Distributed Database Systems

by Raymond Board [1]
Division of Research and Statistics Federal Reserve Board Washington, DC 20551

abstract
A *distributed database system* is a collection of logically-related databases that are connected by a communications network, together with a software system for managing and accessing the data. A distributed database system is designed so that it appears to the user to be a single, unified database.This paper reviews the advantages and disadvantages of distributed database systems, and discusses issues relevant to their design and use. These issues include concurrency control, distributed query processing and transaction management, disaster recovery, reliability, and methods for distributing data.

Background
Much social science research involves the use of large datasets. These datasets are usually shared among many researchers, each of whom periodically wants to read, update, correct, and add to the data. Managing such datasets presents significant problems, and maintaining them is not an easy task. Frequently, specialized computer software systems, known as *database management systems*, are used to handle large datasets accessed by multiple users.

A recent trend in computing has been to move away from the traditional mainframe environment toward a distributed computing environment. In a mainframe environment, all users share the same large, powerful computer. In a distributed environment, many computers — usually smaller machines such as personal computers (PCs) — are linked together by a communication network. Such networks can connect users working on different types of machines and in different locations.

This paper discusses an emerging computer technology that is intended to solve many of the problems associated with large datasets. This technology, called *database management systems*}, takes advantage of a number of the benefits offered by distributed computing environments. As this is a relatively new technology, the reader should note that not all of the features of distributed database systems are currently available in commercial products. Building such systems is a very active area of computer science research; although network database systems with many of the features herein described are already on the market, a complete distributed database management system is still a few years away. Our discussion will be of a general nature; in particular, we will not discuss different data models, such as hierarchical, graph-based, and relational, currently used by database management systems.

Database Management Systems
A *database* is an organized collection of data stored on a computer system. A *database management system,* abbreviated as *DBMS,* is a software system for managing and accessing a database. These systems are typically used to manage data that is[2] accessed by applications programs (e.g. packages for statistical analysis), or to answer direct user queries.

It is certainly possible to store data in ordinary files in a computer's file system; in fact, for small datasets that are needed by only a single user, this is often the most practical solution. But for large datasets, and particularly those datasets that are accessed by multiple users, this practice has a number of drawbacks. DBMS', on the other hand, are designed specifically to handle these sorts of situations, and to overcome the limitations presented by using ordinary files.

In particular, DBMS' are designed to handle large datasets that are accessed (often simultaneously) by many users, for both reading and writing. These datasets are usually of critical importance to the user, so protecting the accuracy and integrity of the data stored in them is of paramount importance. Some of the crucial issues that must be dealt with by a DBMS include the following:

* What should be done if more than one person wants to access the data at the same time?

* What happens if one person is changing data at the same time someone else is reading it?

* Is the data safe if the system crashes? What if someone was making changes to the data when it happened?

* How can the data be accessed quickly, even when the dataset is very large?

These are some of the problems that a database manage-

ment system must address.

Network Computing

In recent years there has been a strong trend toward replacing mainframes with networks of smaller computers. The most common such arrangement is a local- or wide-area network that connects a collection of personal computers or workstations[3]. This migration is largely the result of the more favorable price/performance ratios currently offered by PCs and workstations. The lower cost is partly due to the high servicing expenses (usually provided by the vendor) required by most mainframes; PC maintenance can frequently be handled by the customer herself. The main reason, however, is the low prices resulting from the intense competition among PC and workstation manufacturers. Furthermore, the explosion in the number of these machines now in use has led to the availability of a wide range of application software to run on them. The proliferation of these smaller computers has put computing power and data closer to their end-users, often right on their desktops.

Another factor contributing to the spread of network computing environments is the growing popularity of the UNIX operating system. UNIX is particularly well-suited for networking, so its increased use has encouraged the implementation of network computing environments.

Unlike the case with mainframes, data storage and processing are distributed in a network environment. Data can be stored at many different sites on the network, and computation can be performed on the machines best-suited to the individual tasks. Network file systems can make data location-transparent to users, so that they don't have to know where the data they're using actually resides. Through the use of remote procedure calls, the same can be true of computation. Software can be implemented so that users need not know which machine is running their applications.

Distributed Database Systems}

The popularity of network computing environments has led to the development of *distributed database systems*. A *distributed database* is a collection of logically-related databases that are connected by a communications network. A *distributed database management system* (or *distributed DBMS*) is a software system for managing and accessing a distributed database. A distributed database system consists ofa distributed database and its associated distributed DBMS. The distributed database system is designed so that it appears to the user to be a single, unified database. Typically, the data stored by a distributed database system is spread across a number of computers on the network. The data is distributed with the two following considerations in mind. First, data should be stored close to the machines that are most likely to run applications that use it, thus minimizing network traffic. Second, the amount of data stored on the different computers should be well-balanced, to even out the data processing load and thus eliminate potential botlenecks.

A distributed DBMS differs from a network file system in that its data is logically structured and is accessed by means of a high-level software interface. This interface is written so as to control concurrent access to the data and ensure data integrity. The data in a network file system is only organized into files, and can only be retrieved as files. Note also the distinction between "distributed databases" and "distributed processing". The former refers to spreading databases across two or more computers, while the latter refers to spreading processing across multiple computers. Distributed database systems typically perform distributed processing, but the two terms are not equivalent.

The client-server model

An increasingly popular network database configuration is the *client-server* architecture. This refers to a network computing environment in which one or more machines function as database *servers*. These computers store the data and manage all database operations. Other machines, known as *clients*, run application programs. When a client application needs to read or write to the database, it sends an appropriate request to a server. The server processes the request and returns the result to the client, which then resumes running its application.

It is important to point out that client-server database architectures are not necessarily distributed database systems. One reason is that there need not be more than one server on a network; thus the data is not necessarily distributed. Another reason is that the client-server model does not require location transparency; in this model it is acceptable to require that the user know where her data resides on the network. In a distributed database system, the actual location of the data should be transparent to the user.

Client-server architectures are frequently used in network computing environments, and not just for managing databases. In addition to database servers, individual machines often function as file servers or mail servers for the network, handling client requests for those resources.

Issues

The questions — described above in Section Database Management Systems — that "undistributed" DBMS' must deal with are also important problems for distributed DBMS'. In fact, these problems become more complex in a distributed environment. In addition, new issues arise because of the distributed setting. We discuss some of these issues in this section, and how they are handled by distributed DBMS'.

Concurrency control

Recall that one of the problems that any DBMS must deal with is what to do when more than user wants to access the same data at the same time. This can be especially troublesome when one (or both) of the simultaneous users wishes to change or update the data in the database. In the database world, these questions fall under the heading of {\em concurrency control}. As its name suggests, concurrency control means managing concurrent access to data by multiple users. Two techniques commonly used by DBMS's to provide concurrency control are *data replication* and *locking*.

In one common data replication scheme[4], each user wishing to access a particular set of data in the database is given her own copy of the data. (The copy is transparent to the user; to her it appears as though she's working with the database itself.) If she only wishes to read data, then she only has to refer to her own copy. But if she wishes to *write* to the database, *all* replicated copies must be updated; otherwise, users who are also working with that part of the database will no longer have an up-to-date copy once she has made her changes. Thus each time a user updates the database, all copies of that part of the data that are simultaneously in use must also be updated. This requires a lot of disk writes, which is a slow operation. Thus system performance can suffer under this sort of scheme. In a distributed DBMS, the updates to all replicated data will cause an increase in network traffic, in addition to disk operations. Thus the performance degradation becomes even more of an issue in a distributed DBMS.

An alternative technique often used to enforce concurrency control is locking. When an application program (or a user making direct queries to the database) needs access to a particular piece of data, it requests a lock — a guarantee of temporary exclusive access — to that part of the database. If another process has already been granted a lock on that data, then the application program must wait until that lock has been released. Thus at any time, at most one process has access to any piece of data in the database.

Locking has drawbacks also. The most obvious one is that when multiple processes need access to the same data, all but one of them must wait until they can obtain a lock. This problem can be alleviated somewhat by shrinking the granularity of the lock; that is, by making the lock apply to only a very specific piece of data. This makes conflicts less likely, but also increases system overhead, since processes will have to request locks more frequently. A similar tradeoff takes place when data replication is used. If the granularity of the replicated data is large, then the user needs to request additional data copies less frequently. However, this enhances the probability that she's working with data that is out-of-date. Reducing the granularity, on the other hand, increases the number of data retrievals required, and thus the system overhead.

Another problem introduced by locking is the possibility of *deadlock*.. Suppose that two processes, P_1 and P_2, are running simultaneously, and both need access to both of the data items d_1 and d_2. Suppose further that P_1 requests and is granted a lock on d_1, while at about the same time P_2 requests and is granted a lock on d_2. Now neither process can proceed, since each is waiting for the other to release its lock. This situation is called deadlock, and is clearly something to be avoided. DBMS' typically have subroutines that periodically check for this sort of condition; if it's found, one of the deadlocked processes is forced to relinquish its lock, so that the other process can proceed.

Deadlock detection is more difficult in a distributed DBMS, since the locks that the processes are competing for may be at two different sites on the network. Since each site typically handles the locks on its own data, it's possible that neither of the two sites realizes that it's waiting for a lock to be released on the other machine. This stalemate situation is known as *global deadlock*, and is much harder to detect, since there is no central program managing all of the locks. Distributed DBMS' often detect it by "timeout": Once a certain period of time has elapsed without the locks being granted, a global deadlock condition is assumed to exist.

Concurrency control in distributed DBMS' is further complicated by the possibility of communication or site failure. A message relinquishing a lock may be garbled or lost, or a computer may crash without releasing its locks. Either of these situations results in *dangling locks*, which are no longer needed but have not been released. Distributed DBMS' must have contingency plans for detecting and dealing with these situations.

As an example of concurrency control, consider an airline reservation system. This type of system is centered on a large database that can be accessed by thousands of ticket agents around the world. Clearly some sort of concurrency control is needed to prevent two agents from simultaneously booking the same seat. When one agent tries to reserve a seat, she is granted a lock on that seat. The lock granularity should allow the agent to simultaneously lock two adjacent seats for a couple traveling together, but also allow other agents to book other seats at the same time.

Consistency

Maintaining database *consistency* is another important issue. Consider a bank, and two customers (*A* and *B*) who have accounts there. Suppose that *A* writes a check for $100 to *B*, who deposits it in her account. Recording this

transaction in the database thus requires two operations: The balance in *A* 's account must be decreased by $100 and the balance in *b"*s account must be increased by the same amount. But what happens if an accounting program reads the balances after *A* 's account has been adjusted but not *B*'s? The accounting program will be told that there is $100 less in the bank than is actually the case. The database will be in an inconsistent state.

To prevent this sort of inconsistency, DBMS' allow users to group database operations into *transactions*. These are sequences of database write operations that are treated as a single unit; either all of the operations in the transaction will be carried out (or *committed*), or else none of them will be. Furthermore, no other changes will be made to the database between the time the first operation in the transaction is executed and the time that last operation in the transaction is executed. In the banking example, the DBMS would guarantee that the accounting program could not gain access to the account information until after the adjustments to both of the accounts were made. The accounting program would thus see a consistent version of the database.

In a distributed DBMS it is important that, in transactions that affect data at multiple sites, either all of the sites are updated or else none of them are. This is usually ensured through the use of a *two-phase commit* protocol. In the first phase, the machine initiating the transaction sends a message to all of the sites that will be affected by the transaction, asking them to verify that they are prepared to commit the transaction. If each of these sites responds positively, then the initial machine sends a second message instructing the other sites to actually commit the transaction. If not all of the sites respond positively, perhaps because of a communication failure, then the initial machine sends out a second message cancelling the transaction at all of the sites. This ensures that each site maintains a consistent version of the database. This type of protocol requires a significant amount of system overhead.

Disaster recovery

A very important issue is how to protect the integrity of the data during a system crash. Normally, if the system goes down in between transactions, there is no major problem. This is because at the end of a transaction the database will be in a consistent state (assuming that transactions are managed as described above). Of greater concern is the possibility that the system could crash in the middle of a transaction. In this case, some of the changes made inside the transaction may have been executed (i.e. written to disk), but not others. Thus at the time of the crash the database could be in an inconsistent state. In the example above, if the system failure occurred after *A* 's account was debited but before *B*'s account was credited, then the database would understate the bank's deposits by $100.

DBMS' frequently address this problem, known as *disaster recovery*, by keeping track of all transactions in a log file. Then, in the event of a system crash, the log file is read automatically by the DBMS (once the system is functional again) to see if any transactions were in progress at the time of the failure. If there was such a transaction, all operations in that transaction that were executed prior to the crash are "undone", so that the database is returned to the (consistent) state it was in at the time the transaction started. In a distributed database system each site typically maintains its own log file. Another common technique for guarding against system failure is to make new copies of the disk pages where the data to be updated resides. The transaction operations are performed on the copy. When the transaction is completed, the updated disk page can be remapped to replace the old data in the database in a single, atomic operation.

Query optimization

A primary goal of all DBMS' is to provide fast access to information in the database, even when the database is very large. The speed with which a DBMS can answer a query from a user or an application program relies to a considerable extent on the order in which it carries out the database operations necessary to extract the requested information. Thus to ensure efficient performance, a DBMS must be able to optimize the execution of queries.

As a (rather simplistic) example of the importance of ordering the operations wisely, consider the "Irish presidents" problem. Suppose there is a database containing information on many thousands of current and past US citizens, and a request is made for a list of all people in the database who were both of Irish ancestry and American presidents. One way to satisfy the request would be to first retrieve all people in the database of Irish ancestry, and then select from this (very large) list those people who were also presidents. A much more efficient way to generate the list would be to initially retrieve all US presidents, and then select from this much shorter list those who are also of Irish descent.

Many DBMS' allow precompilation of queries. That is, if a certain type of query will be executed many times, the user can compile it into an optimized form that can then be used for all subsequent invocations; optimization is thus performed only once. For ad hoc queries that will only be executed once, the DBMS must perform the optimization when the query is actually made. Since optimization can be a time-consuming operation, a tradeoff can arise between the time needed to perform the optimization and the time saved by executing an optimized version of the query. In these situations it may be best to perform less extensive (and thus less time-

consuming) optimization.

Query optimization is especially important for distributed databases, particularly since individual queries may involve data stored at more than one site. Communications overhead is a major concern in distributed environments, due to the relatively slow speed of network communication relative to most other operations. Thus it is important to optimize queries so as to minimize the amount of network traffic, in terms of both the frequency of communication and the size of the messages passed. Global optimization, which takes into account communication times between sites, is needed for maximum efficiency, rather than just local optimization at each of the database sites. Note that the more autonomous the individual sites are, the harder this will be to do, since effective global optimization requires that much information on the distributed data be available in a central location.

Data distribution A key aspect of the definition of a distributed database system is that the data is distributed among multiple sites on the network. The way in which the data is distributed can dramatically affect system performance.

The question of how data is to be distributed can be divided into two parts, *fragmentation* and *allocation*. Fragmentation refers to how the data is broken into pieces. Once this has been determined, the data fragments must be allocated to various sites on the network. When large networks and databases are involved, finding an optimal (or near-optimal) fragmentation and allocation can be a very difficult problem. While it's important to put data close to users, it's also crucial to distribute the data so as to reduce network traffic, and to balance the distribution in order minimize the chances of bottlenecks appearing. Thus the frequency of access to the data fragments must be considered. Also, the structure of anticipated queries should be taken into account, with the goal of reducing the number of queries requiring data from multiple sites. A sound distribution design strategy should take into account all of these issues.

In order to improve performance, the system designer may wish to store multiple copies of some data fragments, particularly the most heavily-used data. This can result in frequently-accessed data being stored near many or all of its users, as well as reducing contention for individual copies of this data. The cost of such a strategy is that this may incur substantial system overhead; updates must be made to all copies of the replicated data, resulting in more network traffic and additional concurrency concerns. The replicated data fragments will also require more disk space.

Another important feature of a distributed database system is that the data should be *location-transparent*. That is, no matter how the data is fragmented and allocated, the user shouldn't have to know how or where it is stored. The interface to the distributed DBMS should be such that a user, or an application program that interacts with the database, views the distributed DBMS as a single logical database. Note that location-transparency implies that when the data in the database is moved around or restructured, existing application programs won't have to be altered to adjust for the changes.

Heterogeneous networks A practical consideration of some importance is that distributed DBMS' should be able to work on heterogeneous networks. Here "heterogeneous" refers not only to computer hardware, but also to the different types of network hardware, operating systems, communication protocols, and even database management systems that are commonly encountered. The last of these is critical, since much of the cost-effectiveness and usefulness of a distributed DBMS may result from its ability to link together a number of existing DBMS'.

A distributed DBMS that is able to run on a wide variety of systems enables widespread sharing of data among databases in environments like universities, where different types of computer systems abound. Another advantage to heterogeneity is that if a distributed DBMS runs on many types of systems, then it's easier to add existing databases to it. This way system administrators can protect their investments in existing systems by being able to integrate them into a larger system, rather than having to replace them.

Hardware and protocol heterogeneity can be achieved through the use of low-level communication interfaces called *gateways*. Once these interfaces have been established, there can still be communications problems if the distributed database system links together different types of DBMS'. Thus it may be necessary to translate between the two (or more) different DBMS' query languages. The software programs that translate DBMS requests into alternative query languages and send them to the appropriate sites are, confusingly, also known as gateways. Note that in a distributed database system with many different types of machines, protocols, and DBMS', the number of gateways (of both types) required can be very large. This problem could be ameliorated by the adoption of industry-wide standards for such things as data models, query languages, and concurrency protocols. Such comprehensive standards, however, seem unlikely to be established in the near future.

Advantages and Disadvantages

As might be expected, there are both advantages and disadvantages to distributed database systems. Perhaps the most obvious advantage is that such systems facilitate

the sharing of data among large communities of users — for example, among the faculties of different departments in a university — using existing, possibly heterogeneous, computer networks. Thus more users can have access to more data, without having to know where or how the data is actually stored.

User interfaces One advantage of a distributed DBMS that can be very apparent to end users is the superior variety of user interfaces available on PCs and workstations as compared with most mainframes. Graphical user interfaces (GUIs), allowing multiple windows and (often) bit-mapped displays, are commonly available on these smaller machines, and greatly enhance the enjoyment and productivity of the user. In a distributed database system, the user can work with a GUI to access data stored elsewhere on the network without having to learn and use the less user-friendly style of command interface that still exists on most mainframes.

Performance A related advantage is that computation-intensive applications can be moved off of the database server machine(s) and onto the users' individual PCs and workstations. This takes some of the processing load off of the servers, thus allowing all users faster access to data. In a mainframe environment, user applications compete with the DBMS software for the computer's CPU. But by processing the data locally in a distributed environment, greater processing capacity is achieved by keeping many machines busy.

Another way that performance gains can be realized in a carefully designed distributed database system is by moving data closer to the people who use it. By distributing data on the PCs or workstations of those most likely to use it, not only do those users benefit from faster data access, but other users benefit as well, due to the resultant lightening of the load on the other database servers on the network.

Note that, in addition to offering additional functionality such as concurrency control and data consistency, a distributed DBMS may also achieve better performance than network file systems in certain applications. This is because distributed DBMS' respond to queries, and thus need only send over the network the data that satisfies the specific query. In a network file system, however, only files can be transferred across the network. Thus a much larger amount of data than is actually needed by the requester is likely to be sent, resulting in increased communication time.

Incremental growth Distributed database systems also facilitate the incremental growth of databases. New machines, perhaps with new datasets mounted on their file systems, can be incorporated one by one into a distributed DBMS. Thus as the data to be stored outgrows the existing systems, new machines can be added to expand capacity. In a mainframe environment this type of incremental growth is generally not possible; the entire DBMS would have to be replaced with a new system, a much more expensive solution. A related advantage of distributed DBMS' is that they are well-suited for handling {\em legacy} systems. A legacy is a software program that, although today it might not be the best choice for its job, is so firmly entrenched in the user community (because of years of use, hundreds of applications that invoke it, etc.) that it would not be feasible to replace it. A legacy database system can be incorporated into a distributed DBMS by making it one node in the system. Applications requiring data from that DBMS could still use it, while other programs and users could use data stored on other machines on the network.

Reliability Robustness and reliability are other areas in which distributed DBMS' display advantages. In most cases, a failure at one or more sites on the network will not crash the entire distributed DBMS. In a well-designed distributed DBMS, not only the data but also the control over query processing, concurrency, and disaster recovery is distributed. Thus failure at one point will not render the entire database system useless. Although some data may be temporarily unreachable in the event of such a failure, much of the data should still be accessible. In addition, if the system is designed with careful replication of data on multiple sites, all users may be able to continue working with no ill effects if part of the system goes down.

Disadvantages The most telling drawback of distributed DBMS' is probably the increased complexity of administering and maintaining the database. Instead of just managing a single database, the database manager must now also contend with the network, communications software, data that is replicated on multiple machines, and the backup and recovery of distributed data. There are more possible points of failure, including the machine requesting data, the network, and the machine hosting the data. Testing new applications is harder, since there may be many different combinations of client and server machines that users will want to run the applications on. Software updates are more of a chore; instead of installing an update on a single machine, database administrators must ensure that all machines in the distributed database system are running up-to-date software. Finally, maintaining data security will be more difficult. There is an inherent conflict between granting wider data access to enable many machines on the network to use the database, and restricting access to sensitive data. In environments where sensitive data is stored, a careful balance must be struck between these competing interests.

Another potential disadvantage is that poorly imple-

mented distributed DBMS' may exhibit worse performance than their centralized counterparts. A system with a very high rate of transactions, and data that is not distributed efficiently, could result in very heavy network traffic. This traffic, combined with the overhead of the software managing the distributed DBMS' network communications, could cause communication delays that overcome the expected performance gains described above, and result in unsatisfactory performance. Thus careful thought must be given as to how data is distributed and replicated among the machines on the network.

Concluding Remarks

Although fully-functional distributed database systems, as described in this paper, are not yet a commercial reality, they appear to be a promising means of handling large shared datasets in the near future. Systems with many of the capabilities discussed are now available, and distributed client-server databases have been installed at a number of sites. Distributed DBMS' take advantage of many of the features that have made network computing environments increasingly popular. They distribute the processing load, move the data closer to the people who work with it, and allow cheap, incremental system evolution.

One unknown aspect of distributed DBMS' is how well the algorithms and protocols they use, such as two-phase commit, will scale up as networks grow larger and connect more and more computers. Further research is also needed on data distribution strategies and on improving transaction management and query processing in a distributed environment. In spite of these obstacles, however, the future of distributed database systems seems bright, aided by the continuing growth in popularity of network computing environments.

References

R. Dale. Client-server database: Architecture of the future. *Database Programming and Design*, pages 28-37, August 1990.

H. A. Edelstein. Lions, Tigers, and Downsizing. *Database Programming and Design*, pages 39-45, March 1992.

B. Gold-Bernstein. Does Client-Server Equal Distributed Database? *Database Programming and Design*, pages 52-62, September 1990.

M. Krasowski. Why Choose A Distributed Database? *Database Programming and Design*, pages 46-53, March 1991.

D. McGoveran and C. J. White. Clarifying Client-Server. *DBMS*, pages 78-90, November 1990.

M. T. \"{O}zsu and P. Valdurie. Distributed Database Systems: Where Are We Now? *Computer*, 24(8): 68-78, August 1991.

S. Ram. Heterogeneous Distributed Database Systems. *Computer*, 24(12): 7-10, Decembet 1991.

A. Silberschatz and M. Stonebraker and J. Ullman. Database Systems: Achievements and Opportunities. *Communications of the ACM*, 34(10): 110-120, October 1991.

1. Presented at the IASSIST 92 Conference held in Madison, Wisconsin, U.S.A. May 26 - 29, 1992.

2. In the computer science literature, "data" is invariably treated as singular, rather than plural. This convention will be followed in this paper.}

3. Workstation" here refers to a desktop computer, usually intended for single user operation, that features a faster processor, more memory, and a larger screen than a PC. Most workstations are UNIX-based and offer elaborate graphical user interfaces

4. This is known as the "read one, write all", or "ROWA", protocol.