Parallel and distributed databases

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, Valorize, 1993]. Parallel database systems exploit the parallelism in data management [Boreal, 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. Evaluated, 1993]. In this paper, we present an overview of the distributed DB’S and parallel DB’S technologies, highlight the unique characteristics of each, and indicate the similarities between hem.

This discussion should help establish their unique and complementary roles in data management. Underlying Principles A distributed database (DB) is a collection of multiple, logically interrelated databases distributed over a computer network. A distributed database management system (distributed DB’S) Is then defined as the software system that permits the management of the distributed database and makes the distribution ” 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 outrage 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 ere 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 [Cerci 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 duple 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 he 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 MME number, MME name and Address information is in one

fragment, and MME 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 DB’S are relaxed, one gets a parallel database system. The differences between a parallel DB’S and a distributed DB’S are somewhat unclear. In particular, shared-nothing parallel DB’S architectures, which we discuss below, are quite similar to the loosely interconnected distributed systems. Parallel Dobbs 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 DB’S can be defined as a DB’S implemented on a multiprocessor computer. This includes many alternatives ranging from the straightforward porting of an existing DB’S, 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 rotational trade-off between portability (to several platforms) and efficiency.

The sophisticated approach is better able to fully exploit the opportunities offered by a 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 daintiness 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 inadequate 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 DB’S 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 DB and a collection of files managed by a distributed file system. To form a DB, 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 common interface. . The system has the full

functionality of a DB’S. 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. 2 3. The distribution (including fragmentation and replication) of data across multiple site/processors is not visible to the users. This is called transparency.

The strutted/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 DB’S which is no different from that of a centralized DB’S. Transparency concerns are more pronounced in the case of distributed Dobbs. There are a two fundamental reasons for this. First of all, the multiprocessor system on which a parallel DB’S is implemented is controlled by a single operating system. Therefore, the operating system can be structured to implement some aspects of DB’S functionality thereby

providing some degree of transparency. Secondly, languages which can provide further transparency.

In a distributed DB’S, 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. . 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 Dobbs. 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 unstrained 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 Dobbs 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 DB’S (and to a lesser degree a distributed DB’S) should demonstrate two advantages: linear scalpel and linear speedup.

Linear scalpel refers to a sustained performance for a linear increase in both database size and processing and storage power. Linear speedup refers too linear increase in 3 reference for a constant database size, and a linear increase in processing and storage power. Furthermore, extending the system should require minimal re- 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 Dobbs 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 Dobbs operate within local area networks for which the workstation genealogy is most suitable. The emergence of distributed Dobbs that run on wide- area networks may increase the importance of mainframes. On the other hand, future distributed Dobbs 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 Dobbs provide the same functionality as centralized Dobbs 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 [Airfoil 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 ramifications. 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 DB’S 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 allayed the offering of peer-to-peer distributed DB’S 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 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 Dobbs and distributed Dobbs is basically one of implementation platform, therefore most solutions designed for distributed databases may be re-used in parallel Dobbs. 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. 4 Examples of shared-nothing parallel database systems include the Dearest's DB and Tandem's Nonstop products as well as a number of prototypes such as BUBBY [Boreal et al. 1990], DEEDS [DEEDS, 1990], GAMMA [Dewitt et al. , 1990], GRACE [Fishing et al. , 1986], PRISMS [Papers et al. , 1992] and ARBOR [Lone et al. , 1989]. In the shared-memory approach, any processor has access to any memory module or sis unit through a fast interconnect (e. G. , a high-speed bus or a cross-bar switch). Several new mainframe designs such as the BAMBINI or Bull's DIPS, 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 SPARS [Stockbroker et al. , 1988], DB'S [Bergsten et al. , 1991], and Volcano [Garage, 1990], as well as portions of major Orders on shareholders multiprocessors. In a sense, the implementation of DB on an BAMBINI with 6 processors was the first example. All the shared-memory commercial products (e. G. , INGRESS 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 unprocessed systems. On the other hand, it suffers from higher complexity and potential performance problems. Examples of shared-disk parallel DB'S include VIM'S AIMS, VS. Data Shaman product and DECK'S FAX DB'S and 3rd products.

The implementation of ORACLE on Deck's Vacillates and INCISE computers also uses the shared-disk approach since it requires minimal extensions of the READS 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 Dobbs. 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 Dobbs, the process typically involves two steps: query 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 Seibel, 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 reinstating 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 Dobbs, two more steps are involved between query decomposition and query optimization: data localization and global query optimization. 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. 5 The main role of data localization is to localize the query 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 argumentation 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 Onion) 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 “ 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/So, buffer space, CPU cost, immunization cost, and so on. Generally, it is a weighted combination of 1/0, CUP], and communication costs. Nevertheless, a typical simplification made by distributed Dobbs 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 tactics 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 Seminole operator. The main value of the Seminole 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 sessions 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 to, because of computational intractability, examine all of the possible plans. 6 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 e 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 outflow or independently. We designate as outflow 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 serviceability. 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 inflicts 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 serviceability 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 nonoccurrence control algorithms are strict in that they hold on to their locks until the end of the transaction. In distributed Dobbs, the challenge is to extend both the serviceability 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 serviceability 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 serialize if and only if 1 . The execution of the set of transactions at each site is serialize, and 2. The serialization orders of these transactions at all these sites are identical. Distributed concurrency control algorithms enforce this notion of global serviceability. In locking- 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 7 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 serviceability 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 INGRESS. 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 serviceability would be compromised by the successful completion of that particular transaction, then it is aborted and restarted. Reliability Protocols We indicated earlier that distributed Dobbs are potentially more reliable because there are multiples of each system component, which eliminates single points of failure.