Distributed and Parallel Database Systems

M. Tamer Özsu
Department of Computing Science
University of Alberta
Edmonton, Canada T6G 2H1

Patrick Valduriez
INRIA, Rocquencourt
78153 LE Chesnay Cedex
France

Introduction

The maturation of database management system (DBMS) technology has coincided with significant developments in distributed computing and parallel processing technologies. The end result is the emergence of **distributed database management systems** and **parallel database management systems**. These systems have started to become the dominant data management tools for highly data-intensive applications.

The integration of workstations in a distributed environment enables a more efficient function distribution in which application programs run on workstations, called application servers, while database functions are handled by dedicated computers, called database servers. This has led to the present trend in distributed system architecture, where sites are organized as specialized servers rather than as general-purpose computers.

A parallel computer, or multiprocessor, is itself a distributed system made of a number of nodes (processors and memories) connected by a fast network within a cabinet. Distributed database technology can be naturally revised and extended to implement parallel database systems, i.e., database systems on parallel computers [DeWitt and Gray, 1992, Valduriez, 1993]. Parallel database systems exploit the parallelism in data management [Boral, 1988] in order to deliver high-performance and high-availability database servers at a much lower price than equivalent mainframe computers [DeWitt and Gray, 1992, Valduriez, 1993].

In this paper, we present an overview of the distributed DBMS and parallel DBMS technologies, highlight the unique characteristics of each, and indicate the similarities between them. This discussion should help establish their unique and complementary roles in data management.

Underlying Principles

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 then defined as the software system that permits the management of the distributed database and makes the distribution transparent to the users [Özsu and Valduriez, 1991a]. These definitions point to two identifying architectural principles. The first is that the 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 routines in order to facilitate the data access at data sites. The second is that each site (query or data) is assumed to logically consist of a single, independent computer. Therefore, each site has its own primary and secondary storage, runs its own operating system (which may be the same or different at different sites), and has the capability to execute applications on its own. The sites are interconnected by a computer network rather than a multiprocessor configuration. The important point here is the emphasis on loose interconnection between processors which have their own operating systems and operate independently.

The database is physically distributed across the data sites by **fragmenting** and **replicating** the data [Ceri et al., 1987]. Given a relational database schema, fragmentation subdivides each relation into horizontal or vertical partitions. Horizontal fragmentation of a relation is accomplished by a selection operation which places each tuple of the relation in a different partition based on a fragmentation predicate (e.g., an `Employee` relation may be fragmented according to the location of the employees). Vertical fragmentation, divides a relation into a number of fragments by projecting over its attributes (e.g., the `Employee` relation may be fragmented such that the `Emp_number`, `Emp_name` and `Address` information is in one fragment, and `Emp_number`, `Salary` and `Manager` information is in another fragment). Fragmentation is desirable because it enables the placement of data in close proximity to its place of use, thus potentially reducing transmission cost, and it reduces the size of relations that are involved in user queries.

Based on the user access patterns, each of the fragments may also be replicated. This is preferable when the same data are accessed from applications that run at a number of sites. In this case, it may be more cost-effective to duplicate the data at a number of sites rather than continuously moving it between them.

When the above architectural assumptions of a distributed DBMS are relaxed, one gets a parallel database system. The differences between a parallel DBMS and a distributed DBMS are somewhat unclear. In particular, shared-nothing parallel DBMS architectures, which we discuss below, are quite similar to the loosely interconnected distributed systems. Parallel DBMSs exploit recent multiprocessor computer architectures in order to build high-performance and high-availability database servers at a much lower price than equivalent mainframe computers.

A parallel DBMS can be defined as a DBMS implemented on a multiprocessor computer. This includes many alternatives ranging from the straightforward porting of an existing DBMS, which may require only rewriting the operating system interface routines, to a sophisticated combination of parallel processing and database system functions into a new hardware/software architecture. As always, we have the traditional trade-off between portability (to several platforms) and efficiency. The sophisticated approach is better able to fully exploit the opportunities offered by a multiprocessor at the expense of portability.

The solution, therefore, is to use large-scale parallelism to magnify the raw power of individual components by integrating these in a complete system along with the appropriate parallel database software. Using standard hardware components is essential in order to exploit the continuing technological improvements with minimal delay. Then, the database software can exploit the three forms of parallelism inherent in data-intensive application workloads. **Inter-query parallelism** enables the parallel execution of multiple queries generated by concurrent transactions. **Intra-query parallelism** makes the parallel execution of multiple, independent operations (e.g., select operations) possible within the same query. Both inter-query and intra-query parallelism can be obtained by using data partitioning, which is similar to horizontal fragmentation. Finally, with **intra-operation parallelism**, the same operation can be executed as many sub-operations using function partitioning in addition to data partitioning. The set-oriented mode of database languages (e.g., SQL) provides many opportunities for intra-operation parallelism.

There are a number of identifying characteristics of the distributed and parallel DBMS technology.

1. The distributed/parallel database is a database, not some "collection" of files that can be individually stored at each node of a computer network. This is the distinction between a DDB and a collection of files managed by a distributed file system. To form a DDB, distributed data should be logically related, where the relationship is defined according to some structural formalism (e.g., the relational model), and access to data should be at a high level via a common interface.

2. The system has the full functionality of a DBMS. It is neither, as indicated above, a distributed file system, nor is it a transaction processing system. Transaction processing is only one of the functions provided by such a system, which also provides functions such as query processing, structured organization of data, and others that transaction processing systems do not necessarily deal with.

3. The distribution (including fragmentation and replication) of data across multiple site/processors is not visible to the users. This is called **transparency**. The distributed/parallel 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. This is provided by several forms of transparency: network (and, therefore, distribution) transparency, replication transparency, and fragmentation transparency. Transparent access means that users are provided with a single logical image of the database even though it may be physically distributed, enabling them to access the distributed database as if it were a centralized one. In its ideal form, full transparency would imply a query language interface to the distributed/parallel DBMS which is no different from that of a centralized DBMS.

   Transparency concerns are more pronounced in the case of distributed DBMSs. There are a two fundamental reasons for this. First of all, the multiprocessor system on which a parallel DBMS is implemented is controlled by a single operating system. Therefore, the operating system can be structured to implement some aspects of DBMS functionality thereby providing some degree of transparency. Secondly, software development on parallel systems is supported by parallel programming languages which can provide further transparency.

In a distributed DBMS, data and the applications that access that data can be localized at the same site, eliminating (or reducing) the need for remote data access that is typical of teleprocessing-based timesharing systems. Furthermore, since each site handles fewer applications and a smaller portion of the database, contention for resources and for data access can be reduced. Finally, the inherent parallelism of distributed systems provides the possibility of inter-query parallelism and intra-query parallelism.

If the user access to the distributed database consists only of querying (i.e., read-only access), then provision of inter-query and intra-query parallelism would imply that as much of the database as possible should be replicated. However, since most database accesses are not read-only, the mixing of read and update operations requires support for distributed transactions (as discussed in a later section).

Higher performance is probably the most important objective of parallel DBMSs. In these systems, higher performance can be obtained through several complementary solutions: database-oriented operating system support, parallelism, optimization, and load balancing. Having the operating system constrained and "aware" of the specific database requirements (e.g., buffer management) simplifies the implementation of low-level database functions and therefore decreases their cost. For instance, the cost of a message can be significantly reduced to a few hundred instructions by specializing the communication protocol. Parallelism can increase throughput (using inter-query parallelism) and decrease transaction response times (using intra-query and intra-operation parallelism).

Distributed and parallel DBMSs are intended to improve reliability, since they have replicated components and thus eliminate single points of failure. The failure of a single site or processor, or the failure of a communication link which makes one or more sites unreachable, is not sufficient to bring down the entire system. This means that although some of the data may be unreachable, with proper system design users may be permitted to access other parts of the distributed database. The "proper system design" comes in the form of support for distributed transactions. Providing transaction support requires the implementation of distributed concurrency control and distributed reliability (commit and recovery) protocols, which are reviewed in a later section.

In a distributed or parallel environment, it should be easier to accommodate increasing database sizes or increasing performance demands. Major system overhauls are seldom necessary; expansion can usually be handled by adding more processing and storage power to the system.

Ideally, a parallel DBMS (and to a lesser degree a distributed DBMS) should demonstrate two advantages: **linear scaleup** and **linear speedup**. Linear scaleup refers to a sustained performance for a linear increase in both database size and processing and storage power. Linear speedup refers to a linear increase in performance for a constant database size, and a linear increase in processing and storage power. Furthermore, extending the system should require minimal re-organization of the existing database.

The price/performance characteristics of microprocessors and workstations make it more economical to put together a system of smaller computers with the equivalent power of a single big machine. Many commercial distributed DBMSs operate on minicomputers and workstations in order to take advantage of their favorable price/performance characteristics. The current reliance on workstation technology has come about because most of the commercial distributed DBMSs operate within local area networks for which the workstation technology is most suitable. The emergence of distributed DBMSs that run on wide-area networks may increase the importance of mainframes. On the other hand, future distributed DBMSs may support hierarchical organizations where sites consist of clusters of computers communicating over a local area network with a high-speed backbone wide area network connecting the clusters.

Distributed and Parallel Database Technology

Distributed and parallel DBMSs provide the same functionality as centralized DBMSs except in an environment where data is distributed across the sites on a computer network or across the nodes of a multiprocessor system. As discussed above, the users are unaware of data distribution. Thus, these systems provide the users with a logically integrated view of the physically distributed database. Maintaining this view places significant challenges on system functions. We provide an overview of these new challenges in this section. We assume familiarity with basic database management techniques.

Architectural Issues

There are many possible distribution alternatives. The currently popular **client/server architecture** [Orfali et al., 1994], where a number of client machines access a single database server, is the most straightforward one. In these systems, which can be called multiple-client/single-server, the database management problems are considerably simplified since the database is stored on a single server. The pertinent issues relate to the management of client buffers and the caching of data and (possibly) locks. The data management is done centrally at the single server.

A more distributed and more flexible architecture is the multiple-client/multiple server architecture where the database is distributed across multiple servers which have to communicate with each other in responding to user queries and in executing transactions. Each client machine has a "home" server to which it directs user requests. The communication of the servers among themselves is transparent to the users. Most current database management systems implement one or the other type of the client-server architectures.

A truly distributed DBMS does not distinguish between client and server machines. Ideally, each site can perform the functionality of a client and a server. Such architectures, called peer-to-peer, require sophisticated protocols to manage the data distributed across multiple sites. The complexity of required software has delayed the offering of peer-to-peer distributed DBMS products.

Parallel system architectures range between two extremes, the **shared-nothing** and the **shared-memory** architectures. A useful intermediate point is the **shared-disk** architecture.

In the shared-nothing approach, each processor has exclusive access to its main memory and disk unit(s). Thus, each node can be viewed as a local site (with its own database and software) in a distributed database system. The difference between shared-nothing parallel DBMSs and distributed DBMSs is basically one of implementation platform, therefore most solutions designed for distributed databases may be re-used in parallel DBMSs. In addition, shared-nothing architecture has three main virtues: cost, extensibility, and availability. On the other hand, it suffers from higher complexity and (potential) load balancing problems.

Examples of shared-nothing parallel database systems include the Teradata's DBC and Tandem's NonStopSQL products as well as a number of prototypes such as BUBBA [Boral et al., 1990], EDS [EDS, 1990], GAMMA [DeWitt et al., 1990], GRACE [Fushimi et al., 1986], PRISMA [Apers et al., 1992] and ARBRE [Lorie et al., 1989].

In the shared-memory approach, any processor has access to any memory module or disk unit through a fast interconnect (e.g., a high-speed bus or a cross-bar switch). Several new mainframe designs such as the IBM3090 or Bull's DPS8, and symmetric multiprocessors such as Sequent and Encore, follow this approach. Shared-memory has two strong advantages: simplicity and load balancing. These are offset by three problems: cost, limited extensibility, and low availability.

Examples of shared-memory parallel database systems include XPRS [Stonebraker et al., 1988], DBS3 [Bergsten et al., 1991], and Volcano [Graefe, 1990], as well as portings of major RDBMSs on shared-memory multiprocessors. In a sense, the implementation of DB2 on an IBM3090 with 6 processors was the first example. All the shared-memory commercial products (e.g., INGRES and ORACLE) today exploit inter-query parallelism only (i.e., no intra-query parallelism).

In the shared-disk approach, any processor has access to any disk unit through the interconnect, but exclusive (non-shared) access to its main memory. Each processor can then access database pages on the shared disk and copy them into its own cache. To avoid conflicting accesses to the same pages, global locking and protocols for the maintenance of cache coherency are needed. Shared-disk has a number of advantages: cost, extensibility, load balancing, availability, and easy migration from uniprocessor systems. On the other hand, it suffers from higher complexity and potential performance problems.

Examples of shared-disk parallel DBMS include IBM's IMS/VS Data Sharing product and DEC's VAX DBMS and Rdb products. The implementation of ORACLE on DEC's VAXcluster and NCUBE computers also uses the shared-disk approach since it requires minimal extensions of the RDBMS kernel. Note that all these systems exploit inter-query parallelism only.

Query Processing and Optimization

Query processing is the process by which a declarative query is translated into low-level data manipulation operations. SQL 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: <u>query decomposition</u> and <u>query optimization</u>. 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. Simplification involves the elimination of redundant predicates which may be introduced as a result of query modification to deal with views, security enforcement and semantic integrity control. The simplified query is then restructured as an algebraic query.

For a given SQL query, there are more than one possible algebraic queries. Some of these algebraic queries are "better" than others. The quality of an algebraic query is defined in terms of expected performance. The traditional procedure is to obtain an initial algebraic query by translating the predicates and the target statement into relational operations as they appear in the query. This initial algebraic query is then transformed, using algebraic transformation rules, into other algebraic queries until the "best" one is found. The "best" algebraic query is determined according to a cost function which calculates the cost of executing the query according to that algebraic specification. This is the process of query optimization.

In distributed DBMSs, two more steps are involved between query decomposition and query optimization: <u>data localization</u> and <u>global query optimization</u>.

The input to data localization is the initial algebraic query generated by the query decomposition step. The initial algebraic query is specified on global relations irrespective of their fragmentation or distribution.

The main role of data localization is to localize the query's data using data 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 relations. As indicated earlier, fragmentation is defined through fragmentation rules which can be expressed as relational operations (horizontal fragmentation by selection, vertical fragmentation by projection). A distributed relation can be reconstructed by applying the inverse of the fragmentation rules. This is called a localization program. The localization program for a horizontally (vertically) fragmented query is the union (join) of the fragments. Thus, during the data localization step each global relation is first replaced by its localization program, and then the resulting fragment query is simplified and restructured to produce another "good" query. Simplification and restructuring may be done according to the same rules used in the decomposition step. As in the decomposition step, the final fragment query is generally far from optimal; the process has only eliminated "bad" algebraic queries.

The input to the third step is a fragment query, that is, an algebraic query on fragments. The goal of query optimization is to find an execution strategy for the query which is close to optimal. Remember that finding the optimal solution is computationally intractable. An execution strategy for a distributed query can be described with relational algebra operations and communication primitives (send/receive operations) for transferring data between sites. The previous layers have already optimized the query — for example, by eliminating redundant expressions. However, this optimization is independent of fragment characteristics such as cardinalities. In addition, communication operations are not yet specified. By permuting the ordering of operations within one fragment query, many equivalent query execution plans may be found. Query optimization consists of finding the "best" one among candidate plans examined by the optimizer[1]. The query optimizer is usually seen as three components: a search space, a cost model, and a search strategy. The search space is the set of alternative execution plans to represent the input query. These plans are equivalent, in the sense that they yield the same result but they differ on the execution order of operations and the way these operations are implemented. The *cost model* predicts the cost of a given execution plan. To be accurate, the cost model must have accurate knowledge about the parallel execution environment. The search strategy explores the search space and selects the best plan. It defines which plans are examined and in which order.

In a distributed environment, the cost function, often defined in terms of time units, refers to computing resources such as disk space, disk I/Os, buffer space, CPU cost, communication cost, and so on. Generally, it is a weighted combination of I/O, CPU, and communication costs. Nevertheless, a typical simplification made by distributed DBMSs is to consider communication cost as the most significant factor. This is valid for wide area networks, where the limited bandwidth makes communication much more costly than it is in local processing. To select the ordering of operations it is necessary to predict execution costs of alternative candidate orderings. Determining execution costs before query execution (i.e., static optimization) is based on fragment statistics and the formulas for estimating the cardinalities of results of relational operations. Thus the optimization decisions depend on the available statistics on fragments. An important aspect of query optimization is join ordering, since permutations of the joins within the query may lead to improvements of several orders of magnitude. One basic technique for optimizing a sequence of distributed join operations is through use of the semijoin operator. The main value of the semijoin in a distributed system is to reduce the size of the join operands and thus the communication cost. However, more recent techniques, which consider local processing costs as well as communication costs, do not use semijoins because they might increase local processing costs. The output of the query optimization layer is an optimized algebraic query with communication operations included on fragments.

Parallel query optimization exhibits similarities with distributed query processing. It takes advantage of both intra-operation parallelism, which was discussed earlier, and inter-operation parallelism.

[1]The difference between an optimal plan and the best plan is that the optimizer does not, because of computational intractability, examine all of the possible plans.

Intra-operation parallelism is achieved by executing an operation on several nodes of a multiprocessor machine. This requires that the operands have been previously partitioned, i.e., horizontally fragmented, across the nodes. The way in which a base relation is partitioned is a matter of physical design. Typically, partitioning is performed by applying a hash function on an attribute of the relation, which will often be the join attribute. The set of nodes where a relation is stored is called its home. The home of an operation is the set of nodes where it is executed and it must be the home of its operands in order for the operation to access its operands. For binary operations such as join, this might imply repartitioning one of the operands. The optimizer might even sometimes find that repartitioning both the operands is useful. Parallel optimization to exploit intra-operation parallelism can make use of some of the techniques devised for distributed databases.

Inter-operation parallelism occurs when two or more operations are executed in parallel, either as a dataflow or independently. We designate as dataflow the form of parallelism induced by pipelining. Independent parallelism occurs when operations are executed at the same time or in arbitrary order. Independent parallelism is possible only when the operations do not involve the same data.

Concurrency Control

Whenever multiple users access (read and write) a shared database, these accesses need to be synchronized to ensure database consistency. The synchronization is achieved by means of **concurrency control algorithms** which enforce a correctness criterion such as **serializability**. User accesses are encapsulated as **transactions** [Gray, 1981], whose operations at the lowest level are a set of read and write operations to the database. Concurrency control algorithms enforce the **isolation** 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.

The most popular concurrency control algorithms are **locking**-based. In such schemes, a lock, in either shared or exclusive mode, is placed on some unit of storage (usually a page) whenever a transaction attempts to access it. These locks are placed according to lock compatibility rules such that read-write, write-read, and write-write conflicts are avoided. It is a well known theorem that if lock actions on behalf of concurrent transactions obey a simple rule, then it is possible to ensure the serializability of these transactions: "No lock on behalf of a transaction should be set once a lock previously held by the transaction is released." This is known as **two-phase locking** [Gray, 1979], since transactions go through a growing phase when they obtain locks and a shrinking phase when they release locks. In general, releasing of locks prior to the end of a transaction is problematic. Thus, most of the locking-based concurrency control algorithms are strict in that they hold on to their locks until the end of the transaction.

In distributed DBMSs, the challenge is to extend both the serializability argument and the concurrency control algorithms to the distributed execution environment. In these systems, the operations of a given transaction may execute at multiple sites where they access data. In such a case, the serializability argument is more difficult to specify and enforce. The complication is due to the fact that the serialization order of the same set of transactions may be different at different sites. Therefore, the execution of a set of distributed transactions is serializable if and only if

1. the execution of the set of transactions at each site is serializable, and

2. the serialization orders of these transactions at all these sites are identical.

Distributed concurrency control algorithms enforce this notion of global serializability. In locking-based algorithms there are three alternative ways of enforcing global serializability: centralized locking, primary copy locking, and distributed locking algorithm.

In centralized locking, there is a single lock table for the entire distributed database. This lock table is placed, at one of the sites, under the control of a single lock manager. The lock manager is responsible for setting and releasing locks on behalf of transactions. Since all locks are managed at one site, this is similar to centralized concurrency control and it is straightforward to enforce the global serializability rule. These algorithms are simple to implement, but suffer from two problems. The central site may become a bottleneck, both because of the amount of work it is expected to perform and because of the traffic that is generated around it; and the system may be less reliable since the failure or inaccessibility of the central site would cause system unavailability.

Primary copy locking is a concurrency control algorithm that is useful in replicated databases where there may be multiple copies of a data item stored at different sites. One of the copies is designated as a primary copy and it is this copy that has to be locked in order to access that item. The set of primary copies for each data item is known to all the sites in the distributed system, and the lock requests on behalf of transactions are directed to the appropriate primary copy. If the distributed database is not replicated, copy locking degenerates into a distributed locking algorithm. Primary copy locking was proposed for the prototype distributed version of INGRES.

In distributed (or decentralized) locking, the lock management duty is shared by all the sites in the system. The execution of a transaction involves the participation and coordination of lock managers at more than one site. Locks are obtained at each site where the transaction accesses a data item. Distributed locking algorithms do not have the overhead of centralized locking ones. However, both the communication overhead to obtain all the locks, and the complexity of the algorithm are greater. Distributed locking algorithms are used in System R* and in NonStop SQL.

One side effect of all locking-based concurrency control algorithms is that they cause **deadlocks**. The detection and management of deadlocks in a distributed system is difficult. Nevertheless, the relative simplicity and better performance of locking algorithms make them more popular than alternatives such as timestamp-based algorithms or optimistic concurrency control. Timestamp-based algorithms execute the conflicting operations of transactions according to their timestamps which are assigned when the transactions are accepted. Optimistic concurrency control algorithms work from the premise that conflicts among transactions are rare and proceed with executing the transactions up to their termination at which point a validation is performed. If the validation indicates that serializability would be compromised by the successful completion of that particular transaction, then it is aborted and restarted.

Reliability Protocols

We indicated earlier that distributed DBMSs are potentially more reliable because there are multiples of each system component, which eliminates single points of failure. This requires careful system design and the implementation of a number of protocols to deal with system failures.

In a distributed DBMS, four types of failures are possible: transaction failures, site (system) failures, media (disk) failures and communication line failures. Transactions can fail for a number of reasons. Failure can be due to an error in the transaction caused by input data, as well as the detection of a present or potential deadlock. The usual approach to take in cases of transaction failure is to abort the transaction, resetting the database to its state prior to the start of the database.

Site (or system) failures are due to a hardware failure (eg, processor, main memory, power supply) or a software failure (bugs in system or application code). The effect of system failures is the loss of main memory contents. Therefore, any updates to the parts of the database that are in the main memory buffers (also called **volatile database**) are lost as a result of system failures. However, the database that is stored in secondary storage (also called **stable database**) is safe and correct. To achieve this, DBMSs typically employ **logging protocols**, such as Write-Ahead Logging, which record changes to the database in system logs and move these log records and the volatile database pages to stable storage at appropriate times. From the perspective of distributed transaction execution, site failures are important since the failed sites cannot participate in the execution of any transaction.

Media failures refer to the failure of secondary storage devices that store the stable database. Typically, these failures are addressed by duplexing storage devices and maintaining archival copies of the database. Media failures are frequently treated as problems local to one site and therefore are not specifically addressed in the reliability mechanisms of distributed DBMSs.

The three types of failures described above are common to both centralized and distributed DBMSs. Communication failures, on the other hand, are unique to distributed systems. There are a number of types of communication failures. The most common ones are errors in the messages, improperly ordered messages, lost (or undelivered) messages, and line failures. Generally, the first two of these are considered to be the responsibility of the computer network protocols and are not addressed by the distributed DBMS. The last two, on the other hand, have an impact on the distributed DBMS protocols and, therefore, need to be considered in the design of these protocols. If one site is expecting a message from another site and this message never arrives, this may be because (a) the message is lost, (b) the line(s) connecting the two sites may be broken, or (c) the site which is supposed to send the message may have failed. Thus, it is not always possible to distinguish between site failures and communication failures. The waiting site simply timeouts and has to assume that the other site is incommunicado. Distributed DBMS protocols have to deal with this uncertainty. One drastic result of line failures may be network partitioning in which the sites form groups where communication within each group is possible but communication across groups is not. This is difficult to deal with in the sense that it may not be possible to make the database available for access while at the same time guaranteeing its consistency.

Two properties of transactions are maintained by reliability protocols: **atomicity** and **durability**. Atomicity requires that either all the operations of a transaction are executed or none of them are (all-or-nothing). Thus, the set of operations contained in a transaction is treated as one atomic unit. Atomicity is maintained even in the face of failures. Durability requires that the effects of successfully completed (i.e., committed) transactions endure subsequent failures.

The enforcement of atomicity and durability requires the implementation of atomic commitment protocols and distributed recovery protocols. The most popular atomic commitment protocol is **two-phase commit**. The recoverability protocols are built on top of the local recovery protocols, which are dependent upon the supported mode of interaction (of the DBMS) with the operating system.

Two-phase commit (2PC) is a very simple and elegant protocol that ensures the atomic commitment of distributed transactions. It extends the effects of local atomic commit actions to distributed transactions by insisting that all sites involved in the execution of a distributed transaction agree to commit the transaction before its effects are made permanent (i.e., all sites terminate the transaction in the same manner). If all the sites agree to commit a transaction then all the actions of the distributed transaction take effect; if one of the sites declines to commit the operations at that site, then all of the other sites are required to abort the transaction. Thus, the fundamental 2PC rule states:

1. If even one site rejects to commit (which means it votes to abort) the transaction, the distributed transaction has to be aborted at each site where it executes; and

2. If all the sites vote to commit the transaction, the distributed transaction is committed at each site where it executes.

The simple execution of the 2PC protocol is as follows. There is a coordinator process at the site where the distributed transaction originates, and participant processes at all the other sites where the transaction executes. Initially, the coordinator sends a “prepare” message to all the participants each of which independently determines whether or not it can commit the transaction at that site. Those that can commit send back a “vote-commit” message while those who are not able to commit send back a “vote-abort” message. Once a participant registers its vote, it cannot change it. The coordinator collects these messages and determines the fate of the transaction according to the 2PC rule. If the decision is to commit, the coordinator sends a "global-commit" message to all the participants; if the decision is to abort, it sends a "global-abort" message to those participants who had earlier voted to commit the transaction. No message needs to be sent to those participants who had originally voted to abort since they can assume, according to the 2PC rule, that the transaction is going to be eventually globally aborted. This is known as the unilateral abort option of the participants.

There are two rounds of message exchanges between the coordinator and the participants; hence the name 2PC protocol. There are a number of variations of 2PC, such as the linear 2PC and distributed 2PC, that have not found much favor among distributed DBMS vendors. Two important variants of 2PC are the presumed abort 2PC and presumed commit 2PC [Mohan and Lindsay, 1983]. These are important because they reduce the message and I/O overhead of the protocols. Presumed abort protocol is included in the X/Open XA standard and has been adopted as part of the ISO standard for Open Distributed Processing.

One important characteristic of 2PC protocol is its blocking nature. Failures can occur during the commit process. As discussed above, the only way to detect these failures is by means of a time-out of the process waiting for a message. When this happens, the process (coordinator or participant) that timeouts follows a **termination protocol** to determine what to do with the transaction that was in the middle of the commit process. A non-blocking commit protocol is one whose termination protocol can determine what to do with a transaction in case of failures under any circumstance. In the case of 2PC, if a site failure occurs at the coordinator site and one participant site while the coordinator is collecting votes from the participants, the remaining participants cannot determine the fate of the transaction among themselves, and they have to remain blocked until the coordinator or the failed participant recovers. During this period, the locks that are held by the transaction cannot be released, which reduces the availability of the database.

Assume that a participant timeouts after it sends its commit vote to the coordinator, but before it receives the final decision. In this case, the participant is said to be in READY state. The termination protocol for the participant is as follows. First, note that the participant cannot unilaterally reach a termination decision. Since it is in the READY state, it must have voted to commit the transaction. Therefore, it cannot now change its vote and unilaterally abort it. On the other hand, it cannot unilaterally decide to commit the transaction since it is possible that another participant may have voted to abort it. In this case the participant will remain blocked until it can learn from someone (either the coordinator or some other participant) the ultimate fate of the transaction. If we consider a centralized communication structure where the participants cannot communicate with one another, the participant that has timeout has to wait for the coordinator to report the final decision regarding the transaction. Since the coordinator has failed, the participant will remain blocked. In this case, no reasonable termination protocol can be designed.

If the participants can communicate with each other, a more distributed termination protocol may be developed. The participant that timeouts can simply ask all the other participants to help it reach a decision. If during termination all the participants realize that only the coordinator site has failed, they can elect a new coordinator which can restart the commit process. However, in the case where both a participant site and the coordinator site have failed, it is possible for the failed participant to have received the coordinator's decision and terminated the transaction accordingly. This decision is unknown to the other participants; thus if they elect a new coordinator and proceed, there is the danger that they may decide to terminate the transaction differently from the participant at the failed site. The above case demonstrates the blocking nature of 2PC. There have been attempts to devise non-blocking commit protocols (e.g., three-phase commit) but the high overhead of these protocols has precluded their adoption.

The inverse of termination is recovery. When the failed site recovers from the failure, what actions does it have to take to recover the database at that site to a consistent state? This is the domain of distributed recovery protocols. Consider the recovery side of the case discussed above, in which the coordinator site recovers and the recovery protocol has to now determine what to do with the distributed transaction(s) whose execution it was coordinating. The following cases are possible:

1. The coordinator failed before it initiated the commit procedure. Therefore, it will start the commit process upon recovery.

2. The coordinator failed while in the READY state. In this case the coordinator has sent the "prepare" command. Upon recovery, the coordinator will restart the commit process for the transaction from the beginning by sending the "prepare" message one more time. If the participants had already terminated the transaction, they can inform the coordinator. If they were blocked, they can now resend their earlier votes and resume the commit process.

3. The coordinator failed after it informed the participants of its global decision and terminated the transaction. Thus, upon recovery, it does not need to do anything.

Replication Protocols

In replicated distributed databases[2], each logical data item has a number of physical instances. For example, the salary of an employee (logical data item) may be stored at three sites (physical copies). 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.

If replication transparency is maintained, transactions will issue read and write operations on a logical data item x. The replica control protocol is responsible for mapping operations on x to operations on physical copies of x ($x_1, \ldots, x_n$). A typical replica control protocol that enforces one copy serializability is known as **Read-Once/Write-All** (ROWA) protocol. ROWA maps each read on x [Read(x)] to a read on one of the physical copies x_i [Read(x_i)]. The copy which is read is insignificant from the perspective of the replica control protocol and may be determined by performance considerations. On the other hand, each write on logical data item x is mapped to a set of writes on *all* copies of x.

ROWA protocol is simple and straightforward, but it requires that all copies of all logical data items that are updated by a transaction be accessible for the transaction to terminate. 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.

This idea of possibly updating only a subset of the copies, but nevertheless successfully terminating the transaction, has formed the basis of quorum-based voting for replica control protocols. The majority consensus algorithm can be viewed from a slightly different perspective: It assigns equal votes to each copy and a transaction that updates that logical data item can successfully complete as long as it has a majority of the votes. Based on this idea, an early **quorum-based voting algorithm** [Gifford, 1979] assigns a (possibly unequal) vote to each copy of a replicated data item. Each operation then has to obtain a read quorum (V_r) or a write quorum (V_w) to read or write a data item, respectively. If a given data item has a total of V votes, the quorums have to obey the following rules:

1. $V_r + V_w > V$ (a data item is not read and written by two transactions concurrently, avoiding the read-write conflict);

2. $V_w > V/2$ (two write operations from two transactions cannot occur concurrently on the same data item; avoiding write-write conflict).

[2]Replication is not a significant concern in parallel DBMSs because the data is normally not replicated across multiple processors. Replication may occur as a result of data shipping during query optimization, but this is not managed by the replica control protocols.

The difficulty with this approach is that transactions are required to obtain a quorum even to read data. This significantly and unnecessarily slows down read access to the database. An alternative quorum-based voting protocol that overcomes this serious performance drawback [Abbadi et al., 1985] has also been proposed. However, this protocol makes unrealistic assumptions about the underlying communication system. It requires that failures that change the network's topology are detected by all sites instantaneously, and that each site has a view of the network consisting of all the sites with which it can communicate. In general, communication networks cannot guarantee to meet these requirements. The single copy equivalence replica control protocols are generally considered to be restrictive in terms of the availability they provide. Voting-based protocols, on the other hand, are considered too complicated with high overheads. Therefore, these techniques are not used in current distributed DBMS products. More flexible replication schemes have been investigated where the type of consistency between copies is under user control. A number of replication servers have been developed or are being developed with this principle. Unfortunately, there is no clear theory that can be used to reason about the consistency of a replicated database when the more relaxed replication policies are used. Work in this area is still in its early stages.

Research Issues

Distributed and parallel DBMS technologies have matured to the point where fairly sophisticated and reliable commercial systems are now available. As expected, there are a number of issues that have yet to be satisfactorily resolved. In this section we provide an overview of some of the more important research issues.

Data Placement

In a parallel database system, proper data placement is essential for load balancing. Ideally, interference between concurrent parallel operations can be avoided by having each operation work on an independent dataset. These independent datasets can be obtained by the declustering (horizontal partitioning) of the relations based on a function (hash function or range index) applied to some placement attribute(s), and allocating each partition to a different disk. As with horizontal fragmentation in distributed databases, declustering is useful to obtain inter-query parallelism, by having independent queries working on different partitions; and intra-query-parallelism, by having a query's operations working on different partitions. Declustering can be single-attribute or multi-attribute. In the latter case [Ghandeharizadeh et al., 1992], an exact match query requiring the equality of multi-attributes can be processed by a single node without communication. The choice between hashing or range index for partitioning is a design issue: hashing incurs less storage overhead but provides direct support for exact-match queries only, while range index can also support range queries. Initially proposed for shared-nothing systems, declustering has been shown to be useful for shared-memory designs as well, by reducing memory access conflicts [Bergsten et al., 1991].

Full declustering, whereby each relation is partitioned across all the nodes, causes problems for small relations or systems with large numbers of nodes. A better solution is variable declustering where each relation is stored on a certain number of nodes as a function of the relation size and access frequency [Copeland et al., 1988]. This can be combined with multirelation clustering to avoid the communication overhead of binary operations.

When the criteria used for data placement change to the extent that load balancing degrades significantly, dynamic reorganization is required. It is important to perform such dynamic reorganization on-line (without stopping the incoming of transactions) and efficiently (through parallelism). By contrast, existing database systems perform static reorganization for database tuning [Shasha, 1992]. Static reorganization takes place periodically when the system is idle to alter data placement according to changes in either database size or access patterns. In contrast, dynamic reorganization does not need to stop activities and adapts gracefully to changes. Reorganization should also remain transparent to compiled programs that run on the parallel system. In particular, programs should not be recompiled because of reorganization. Therefore, the compiled programs should remain independent of data location. This implies that the optimizer does not know the actual disk nodes where a relation is stored or where an operation will actually take place. The set of nodes where a relation is stored when a certain operation is to be executed, is called its home. Similarly, the set of nodes where the operation will be executed is called the home of the operation. However, the optimizer needs abstract knowledge of the home (e.g., relation R is hashed on A over 20 nodes) and the run-time system makes the association between the home and the actual nodes.

A serious problem in data placement is how to deal with skewed data distributions which may lead to non-uniform partitioning and negatively affect load balancing. Hybrid architectures with nodes of different memory and processing power can be exploited usefully here. Another solution is to treat non-uniform partitions appropriately, e.g., by further declustering large partitions. Separation between logical and physical nodes is also useful since a logical node may correspond to several physical nodes.

A final complicating factor in data placement is data replication for high availability. A naive approach is to maintain two copies of the same data, a primary and a backup copy, on two separate nodes. However, in case of a node failure, the load of the node having the copy may double, thereby hurting load balancing. To avoid this problem, several high-availability data replication strategies have been proposed and recently compared [Hsiao and DeWitt, 1991]. An interesting solution is Teradata's interleaved declustering which declusters the backup copy on a number of nodes. In failure mode, the load of the primary copy is balanced among the backup copy nodes. However, reconstructing the primary copy from its separate backup copies may be costly. In normal mode, maintaining copy consistency may also be costly. A better solution is Gamma's chained declustering which stores the primary and backup copy on two adjacent nodes. In failure mode, the load of the failed node and the backup nodes are balanced among all remaining nodes by using both primary and backup copy nodes. In addition, maintaining copy consistency is cheaper. An open issue remains how to perform data placement taking into account data replication. As with the fragment allocation in distributed databases, this should be considered an optimization problem.

Network Scaling Problems

The database community does not have a full understanding of the performance implications of all the design alternatives that accompany the development of distributed DBMSs. Specifically, questions have been raised about the scalability of some protocols and algorithms as the systems become geographically distributed [Stonebraker, 1989] or as the number of system components increase [Garcia-Molina and Lindsay, 1990]. Of specific concern is the suitability of the distributed transaction processing mechanisms (i.e., the two-phase locking, and, particularly, two-phase commit protocols) in wide area network-based distributed database systems. As mentioned before, there is a significant overhead associated with these protocols, and implementing them over a slow wide area network may be difficult [Stonebraker, 1989].

Scaling issues are only one part of a more general problem, namely that we do not have a good handle on the role of the network architectures and protocols in the performance of distributed DBMSs. Almost all the performance studies of which we are aware assume a very simple network cost model — sometimes as unrealistic as using a fixed communication delay that is independent of all network characteristics such as load, message size, network size, and so on. In general, the performance of the proposed algorithm and protocols in different local area network architectures is not well understood, nor is their comparative behavior in moving from local area networks to wide area networks. The proper way to deal with scalability issues is to develop general and sufficiently powerful performance models, measurement tools and methodologies. Such work on centralized DBMSs has been going on for some time, but has not yet been sufficiently extended to distributed DBMSs.

Even though there are plenty of performance studies of distributed DBMSs, these usually employ simplistic models, artificial workloads, conflicting assumptions or consider only a few special algorithms. This does not mean that we do not have some understanding of the trade-offs. In fact, certain trade-offs have long been recognized and even the earlier systems have considered them in their design. However, these trade-offs can mostly be spelled out only in qualitative terms; their quantification requires more research on performance models.

Distributed and Parallel Query Processing

As discussed earlier, global query optimization generates an optimal execution plan for the input fragment query by making decisions regarding operation ordering, data movement between sites and the choice of both distributed and local algorithms for database operations. There are a number of problems related to this step. They have to do with the restrictions imposed on the cost model, the focus on a subset of the query language, the trade-off between optimization cost and execution cost, and the optimization/reoptimization interval.

The cost model is central to global query optimization since it provides the necessary abstraction of the distributed DBMS execution system in terms of access methods, as well as an abstraction of the database in terms of physical schema information and related statistics. The cost model is used to predict the execution cost of alternative execution plans for a query. A number of important restrictions are often associated with the cost model, limiting the effectiveness of optimization in improving throughput. Work in extensible query optimization [Freytag, 1987] can be useful in parameterizing the cost model which can then be refined after much experimentation. Even though query languages are becoming increasingly powerful (e.g., new versions of SQL), global query optimization typically focuses on a subset of the query language, namely select-project-join (SPJ) queries with conjunctive predicates. This is an important class of queries for which good optimization opportunities exist. As a result, a good deal of theory has been developed for join and semijoin ordering. However, there are other important queries that warrant optimization, such as queries with disjunctions, unions, fixpoint, aggregations or sorting. A promising solution is to separate the language understanding from the optimization itself which can be dedicated to several optimization "experts."

There is a necessary trade-off between optimization cost and quality of the generated execution plans. Higher optimization costs are probably acceptable to produce "better" plans for repetitive queries, since this would reduce query execution cost and amortize the optimization cost over many executions. However, high optimization cost is unacceptable for ad hoc queries which are executed only once. The optimization cost is mainly incurred by searching the solution space for alternative execution plans. In a distributed system, the solution space can be quite large because of the wide range of distributed execution strategies. Therefore, it is critical to study the application of efficient search strategies that avoid the exhaustive search approach. Global query optimization is typically performed prior to the execution of the query, hence it is called static. A major problem with this approach is that the cost model used for optimization may become inaccurate because of changes in the fragment sizes or the database reorganization which is important for load balancing. The problem, therefore, is to determine the optimal intervals of re-compilation/reoptimization of the queries taking into account the trade-off between optimization and execution cost.

The crucial issue in terms of search strategy is the join ordering problem which is NP-complete in the number of relations [Ibaraki and Kameda, 1984]. A typical approach to solving the problem is to use dynamic programming [Selinger et al., 1979] , which is a deterministic strategy. This strategy is almost exhaustive and assures that the best of all plans is found. It incurs an acceptable optimization cost (in terms of time and space) when the number of relations in the query is small. However, this approach becomes too expensive when the number of relations is greater than 5 or 6. For this reason, there has been recent interest in *randomized* strategies, which reduce the optimization complexity but do not guarantee the best of all plans. Randomized strategies investigate the search space in a way which can be fully controlled such that optimization ends after a given optimization time budget has been reached. Another way to cut off optimization complexity is to adopt a heuristic approach. Unlike deterministic strategies, randomized strategies allow the optimizer to trade optimization time for execution time [Ioannidis and Wong, 1987, Swami and Gupta, 1988, Ioannidis and Kang, 1990].

Distributed Transaction Processing

Despite the research done to date, there are still topics worthy of investigation in the area of distributed transaction processing. We have already discussed the scaling problems of transaction management algorithms. Additionally, replica control protocols, more sophisticated transaction models, and non-serializable correctness criteria require further attention. The field of data replication needs further experimentation; research is required on replication methods for computation and communication; and more work is necessary to enable the systematic exploitation of application-specific properties. Experimentation is required to evaluate the claims that are made by algorithm and system designers, and we lack a consistent framework for comparing competing techniques.

One of the difficulties in quantitatively evaluating replication techniques lies in the absence of commonly accepted failure incidence models. For example, Markov models that are sometimes used to analyze the availability achieved by replication protocols assume the statistical independence of individual failure events, and the rarity of network partitions relative to site failures. Currently, we do not know that either of these assumptions is tenable, nor are we aware how sensitive Markov models are to these assumptions. The validation of the Markov models by simulation requires empirical measurement, since simulations often embody the same assumptions that underlie the Markov analysis. There is a need, therefore, for empirical studies to monitor failure patterns in real-life production systems, with the purpose of constructing a simple model of typical failure loads.

To achieve the twin goals of data replication, namely availability and performance, it is necessary to provide integrated systems in which the replication of data goes hand in hand with the replication of computation and communication (including I/O). Only data replication has been studied intensively; relatively little has been done in the replication of computation and communication.

In addition to replication, and related to it, work is required on more elaborate transaction models, especially those that exploit the semantics of the application [Elmagarmid, 1992, Weihl, 1989]. Greater availability and improved performance, as well as concurrency, can be achieved with such models. As database technology enters new application domains such as engineering design, software development, and office information systems, the nature of and requirements for transactions change. Thus, work is needed on more complicated transaction models as well as on correctness criteria different from serializability.

Complex transaction models are important in distributed systems for a number of reasons. The most important is that the new application domains that distributed DBMSs will support in the future (e.g., engineering design, office information systems, cooperative work, etc.) require transaction models that incorporate more abstract operations that execute on complex data. Furthermore, these applications have a different sharing paradigm than the typical database access to which we are accustomed. For example, computer-assisted cooperative work environments require participants to cooperate in accessing shared resources rather than competing for them, as is usual in typical database applications. These changing requirements necessitate the development of new transaction models and accompanying correctness criteria.

Object-oriented DBMSs are now being investigated as potential candidates for meeting the requirements of such "advanced" applications. These systems encapsulate operations (methods) with data. Therefore, they require a clear definition of their update semantics, and transaction models that can exploit the semantics of the encapsulated operations [Özsu, 1994].

Summary

Distributed and parallel DBMSs have become a reality in the last few years. They provide the functionality of centralized DBMSs, but in an environment where data is distributed over the sites of a computer network or the nodes of a multiprocessor system. Distributed databases have enabled the natural growth and expansion of databases by the simple addition of new machines. The price-performance characteristics of these systems are favorable in part due to the advances in computer network technology. Parallel DBMSs are perhaps the only realistic approach to meet the performance requirements of a variety of important applications which place significant throughput demands on the DBMS. In order to meet these requirements, distributed and parallel DBMSs need to be designed with special consideration for the protocols and strategies. In this article, we provide an overview of these protocols and strategies.

There are a number of related issues that we did not cover. Two important topics that we omitted are multidatabase systems and distributed object-oriented databases. Most information systems evolve independent of other systems, with their own DBMS implementations. Later requirements to "integrate" these autonomous and possibly heterogeneous systems pose significant difficulties. Systems which provide access to independently designed and implemented, and possibly heterogeneous, databases are called multidatabase systems [Sheth and Larson, 1990].

The penetration of database management technology into areas (e.g., engineering databases, multimedia systems, geographic information systems, image databases) which relational database systems were not designed to serve has given rise to a search for new system models and architectures. A primary candidate for meeting the requirements of these systems is the object-oriented DBMS [Dogac et al., 1994]. The distribution of object-oriented DBMSs gives rise to a number of issues generally categorized as distributed object management [Özsu et al., 1994]. We have ignored both multidatabase system and distributed object management issues in this paper.

Defining 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).

Client/server architecture: A distributed/parallel DBMS architecture where a set of client machines with limited functionality access a set of servers which manage data.

Concurrency control algorithm: Algorithms that synchronize the operations of concurrent transactions that execute on a shared database.

Data independence: The immunity of application programs and queries to changes in the physical organization (physical data independence) or logical organization (logical data independence) of the database and vice versa.

Distributed database management system: A database management system that manages a database that is distributed across the nodes of a computer network and makes this distribution transparent to the users.

Deadlock: An occurrence where each transaction in a set of transactions circularly waits on locks that are held by other transactions in the set.

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

Inter-query parallelism: The parallel execution of multiple queries generated by concurrent transactions.

Intra-query parallelism: The parallel execution of multiple, independent operations possible within the same query.

Intra-operation parallelism: The execution of one relational operation as many sub-operations.

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.

Linear scaleup: Sustained performance for a linear increase in both database size and processing and storage power.

Linear speedup: Linear increase in performance for a constant database size and linear increase in processing and storage power.

Locking: A method of concurrency control where locks are placed on database units (e.g., pages) on behalf of transactions that attempt to access them.

Logging protocol: The protocol which records, in a separate location, the changes that a transaction makes to the database before the change is actually made.

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.

Parallel database management system: A database management system that is implemented on a tightly-coupled multiprocessor.

Quorum-based voting algorithm: A replica control protocol where transactions collect votes to read and write copies of data items. They are permitted to read or write data items if they can collect a quorum of votes.

Read-Once/Write-All protocol: The replica control protocol which maps each logical read operation to a read on one of the physical copies and maps a logical write operation to a write on all of the physical copies.

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.

Shared-disk architecture: A parallel DBMS architecture where any processor has access to any disk unit through the interconnect but exclusive (non-shared) access to its main memory.

Shared-memory architecture: A parallel DBMS architecture where any processor has access to any memory module or disk unit through a fast interconnect (e.g., a high-speed bus or a cross-bar switch).

Shared-nothing architecture: A parallel DBMS architecture where each processor has exclusive access to its main memory and disk unit(s).

Stable database: The portion of the database that is stored in secondary storage.

Termination protocol: A protocol by which individual sites can decide how to terminate a particular transaction when they cannot communicate with other sites where the transaction executes.

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.

Two-phase locking: A locking algorithm where transactions are not allowed to request new locks once they release a previously held lock.

Volatile database: The portion of the database that is stored in main memory buffers.

References

[Abbadi et al., 1985] A. E. Abbadi, D. Skeen, and F. Cristian. "An Efficient, Fault–Tolerant Protocol for Replicated Data Management", In *Proc. 4th ACM SIGACT–SIGMOD Symp. on Principles of Database Systems*, Portland, Oreg., March 1985, pp. 215–229.

[Apers et al., 1992] P. Apers, C. van den Berg, J. Flokstra, P. Grefen, M. Kersten, A. Wilschut. "Prisma/DB: a Parallel Main-Memory Relational DBMS", *IEEE Trans. on Data and Knowledge Eng.* (1992), 4(6): 541–554.

[Bell and Grimson, 1992] D. Bell and J. Grimson. *Distributed Database Systems*, Reading, MA: Addison-Wesley, 1993.

[Bergsten et al., 1991] B. Bergsten, M. Couprie, P. Valduriez. "Prototyping DBS3, a Shared-Memory Parallel Database System", In *Proc. Int. Conf. on Parallel and Distributed Information Systems*, Miami, Florida, December 1991, pp 226–234.

[Bernstein et al., 1987] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Reading, Mass.: Addison-Wesley, 1987.

[Boral, 1988] H. Boral. "Parallelism and Data Management", In *Proc. 3rd Int. Conf. on Data and Knowledge Bases*, Jerusalem, June 1988, pp. 362–373.

[Boral et al., 1990] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith and P. Valduriez. "Prototyping Bubba, a Highly Parallel Database System", *IEEE Trans. on Knowledge and Data Engineering* (March 1990), 2(1): 4-24.

[Ceri and Pelagatti, 1984] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems*. New York: McGraw-Hill, 1984.

[Ceri et al., 1987] S. Ceri, B. Pernici, and G. Wiederhold. "Distributed Database Design Methodologies", *Proc. IEEE* (May 1987), 75(5): 533–546.

[Copeland et al., 1988] G. Copeland, W. Alexander, E. Bougherty, and T. Keller. "Data Placement in Bubba", In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Chicago, May 1988, pp. 99–108.

[DeWitt et al., 1990] D.J. DeWitt, S. Ghandeharizadeh, D.A. Schneider, A. Bricker, H.-I Hsiao, and R. Rasmussen. "The GAMMA Database Machine Project", *IEEE Trans. on Knowledge and Data Eng.* (March 1990), 2(1): 44–62.

[DeWitt and Gray, 1992] D. DeWitt and J. Gray. "Parallel Database Systems: The Future of High-Performance Database Systems", *Communications of ACM* (June 1992), 35(6):85–98.

[Dogac et al., 1994] A. Dogac, M.T. Özsu, A. Biliris and T. Sellis (eds.). *Advances in Object-Oriented Database Systems*, Berlin: Springer-Verlag, 1994.

[EDS, 1990] European Declarative System (EDS) Database Group. EDS-Collaborating for a High-Performance Parallel Relational Database. In *Proc. ESPRIT Conf.*, Brussels, November 1990.

[Elmagarmid, 1992] A.K. Elmagarmid (ed.). *Transaction Models for Advanced Database Applications*. San Mateo, CA: Morgan Kaufmann, 1992.

[Freytag et al., 1993] J-C. Freytag, D. Maier, and G. Vossen. *Query Processing for Advanced Database Systems*. San Mateo: Morgan Kaufmann, 1993.

[Freytag, 1987] J-C. Freytag. "A Rule-based View of Query Optimization", In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, San Francisco, 1987, pp 173–180.

[Fushimi et al., 1986] S. Fushimi, M. Kitsuregawa and H. Tanaka. "An Overview of the System Software of a Parallel Relational Database Machine GRACE", In *Proc. 12th Int. Conf. on Very Large Data Bases*, Kyoto, August 1986, pp. 209-219.

[Garcia-Molina and Lindsay, 1990] H. Garcia-Molina and B. Lindsay. "Research Directions for Distributed Databases", *IEEE Q. Bull. Database Eng.* (December 1990), 13(4): 12–17.

[Ghandeharizadeh et al., 1992] S. Ghandeharizadeh, D. DeWitt, W. Quresh., "A Performance Analysis of Alternative Multi-Attributed Declustering Strategies", *ACM SIGMOD Int. Conf. on Management of Data*, San Diego, California, June 1992, pp 29–38.

[Gifford, 1979] D. K. Gifford. "Weighted Voting for Replicated Data", In *Proc. 7th ACM Symp. on Operating System Principles*, Pacific Grove, Calif., December 1979, pp. 150–159.

[Graefe, 1990] G. Graefe. "Encapsulation of Parallelism in the Volcano Query Processing Systems", In *Proc. ACM SIGMOD Int. Conf.*, Atlantic City, NJ, May 1990, pp. 102-111.

[Gray, 1981] J. Gray. "The Transaction Concept: Virtues and Limitations", In *Proc. 7th Int. Conf. on Very Large Data Bases*, Cannes, France, September 1981, pp. 144–154.

[Gray, 1979] J. N. Gray. "Notes on Data Base Operating Systems", In *Operating Systems: An Advanced Course*, R. Bayer, R. M. Graham, and G. Seegmüller (eds.), New York: Springer-Verlag, 1979, pp. 393–481.

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

[Hsiao and DeWitt, 1991] H.-I. Hsiao, D. De Witt. A "Performance Study of three High-Availability Data Replication Strategies", In *Proc. Int. Conf. on Parallel and Distributed Information Systems*, Miami, December 1991, pp. 18–28.

[Ibaraki and Kameda, 1984] T. Ibaraki and T. Kameda. "On the Optimal Nesting Order for Computing *N*-Relation Joins", *ACM Trans. Database Syst.* (September 1984), 9(3): 482–502.

[Ioannidis and Wong, 1987] Y. Ioannidis and E. Wong. "Query Optimization by Simulated Annealing", In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, 1987, pp. 9–22.

[Ioannidis and Kang, 1990] Y. Ioannidis and Y.C. Kang. "Randomized Algorithms for Optimizing Large Join Queries", In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, 1990, pp. 312–321.

[Lorie et al., 1989] R. Lorie, J-J. Daudenarde, G. Hallmark, J. Stamos, H. Young. "Adding Intra-parallelism to an Existing DBMS: Early Experience", *IEEE Bull. on Database Engineering* (March 1989), 12(1): 2–8.

[Mohan and Lindsay, 1983] C. Mohan and B. Lindsay. "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions", In *Proc. 2nd ACM SIGACT–SIGMOD Symp. on Principles of Distributed Computing*, 1983, pp. 76–88.

[Orfali et al., 1994] R. Orfali, D. Harkey and J. Edwards. *Essential Client/Server Survival Guide*, New York, John Wiley, 1994.

[Özsu, 1994] M.T. Özsu. "Transaction Models and Transaction Management in Object-Oriented Database Management Systems", In *Advances in Object-Oriented Database Systems*, A. Dogac, M.T. Özsu, A. Biliris, and T. Sellis (eds.), Berlin: Springer-Verlag, 1994, pp. 147–183.

[Özsu and Valduriez, 1991a] M.T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*, Englewood Cliffs, NJ: Prentice-Hall, 1991.

[Özsu and Valduriez, 1991b] M.T. Özsu and P. Valduriez. "Distributed Database Systems: Where Are We Now?", *IEEE Computer* (August 1991), 24(8): 68–78.

[Özsu et al., 1994] M.T. Özsu, U. Dayal and P. Valduriez (eds.). *Distributed Object Management*, San Mateo: Morgan Kaufmann, 1994

[Selinger et al., 1979] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price. "Access Path Selection in a Relational Database Management System", In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Boston, Mass., May 1979, pp. 23–34.

[Shasha, 1992] D. Shasha. *Database Tuning: a Principled Approach*, Englewood Cliffs, NJ, Prentice Hall, 1992.

[Sheth and Larson, 1990] A. Sheth and J. Larson. "Federated Databases: Architectures and Integration", *ACM Comput. Surv.* (September 1990), 22(3): 183–236.

[Stonebraker, 1989] M. Stonebraker. "Future Trends in Database Systems", *IEEE Trans. Knowledge and Data Eng.* (March 1989) 1(1): 33–44.

[Stonebraker et al., 1988] M. Stonebraker R. Katz, D. Patterson and J. Ousterhout. "The Design of XPRS", In *Proc. 14th Int. Conf. on Very Large Data Bases*, Los Angeles, September 1988, pp. 318–330.

[Swami and Gupta, 1988] A. Swami and A. Gupta. "Optimization of Large Join Queries", In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, 1988, pp.8–17.

[Valduriez, 1993] P. Valduriez. "Parallel Database Systems: Open Problems and New Issues", *Distributed and Parallel Databases* (April 1993), 1(2): 137–165.

[Weihl, 1989] W. Weihl. "Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types", *ACM Trans. Prog. Lang. Syst.* (April 1989), 11(2): 249–281.

Further Information

There are two current textbooks on distributed and parallel databases. One is our book [Özsu and Valduriez, 1991a] and the other book is [Bell and Grimson, 1992]. The first serious book on this topic was [Ceri and Pelagatti, 1984] which is now quite dated. Our paper [Özsu and Valduriez, 1991b], which is a companion to our book, discusses many open problems in distributed databases. Two excellent papers on parallel database systems are [DeWitt and Gray, 1992, Valduriez, 1993].

There are a number of more specific texts. On query processing, [Freytag et al., 1993] provide an overview of many of the more recent research results. [Elmagarmid, 1992] has descriptions of a number of advanced transaction models. [Gray and Reuter, 1993] is an excellent overview of building transaction managers. Another classical textbook on transaction processing is [Bernstein et al., 1987]. These books cover both concurrency control and reliability.