these systems must be the same. SAA SQL, as the database element of the Common Programming Interface, provides that common API.

SAA SQL, as the database element of the Common Programming Interface, provides the common API.

As long as the data remain in one location and not too many locations are involved in any given set of processing, remote unit of work processing is probably enough. However, if data start moving on a regular basis or coordinated updates are required at more than one location, the next level of distributed database is required.

Distributed unit of work

There are two key extensions provided by distributed unit of work or distributed transaction processing over remote unit of work. First, the DBMS knows or finds out which system manages the data to be read or changed by each request. Second, the DBMS coordinates updates at several locations in a single transaction. Within the scope of one transaction or unit of work, coordinated updates can be made to the database on the mainframe *and* the database on a workstation.

One of the application areas where this is especially useful is data gathering from workstations into host systems. Records can be moved from the workstation to the host without fear of loss or duplication. Related updates at multiple locations can also be performed as part of a single transaction. As an example, consider removing an item from inventory at one warehouse's DBMS, showing the item in-transit at another's, and adjusting the financial records for both warehouses at a central accounting location as three related requests. As demanded by the semantics of transactions, no partial updates are allowed; either all of the related changes will be committed into the database(s), or none of them will.

From an architectural perspective, two-phase commit processing is required over the network of participating DBMS locations. Each must have a say in whether commitment of the transaction is possible or whether the operations must be rolled back. SNA's LU 6.2 provides the architecture to allow this to occur.

From a practical perspective, the various warehouse and accounting systems may have been in existence prior to the introduction of the transaction to perform the coordinated update. In this case, aggregation of previously separate functions into a single, apparently collected, integrated database is performed. Another case involves a centralized system which has outgrown its machine, or one which is being decentralized to allow more local control at the remote locations. In this case, the old transaction programs which worked against local databases continue to operate, even though the data have been distributed from their original location.

In both of these cases, it is very important that the SQL language be the same in all of these systems. If the statements cease to execute when the data move, or worse yet, they execute but produce different answers, the application programmer must constantly test and adjust his programs to accommodate changes in the environment. This is likely to become too expensive or impossible to manage—so performance on the *real* job (e.g., moving goods) suffers.

With SAA SQL as a language base, and the IBM relational DBMS products providing compiled as well as dynamic SQL, a simple installation step is all that is required to update the system. An operation called BIND is performed; no application parameters are required. The DBMS examines the statements in the application, determines the location of all data being accessed by the application, ensures that each location involved computes the optimum access algorithm to perform the operations required at that location, and prepares to coordinate execution of the transaction when requested. The whole process is

automatic, and the user is unaware of the distribution which happens to satisfy his requests.

Applications are not sensitive to the true location of data or how data are supported by performance assists such as indexes. At this level of distributed support, the application only has to be sure that the data referenced by any single statement reside at one location.

Some application statements may fail when data are moved from one location to another because the single-site-per-statement restriction is not honored. If this is a rare event, the application can be adjusted to avoid the problem, or selected data can be duplicated. However, if it occurs frequently or the application cannot be made flexible enough at reasonable expense, the next level of distributed database processing is required.

Distributed request

In the distributed request environment, all data location restrictions are removed. Within a single SQL statement, relational data from many locations can be combined to produce the desired result. Whatever would be possible with all data local is now possible in a distributed environment as well. The distributed database looks like one very large single DBMS.

However, the user still must consider the reality of the situation; there is no magic. If it would have taken a long time to perform the operation locally, it is still likely to take a long time in a distributed environment. However, there are some possibilities for improved performance. For example, if local data at a workstation are made remote on a big System/370, the results can come back to the application faster; there is more CPU power available, faster DASD, etc. However, there are also opportunities for slower response. When communication channels are introduced between processing steps, delays are inevitable. Depending on the bandwidth of the channel being used and the amount of data which must flow, this may or may not be significant for the whole job. Consider the following extreme example:

SELECT COUNT(*) FROM A,B where Col_from_A = Col_from_B.

The answer in this case will be only a few characters, regardless of the size of tables A and B. The SQL statement itself is under 60 characters. Neither of these sizes will significantly affect the time it takes to get an answer. However, if Tables A and B are on different systems, the bandwidth of the path between them is crucial. If one of the systems is slower than the other, this could negatively affect the time required. Even if all the data are on one system, the speed of that system will make a difference. Two PS/2 systems connected over a typically fast local area network may be able to outperform two System/370s connected over voice-grade communication facilities. The SQL DBMS optimizers can take all this variability into account as they compute the best (quickest) method to produce the desired result and have it delivered to the application which requested it. This represents a significant leap in technology in the DBMSs.

Obviously, this optimization would be impossible if each of the SQL DBMSs spoke a different dialect of SQL with different semantics and produced results different from the others. To produce a result the user can trust, each of the DBMSs must be directly substitutable for the others in the sequence of operations required to produce the required answer. It cannot matter which order the optimizer picks or how it divides the work between the DBMSs to get the job done.

SAA SQL provides an effective methodology for consistent semantics and success in this environment.

Summary

This paper has described the distributed relational database environment by beginning with a basic nondistributed environment and then discussing levels of data distribution which can be built on that base. Each of the five levels of distributed processing builds on the knowledge and technology of the previous level; new technologies are required to make each step along the way. Finally, examples have been presented of application circumstances which depended on each of the levels of distributed database availability.

Some products are already available which provide distributed services. A sampler follows.

In April 1984, IBM announced DXT. That product, in combination with the load facilities on DB2 and SQL/DS, provided support for the first level of distributed DBMS data. Later announcements expanded the types of data covered. There are many other products which depend on or support this level of distribution.

In September 1986, IBM announced ECF as a program which used the Server and Requestor Processing

Interfaces in PCs and MVS and VM to perform the second level of support for distributed database data. Later announcements have expanded the environments covered and data formats supported. Initial ECF support included remote request processing using the virtual SQL requestor (VSQL) against DB2 and SQL/DS data, making these data available to applications running on DOS and OS/2 systems.

In October 1987, IBM announced SQL/DS with Remote Relational Access Support. This is the first taste of remote unit of work support for database; it allows a cluster of SQL/DSs to operate with applications and DBMSs on different physical machines.

In a distributed environment, portability is the rule rather than the exception.

Announcements will continue as Systems Application Architecture is enhanced and as the relational database products implement those enhanced architectural elements.

In this paper, the author has tried to show the magnitude of the problems at hand and provide some clues as to the likely sequence of events for distributed database processing. Space constraints have made it impossible even to touch on some of the real challenges facing implementers of distributed DBMSs and those who would use them. There are issues related to the naming of users and database objects, providing effective diagnostic information and tools, providing distributed data administration tools and facilities, managing copies of data as either point-in-time snapshots or replicates or fragments of databases, providing adequate security mechanisms for the very secure while not burdening those who are less concerned, etc. Even at the conceptual level chosen for this paper, each of these topics in its own right could support a separate paper. Each of these areas, and others as well, must be addressed before actual implementation can take place.

This paper was intended to provide insight into distributed database requirements, to help the reader see the forest, a little, in a technological area which could look like just so many trees—or even just leaves blowing in the wind. This writer has attempted, without resorting to rules, to identify key characteristics of the problems to be addressed with distributed database systems in a way that can be directly applied to real user environments. With the level of understanding offered here, the writer hopes that the reader will be able to analyze his own situation from a more practical perspective and see where distributed database processing may help him solve some of his application system problems.

Conclusion

The Systems Application Architecture database language SQL provides an excellent base for portable applications. In a distributed environment, portability is the rule rather than the exception: No application can ever escape cooperating with both older and newer versions of itself. Some of these contacts will be overt and planned. Some will be unplanned and almost accidental. Almost all of them involve intersections of interesting information which must be *shared* to realize maximum return on investment. SAA SQL is a key ingredient in achieving success in this environment.

Cited references and note

1. Fundamental differences in machine architectures might show in the final result. For example, the difference between bit encodings of ECBDIC and ASCII machines produces different orders for alphabetic sorts. Also, System/370 floating-point numbers can be much larger (or smaller) than IEEE floating-point numbers; this might show through.
2. *Introduction to IBM System/370 to IBM Personal Computer Enhanced Connectivity Facilities*, GC23-0957, IBM Corporation; available through IBM branch offices.
3. *IBM Enhanced Connectivity Facility PC Requester's Reference*, SK2T-0002, IBM Corporation; available through IBM branch offices.

General references

Systems Application Architecture Common Programming Interface Data Base Reference, SC26-4348, IBM Corporation; available through IBM branch offices.

D. E. Wolford, "Application enabling in SAA," *IBM Systems Journal* **27**, No. 3, 301–305 (1988, this issue).

R. Williams, D. Daniels, L. Haas, G. Lapis, B. G. Lindsay, P. Ng, R. Obermarck, P. Selinger, A. Walker, P. Wilms, and R. Yost, "R*: An overview of the architecture," *Proceedings of the 2nd*

International Conference on Databases: Improving Usability and Responsiveness, Jerusalem, Israel, June 1982. Published in Improving Usability and Responsiveness, P. Scheuermann, Ed., Academic Press, New York, pp. 1–27.

D. Daniels, P. Selinger, L. Haas, B. Lindsay, C. Mohan, A. Walker, and P. Wilms, "An introduction to distributed query compilation in R*," *Proceedings of the 2nd International Symposium on Distributed Databases*, Berlin, September 1982.

L. M. Haas, P. Selinger, E. Bertino, D. Daniels, B. Lindsay, G. Lohman, Y. Masunaga, C. Mohan, P. Ng, P. Wilms, and R. Yost "R*: A research project on distributed relational DBMS," *IEEE Database Engineering* **5**, No. 4, 28–32 (December 1982).

P. F. Wilms, B. G. Lindsay, and P. G. Selinger "Distributed execution protocols for data definition in R*," *Proceedings of SIGMOD 83*, May 1983.

Roger A. Reinsch *IBM Programming Systems Division, 555 Bailey Avenue, San Jose, California 95141.* Mr. Reinsch is a senior technical staff member at IBM's Santa Teresa Laboratory. He joined the Data Processing Division in 1967 at the Chicago Distribution Branch Office on the Sears Selected National Account team. He was the communications expert of the team and worked with message switching, data collection, and transaction-oriented systems. In 1974, Mr. Reinsch joined a design group in the Systems Development Division in Palo Alto, California. He later had planning and development responsibilities for interactive products and distributed data. In 1978, he joined the DB2 development group as a team leader of the boundary functions between DB2 and its application environments. For this work, he earned an Outstanding Innovation Award and a First Patent Award. In 1981, Mr. Reinsch became a manager of development in DB2, and subsequently managed planning, performance, design, and technical advisory activities for the DB2 organization. In 1986, Mr. Reinsch gave up his management responsibilities to devote full time to the technical aspects of distributed database.

Reprint Order No. G321-5331.