4-06-45 Distributed Processing for Distributed Data Bases

Michael D. Krasowski

Payoff

Distributed data management promotes the sharing and integration of data between corporate departments and systems and the access to that data in heterogeneous systems environments—an arrangement that reflects corporate structure and business requirements. This article explores some of the business drivers that influence the design of distributed data base technology, and identifies and discusses the 13 functional requirements—or rules—that are the foundation of distributed data base management systems (DDBMSs).

Introduction

Information architecture (IA)—the corporate systems infrastructure—can involve networking and telecommunications, standards, and the applications life cycle, as well as data management (e.g., data modeling methodologies, standards, tools, data administration policies, and data base management systems [DBMSs]).

Data distribution, which evolves naturally in business environments, is not always unwelcome. Data that is physically dispersed but centrally managed is the essence of distributed data base technology; data that is physically dispersed without central control can cause problems.

A recent Arthur D. Little study of senior executives found that 94% of the surveyed companies are hierarchical, having strict rules on how information is accessed. These executives also say that their companies soon will have fewer layers of management, will be more responsive to changing business conditions, and will empower their employees by giving them total responsibility, accountability, and authority over the processes assigned to their work groups. These companies will also be more responsive to their internal and external customers, and will eliminate layers of middle management.

Because they will be information-based, these companies must make their data easily accessible to and manageable by each work group. Because data is viewed as a corporate asset, shareable by other work groups, it must be accessible to all. Distributed data base technology supports autonomy while providing the ability to share the data.

Legacy Data

Legacy data is stored in DBMSs but is not easily shared, and legacy data bases are usually built without consideration for the company’s future business requirements. They are generally developed as point solutions that consist of specific hardware, operating systems, and DBMS software (e.g., engineering applications independent of business applications).

Traditionally, MIS departments relied on a single vendor. Now, many applications need multivendor solutions, and many installations include multiple hardware vendors, operating systems, and DBMSs. Although the distribution of data is acceptable, the inability to share it is unacceptable. Distributed data base technology allows data to be shared and gives autonomy to the users who most need the data.
Incremental Growth

Although large mainframes can cost $150,000 to $175,000 per Millions of Instructions Per Second, quite powerful workstations are available for $5,000 per MIPS, or less than 5% of the cost per mainframe MIPS. Although workstation performance cannot be equated simply with that of mainframes, workstations present an opportunity for work groups to manage their own data.

Adding computing resources has generally involved upgrading to a large mainframe or adding a new mainframe and spreading the costs to all users of the service. Most in-house chargeback algorithms, however, do not fairly distribute the costs of the service to the departments that most use it. Incremental additions to the workstation hardware environment support not only a flexible IA environment but the financial philosophy that users should be directly responsible for the costs they incur in managing their data. Centralized, monolithic data bases on expensive mainframes inhibit incremental growth; distributed data base technology permits incremental hardware growth at reasonable cost.

Reliability and Availability

Performance can be improved by using DDBMSs because data can be stored locally on the workstation or server on a local area network (LAN) without compromising the ability to share data. Workstations attached to LANs that manage data through a DBMS are called data base servers. Terminals or workstations must communicate with the mainframe through Wide Area Network (WANs) or broader backbone networks, limiting the effects of slower response times and higher communications costs.

Distributed data base technology allows data to be replicated in an orderly way. Users can then access copies of data in the event of Central Processing Unit failure, network failure, or virtually any other system problem. Distributed data base technology improves performance because the data is closer to the work group, which does not depend on a single, central site.

The Rules of Distribution

A DDBMS should support 12 functional requirements (i.e., rules) and a zero rule. These rules are a subset of the system's total requirements; MIS managers must still quantify the importance of distributed technology to their companies. Distributed data bases do not necessarily imply distributed processing, although DDBMSs may support distributed processing.

Each DDBMS-connected computer is an autonomous DBMS, and its users can access any data in the network as if it were stored at their site, although the data is physically stored in distinct DBMSs at various sites (see Exhibit 1). The 13 rules are discussed in the following sections; the examples cited are based on the relational model. (Rules 1 through 12, which are subsidiary rules that refine the functional capabilities required by rule 9, affect each other and should not be viewed as having equal importance; some may be more or less important than others, depending on an individual company's IA.)

DDBMS Model

Rule 0.
Rule 0 is a system as it appears to the user.
Rule 1: Local Autonomy.
Local data is locally owned, managed, and controlled; each site in the network is autonomous. Local operations remain local, and the fact that the site is participating in a DDBMS should not affect the applications that ran successfully before this participation was implemented. This rule helps preserve the investment in legacy systems and data while allowing interconnection with other systems and incremental capacity growth at each site.

Rule 2: Independence from Central Site.
Implied by rule 1, this rule helps avoid bottlenecks and weaknesses at central sites. If all users rely on a central site that fails, all users are affected. Therefore, no single site should have the central dictionary, central query processing, central concurrency control, central recovery control, nor should it store all the data. Other sites that participate in the DDBMS may have copies of the data needed (see Rule 6).

Rule 3: Continuous Operation.
The DDBMS should operate continuously and should not have any planned shutdowns. It should also allow for the addition or removal of sites from the distributed data base network without affecting service at other sites or of existing applications. Relations (i.e., tables), pieces of relations (i.e., fragments), or copies of relations (i.e., replicas) should be capable of being created or removed without disrupting processing at other sites. If a new release of the DDBMS software is installed at one or more sites (but not at all sites), there should be no disruption of service.

Rule 4: Location Independence.
Users of distributed data bases should never have to know where data is physically located. The data they are using may be at their local site, at remote sites, or both. Applications logic can be simplified in that the programmer does not have to specify at which site the data is located, reducing application code. This information should be kept in a catalog or data dictionary to be referenced by the programs. The data or tables should be movable, however, from one location or site to another (i.e., data migration) for performance reasons without any effect on application logic.

The data dictionary of the DDBMS must be able to map local names (e.g., EMPNUM) to global names (e.g., EMPLOYEE-NUM at New York but created at Los Angeles). If the EMPNUM table is moved, the data dictionary must be updated to reflect the new location; the data dictionary subsystem itself must be distributed and consistent at all sites, and the global name must never change.

Rule 5: Fragmentation Independence.
Because a table or relation in a DDBMS environment is a single, logical object, it may be necessary for performance reasons to organize a table into multiple, physical subtables. Although the fragments are totally unknown to the user, the data base administrator must be aware of the fragments, define them, and allocate them to the appropriate node. The following is an example of a Structured Query Language fragment definition in which customer accounts must be physically split for each branch:

```
CREATE TABLE ACCOUNTS
ACCOUNTS#, BRANCH, CUSTOMER, BALANCE, TYPE)
CREATE FRAGMENT SF1
AS SELECT ACCOUNT#, BRANCH, CUSTOMER, BALANCE, TYPE
FROM ACCOUNTS WHERE BRANCH-SF
```
CREATE FRAGMENT LA1
AS SELECT ACCOUNT#, BRANCH, CUSTOMER, BALANCE, TYPE
FROM ACCOUNTS
WHERE BRANCH=LA

The data base administrator must also be able to define the location of the fragment or
table to the global catalog; that portion of the global catalog is often called the file or table
allocation catalog—for example:

LOCATE FRAGMENT LA1 AT CPU12
LOCATE FRAGMENT SF1 AT CPU65

Users see only the table ACCOUNTS and should not have access to any fragment name
or knowledge about the physical location of the fragments. Data base administrators have
access to and control of the fragment and allocation portion of the catalog.

The DDBMS software can reconstruct the fragments by using the relational operators
UNION and JOIN. The example in Exhibit 2 is called a horizontal fragment because it was
split on a tuple (i.e., row) boundary. Fragments SF1 and LA1 can be recombined by using
the UNION operator: SF1 LA1 is equivalent to ACCOUNTS. A table split on an attribute (i.e.,
column) boundary is called a vertical fragment and can be recombined by using the JOIN
operator.

Replicas

Vendor support of fragments involves many difficult implementation issues. For
example, if a user updates the table ACCOUNTS, updating one or both of the fragments may
be necessary. In addition, updating a table (actually a fragment) would cause the record (or
tuples) to migrate to the appropriate site to conform with the fragment definition in the
catalog (e.g., account record ACCOUNT#=123 is located in the fragment SF1 because
ACCOUNTS BRANCH=SF; if the branch is changed to LA for account record 123, the
DDBMS would migrate that record to fragment LA1).

Fragments behave like relational VIEWs. Updateable VIEWs are either not supported by
some vendors or supported only on views of a single table. If vendors do not solve the
updateable view issue, they cannot solve this problem.

Rule 6: Replication Independence.
A relation or fragment can be represented by more than one distinct, physically stored
replica or copy at more than one site. Replicas are totally unknown to the user(except for
the data base administrator) and serve two purposes:

- Performance—If a query can be serviced locally(using a replica and avoiding the
  network overhead), performance over a nonlocal query can be improved. The term
  query refers not only to requests to read data but also to insert, delete, and update
  operations. These operations may be initiated interactively or embedded in an
  application.

- Availability of data—If the site where the data resides does not respond to the query,
  the DDBMS can search for the closest replica and retrieve the data (see Exhibit 2). The
  following is an example of a SQL replica definition in which customer accounts must
  be physically replicated at other branches:
CREATE TABLE ACCOUNTS
(ACCOUNT#, BRANCH, CUSTOMER, BALANCE, TYPE)
CREATE REPLICA ACCOUNTS_PRIME
FROM ACCOUNTS

The database administration must also be able to define the location of the replica to the global catalog through the table allocation catalog—for example:

LOCATE ACCOUNTS AT CPU12
LOCATE REPLICA ACCOUNTS_PRIME AT CPU65

Because replicas are live copies, when an update is performed on the primary copy of the table, all replicas must also be updated; this process is called update propagation. Management of update propagation is tricky, depending on the algorithm for handling the propagation of the updates. Each copy must reflect the update. As the number of replicas increases, although read-only access times decrease, update response times increase (because there are more replicas to keep in synch).

Other schemes or partial solutions include deriving read-only copies, called snapshots, which resolve the update propagation issue but introduce another problem: retention of data that no longer supports business requirements. (Surprisingly, users can live with this data, especially when it is consolidation or summary data.) Snapshots are like views, except that they are stored in the database. The following is an example of a Structured Query Language snapshot definition in which a replica of the ACCOUNTS table must be refreshed or updated every day:

CREATE SNAPSHOT ACCOUNTS_PRIME
AS SELECT * FROM ACCOUNTS
REFRESH EVERY DAY
LOCATE SNAPSHOT ACCOUNTS_PRIME AT CPU65

**Rule 7: Distributed Query Processing.**

Distributed query processing is a query (e.g., read or update) that can span more than one site and is more complex than it appears. A request should support multisite views and integrity constraints, which can be foreign key across nodes, primary key across nodes in the case of fragments, or a logic constraint on a query. For example, certain employee information in a particular department is located at a different site from the employee and department tables, but multisite views and integrity constraints still appear to the user as a single-site database.

Due to such complexities and others (e.g., cardinality, including the table's number of tuples or records, machine capacity of each site, and network speeds and disk space constraints), a query optimizer should be an integral part of any DDBMS. Query optimization becomes much more complicated in a distributed environment, however, where queries can involve the services of several local servers. The optimizer must determine which requests to send to which site, how best to combine the results, and how to analyze communications speeds and the amount of traffic on the data lines. For example, when a distributed database is first activated, it polls the participating local data bases and updates its system catalog or dictionary with current information about those local data bases. The optimizer's output is the query execution plan used by the coordinator node; the optimizer organizes the query into subqueries, sends the subqueries to appropriate nodes,
receives and consolidates the results, and presents the resultant data to the process that initiated the query.

Because the optimization process can involve system overhead at one or more sites, the ability to compile queries before they are used can be desirable. Over time, the execution plan derived from a combined query becomes less efficient as the dynamics of the database change.

**Rule 8: Distributed Transaction Management.**
This rule covers two important areas of transaction management: recovery and concurrency. A transaction is an all-or-nothing proposition; either all activities required to complete the transaction are completed or none can be completed. A single transaction can involve many sites and many updates at each site.

Recovery is the process of correcting errors caused by failure and restoring the database to a consistent (i.e., correct) state. In a distributed DBMS environment, there is a greater possibility for failure, yet the effect of a failure is less than in a centralized DBMS environment because data may still be available locally or at other remote sites that have not failed. If a transaction performs some, but not all, updates and then fails, those updates (called uncommitted updates) must be rolled back to their state prior to the update.

If there is failure and the transaction completes successfully, the uncommitted updates must be committed, or made permanent; rolling back uncommitted updates is the recovery control process. A method or protocol is required to ensure that the transaction is either committed at all sites or rolled back at all sites. The two-phase commit protocol is the most common method for ensuring reasonable consistency.

The two-phase commit protocol requires that one site manage the committing or the rolling back of the transaction; the site that initiated the transaction is generally responsible for managing this process and is called the coordinator. If one site is selected to manage every transaction and if that site fails, none of the transactions in any of the sites can be initiated or completed. This loss of autonomy violates the spirit of local autonomy (Rule 2). The sites that help complete the transaction are called participants. Because transactions could come from any site and could be required to execute at any site, each site must be able to act as coordinator and participant.

Concurrency control helps ensure that two processes do not operate on the same data simultaneously. Three forms of inconsistency can result:

- **Lost updates**—Updates are lost through improper backups or undoing a transaction.
- **Dirty reads**—Data is read before a transaction is committed.
- **Nonrepeatable reads**—The system reads data before and after an update.

To prevent these anomalies, systems use locking. For example, if two transactions from two coordinators (e.g., from sites B and C) are pending at site A, site A is responsible for locking out one transaction while allowing the other transaction access to the data.

There are many varieties of locking schemes and different levels of granularity (e.g., table, tuples, or field). In distributed database environments, each local site is responsible for managing the locking of its data. Because each site is a different system, site A may not know that it is waiting for a lock to release at site B, and site B is waiting for site A to release a lock, a condition called global deadlock. The easiest and most common deadlock detection scheme is the time-out. With time-out, after a predetermined period of time, the
DDBMS assumes that a global deadlock has occurred and the coordinator begins rolling back the transaction.

**Rule 9: Hardware Independence.**
A DDBMS has achieved hardware independence when the same DDBMS software can operate on multiple hardware platforms. The fact that one vendor's product can run on 50 platforms and another on only five does not mean that the 50-platform product is better. If the information architecture specifies three hardware platforms, the vendor that supports those platforms has achieved hardware independence.

**Rule 10: Operating System Independence.**
Operating system independence closely resembles hardware independence (rule 9) and requires no further explanation.

**Rule 11: Network Independence.**
A DDBMS that can be supported on different hardware and operating systems must support a variety of communications protocols. From the viewpoint of the DDBMS, the hardware, the operating system, and the communications network can all be considered as providers of reliable services.

**Rule 12: DBMS Independence.**
DBMS independence is the next logical sequence of data independence following rules 9, 10, and 11. Different (i.e., heterogeneous) DBMSs can participate in a distributed database environment. What appears to users as a single logical data base may be several physical data bases, each supported by different DBMS software. For example, users of DBMS software product X want to access data at another site that uses DBMS software product Y. A common interface is required to allow each vendor's DBMS product to participate in a multivendor distributed database environment by handling all the functional requirements of the rules. No common interface currently exists, however, although there are provisions, called gateways, for retrieval and update between relational DBMSs as well as between nonrelational DBMSs, which use very complicated gateways.

A gateway is an interface between DDBMS products from different vendors (see Exhibit 3). If DDBMS product X must interface with three other DBMS products, it must supply three gateways. Gateways work in one direction only, however, so if DDBMS product X has a gateway to DBMS Y, X's users can access data in Y, but not the other way around—unless a gateway that interfaces with X is provided by product Y. The number of gateways proliferates quickly for vendors willing to participate in heterogeneous DDBMS environments. A common distributed database interface is needed to address all the issues, including recovery, two-phase commit, locking protocols, and internal data representation translation.

### Gateways

**Conclusion**

Although DDBMS technology is immature, and DDBMS vendors implement only some of the functional requirements identified in the rules described in this article, the vendors' strengths are primarily read-only access and limited update in a heterogeneous hardware,
software, and network environment that includes gateways. As DDBMSs evolve, however, sharing and integration capabilities should improve.

Distributed data base technology should also enable companies to protect investments in their current DBMSs while allowing new data base developments in which incremental costs are added more closely to the users of the data. Adding distributed data base technology to the information architecture should support the integration of some legacy data as well as the business requirements that drive the distribution of data throughout the company and around the world.

**Bibliography**


“What is a Distributed Database System?” InfoDB(Summer and Fall 1987).

Doak, J.A. “IBM Distributed Relational Data.“ InfoDB(Spring 1989).


**Author Biographies**

Michael D. Krasowski

Michael D. Krasowski is senior principal consultant at McDonnell Douglas Corp, responsible for developing Douglas Aircraft Co's IA strategic plan and the accompanying planning process, content, and support facilities.
a. User's View

b. Database Administrator's View