DISTRIBUTED DATABASES

M. Tamer Özsu

University of Alberta

A distributed database (DDB) is a collection of multiple, logically interrelated databases distributed over a computer network. A distributed database management system (distributed DBMS) is the software system that permits the management of the distributed database and makes the distribution transparent to the users [1]. The term distributed database system (DDBS) is typically used to refer to the combination of DDB and the distributed DBMS. Distributed DBMSs are similar to distributed file systems (*see* Distributed File Systems) in that both facilitate access to distributed data. However, there are important differences in structure and functionality, and these characterize a distributed database system:

1. Distributed file systems simply allow users to access files that are located on machines other than their own. These files have no explicit structure (i.e., they are flat) and the relationships among data in different files (if there are any) are not managed by the system and are the users responsibility. A DDB, on the other hand, is organized according to a *schema* that defines both the structure of the distributed data, and the relationships among the data. The schema is defined according to some data model, which is usually relational or object-oriented (*see* Distributed Database Schemas).
2. A distributed file system provides a simple interface to users which allows them to open, read/write (records or bytes), and close files. A distributed DBMS system has the full functionality of a DBMS. It provides high-level, declarative query capability, transaction management (both concurrency control and recovery), and integrity enforcement. In this regard, distributed DBMSs are different from transaction processing systems as well, since the latter provide only some of these functions.
3. A distributed DBMS provides *transparent* access to data, while in a distributed file system the user has to know (to some extent) the location of the data. A DDB may be partitioned (called *fragmentation*) and replicated in addition to being distributed across multiple sites. All of this is not visible to the users. In this sense, the distributed database technology extends the concept of *data independence*, which is a central notion of database management, to environments where data are distributed and replicated over a number of machines connected by a network. Thus, from a user s perspective, a DDB is logically a single database even if physically it is distributed.

Architectural Alternatives

Architecturally, a distributed database system consists of a (possibly empty) set of *query sites* and a non-empty set of *data sites*. The data sites have data storage capability while the query sites do not. The latter only run the user interface (in addition to applications) in order to facilitate data access at data sites (Figure 1).

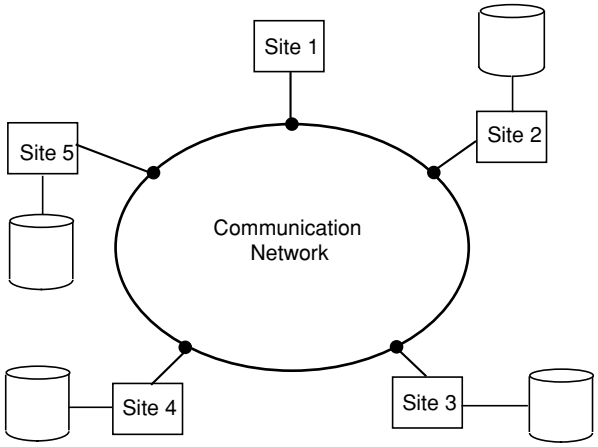

Figure 1. Distributed Database Environment

If the distributed database systems at various sites are autonomous and (possibly) exhibit some form of heterogeneity, they are referred to as *multidatabase systems* (*see* Multidatabase Systems) or *federated database systems* (*see* Federated Database Systems). If the data and DBMS functionality distribution is accomplished on a multiprocessor computer, then it is referred to as a parallel database system (*see* Parallel Databases). These are different than a distributed database system where the logical integration among distributed data is tighter than is the case with multidatabase systems or federated database systems, but the physical control is looser than that in parallel DBMSs.

There are a number of different architectural models for the development of a distributed DBMS, ranging from client/server systems [2], where the query sites correspond to clients while the data sites correspond to servers (see Client-Server Computing), to a peer-to-peer system where no distinction is made among the client machines and the server machines. These architectures differ with respect to the location where each DBMS function is provided.

In the case of client/server DBMSs, the server does most of the data management work. This means that all of query processing and optimization, transaction management and storage management is done at the server. The client, in addition to the application and the user interface, has a *DBMS client* module that is responsible for managing the data that is cached to the client and (sometimes) managing the transaction locks that may have been cached as well. A typical client/server functional distribution is given in Figure 2.

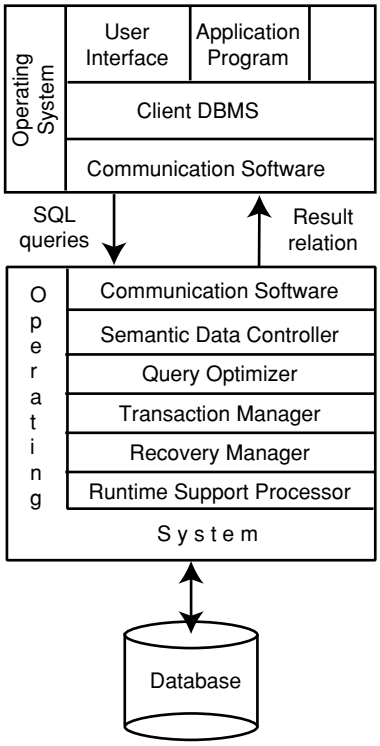

Figure 2. Client/Server Architecture

The simplest client/server architecture is a *multiple-client/single-server* system. From a data management perspective, this is not much different from centralized databases since database is stored on only one machine (the server) which also hosts the software to manage it. However, there are some important differences from centralized systems in the way transactions are executed and caches are managed. A more sophisticated client/server architecture is one where there are multiple serves in the system (the so-called *multiple-client/multiple-server* approach). In this case, two alternative maangement strategies are possible: either each DBMS client manages its own connection to the appropriate server or each client knows only its home server_ which then communicates with other servers as required. The former approach simplifies server code, but loads the client machines with additional responsibilities (heavy client) while the latter approach concentrates data management functionality at the servers and provides transparency of data access at the server interface (ight client).

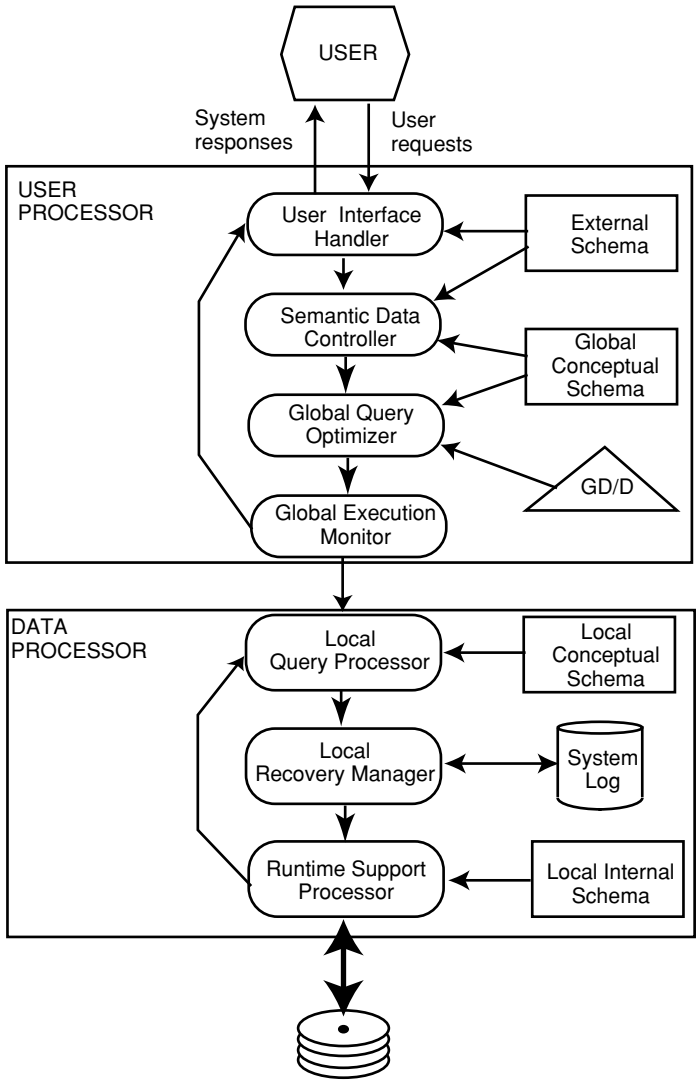

Figure 3. Peer-to-Peer Distributed DBMS Functionality

In the case of peer-to-peer systems, there is no distinction between clients and servers and each site in the system can perform the same functionality. It is still possible to separate the modules that serve user requests from others that manage data (see Figure 3), but this is only a logical separation and does not imply any functionality distribution. In executing queries (transactions), it is possible for the global query optimizer (global execution monitor) to communicate directly with the local query processors (local recovery managers) where parts of the query (transaction) are executed. Thus, the communication mechanism is more involved, leading to more complicated software structures.

Technical Issues

As indicated above, distributed DBMSs provide the same functionality as their centralized counterparts. Thus, the same technical issues arise, albeit in a distributed context. The distribution also increases the difficulties in dealing with these issues due to data fragmentation/replication/distribution, lack of instantaneous knowledge about what is going on at another site, and the general difficulty of managing a distributed environment.

Query Processing and Optimization

Query processing is the process by which a declarative query is translated into low-level data manipulation operations. SQL [3] is the standard query language that is supported in current DBMSs. Query optimization refers to the process by which the best execution strategy for a given query is found from among a set of alternatives.

In centralized DBMSs, the process typically involves two steps: *query decomposition* and *query optimization.* Query decomposition takes an SQL query and translates it into one expressed in relational algebra. In the process, the query is analyzed semantically so that incorrect queries are detected and rejected as easily as possible, and correct queries are simplified. For a given SQL query, there are more than one possible algebraic queries, some of which are better than others, as defined in terms of expected execution performance. Finding the best among the alternatives is the process of query optimization and requires consideration of the physical organization of the data (i.e., the availability of indexes and hash structures).

In distributed query processing/optimization (*see* Distributed Query Processing), the objective is to ensure that the user query, which is posed as if the database was centralized (i.e., logically integrated), executes correctly and efficiently over data that is distributed. Therefore, two more steps are involved between query decomposition and query optimization: data localization and *global query optimization*. Data localization takes the algebraic query that is obtained as a result of query decomposition and localizes the query s data using data fragmentation/distribution information. In this step, the fragments which are involved in the query are determined and the query is transformed into one that operates on fragments rather than global data units specified in the user query. Global query optimization step takes the localized query and determines the best method of executing operations that span multiple sites. These operations are the usual binary relational operators such as join, union, intersect, etc.

Distributed Transaction Management

A transaction is a unit of atomicity and consistency, and ensure that the database integrity is maintained in the face of concurrent accesses and system failures (*see* Transaction Management, Transaction Processing). DBMSs provide support for what is called ACID (Atomic, Consistent, Isolated, and Durable) transactions: a transaction is an atomic unit of execution, either all of its actions are completed or none of them are; a transaction is consistent in that it transforms a consistent database state to another consistent database state; a transaction is isolated from other running transactions by hiding its modifications until it is completed and not accessing the modifications that other transactions are concurrently making to the database; a transaction is durable so that, upon successful completion, the results will survive any system failure [4].

Supporting ACID transactions in a distributed setting involve, in addition to the concurrency control and reliability protocols described below, architectural issues having to do with the control and coordination of executing parts of transactions at different sites. In the model presented in Figure 3, this coordination is accomplished by the Global Execution Monitor at the site where the transaction is first initiated. The Global Execution Monitor submits parts of this transaction to the relevant Local Recovery Modules for actual processing and controls that the ACID properties are maintained.

Distributed Concurrency Control

Concurrency control involves the synchronization of concurrent accesses to the distributed database, such that the integrity of the database is maintained. If the result on the database of concurrent execution of a set of transactions is equivalent to some serial (one after the other) execution of the same set of transactions, then the concurrently executing transactions are said to have maintained the consistency of the distributed database. This is known as the *serializability* condition. In terms of the ACID properties, concurrency control algorithms maintain the consistency and isolation properties of transactions.

Concurrency control of distributed transactions requires a distributed synchronization algorithm which has to ensure that concurrent transactions are not only serializable at each site where they execute, but that they are also globally serializable. This implies that the order in which they execute at each site have to be identical.

Distributed concurrency control algorithms can be grouped into two general classes as *pessimistic*, which synchronize the execution of user requests before the transaction starts, and *optimistic*, which execute the requests and then perform a validation check to ensure that the execution has not compromised the consistency of the database. Two fundamental primitives that can be used with both approaches are *locking*, which is based on the mutual exclusion of accesses to data items (similar to semaphores in operating systems), and *timestamping*, where the transactions are executed in some order. There are variations of these schemes as well as hybrid algorithms that attempt to combine the two basic mechanisms. Locking-based algorithms cause distributed deadlocks requiring protocols to handle them (*see* Distributed Deadlock Detection).

Distributed Reliability Protocols

Distributed database systems are potentially more reliable since there are multiple copies of each system component, which eliminates single points-of-failure, and data may be replicated to ensure that access can be provided in case of system failures. Distributed reliability protocols maintain the atomicity and durability properties by (a) ensuring that either all of the actions of a transaction that is executing at different sites complete successfully (called *commit*) or none of them complete successfully (called *abort*), and (b) ensuring that the modifications made to the database by committed transactions survive failures. The first requires *atomic commit protocols* (*see* Commit Protocols), while the second requires *recovery protocols*.

The most common atomic commit protocol is the *two-phase commit* (2PC) where the transaction is committed in two steps: first all the sites where the transaction has executed are polled to make sure they are ready to commit, and then they are instructed to actually commit the transaction (*see* Two-Phase Commit). The result is uniform commitment (or abort) at every site where the transaction executes.

Recovery protocols are the converse of atomic commitment protocols. They ensure that the system can recover to a consistent state following a failure.

Replication Protocols

In replicated distributed databases each *logical data item* has a number of physical instances. Queries and transactions refer to logical data items and the replication protocols reflect their actions on the physical instances. The issue in this type of a database system is to maintain some notion of consistency among the copies. The most discussed consistency criterion is *one copy equivalence*, which asserts that the values of all copies of a logical data item should be identical when the transaction that updates it terminates [5].

A typical replica control protocol that enforces one copy equivalence is the *Read-Once/Write-All* (ROWA) protocol which maps each read operation a logical data item to a read on one of the physical copies which may be determined by performance considerations. On the other hand, each write operation on a logical data item is mapped to a set of writes on *all* physical replicas. ROWA protocol is simple and straightforward, but failure of one site may block a transaction, reducing database availability. A number of alternative algorithms have been proposed which reduce the requirement that all copies of a logical data item be updated before the transaction can terminate. They relax ROWA by mapping each write to only a subset of the physical copies (*see* Replication Control Protocols).

References

[1] M.T. zsu and P. Valduriez, *Principles of Distributed Database Systems, 2nd edition*. Prentice-Hall, Inc. 1998.

[2] R. Orfali, D. Harkey, and J. Edwards, *Essential Client/Server Survival Guide, 2nd edition*. Wiley, 1996.

[3] C.J. Date and H. Darwen, *A Guide to the SQL Standard - A User's Guide to the Standard Database Language SQL, 4th edition*, Addison-Wesley, 1997.

[4] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[5] A. A. Helal, A. A. Heddaya, and B. B. Bhargava, *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, 1997.

Cross Reference:

Client-Server Computing *see* Databases, Distributed.

Database Clustering, Distributed *see* Databases, Distributed.

Database Schemas, Distributed *see* Databases, Distributed.

Deadlock Detection, Distributed *see* Databases, Distributed.

Federated Databases *see* Databases, Distributed.

File Systems, Distributed *see* Databases, Distributed.

Interoperability *see* Databases, Distributed.

Multidatabase Systems *see* Databases, Distributed.

Parallel Databases *see* Databases, Distributed.

Query Processing, Distributed *see* Databases, Distributed.

Relational Databases, Distributed *see* Databases, Distributed.

Replicated File Systems *see* Databases, Distributed.

Replication Control Protocols *see* Databases, Distributed.

Transaction Management *see* Databases, Distributed.

Transaction Processing *see* Databases, Distributed.

Two-Phase Commit *see* Databases, Distributed.

Dictionary Terms:

Atomicity

The property of transaction processing whereby either all the operations of a transaction are executed or none of them are (all-or-nothing).

Distributed database

A collection of multiple, logically interrelated databases distributed over a computer network.

Distributed database management system

The software system that permits the management of the distributed database and makes the distribution transparent to the users

Durability

The property of transaction processing whereby the effects of successfully completed (i.e., committed) transactions endure subsequent failures.

Isolation

The property of transaction execution which states that the effects of one transaction on the database are isolated from other transactions until the first completes its execution.

One copy equivalence

Replica control policy which asserts that the values of all copies of a logical data item should be identical when the transaction that updates that item terminates.

Query optimization

The process by which the best_ execution strategy for a given query is found from among a set of alternatives.

Query processing

The process by which a declarative query is translated into low-level data manipulation operations.

Serializability

The concurrency control correctness criterion which requires that the concurrent execution of a set of transactions should be equivalent to the effect of some serial execution of those transactions.

Transaction

A unit of consistent and atomic execution against the database.

Transparency

Extension of data independence to distributed systems by hiding the distribution, fragmentation and replication of data from the users.

Two-phase commit

An atomic commitment protocol which ensures that a transaction is terminated the same way at every site where it executes. The name comes from the fact that two rounds of messages are exchanged during this process.