PARALLEL DATABASE MANAGEMENT SYSTEMS

A *database* is a collection of data that is managed by a *database management system,* also called a *DBMS*. A DBMS allows users to create a new database by specifying the logical structure of the data. For instance, the world is represented as a collection of tables in relational DBMSs. This model is very simple, but is useful for many applications. It is the model on which the major commercial DBMSs are based today. After a database has been created, the users are allowed to insert new data. They can also query and modify existing data. The DBMS gives them the ability to access the data simultaneously, without allowing the action of one user to affect other users. The DBMS ensures that no simultaneous accesses can corrupt the data accidentally. In this article the reader will learn how parallel processing technology can be used to effectively address the performance bottleneck in DBMSs. After a brief discussion of the various parallel computer architectures suitable for DBMSs, we learn the techniques for organizing data in such machines, and the strategies for processing these data using multiple processors. Finally, we discuss some future directions and research problems.

Modern database management systems (DBMSs) are designed to support the client–server computing model. In this environment, applications running on client computers or workstations are allowed to store and access data from a remote database server. This configuration makes best use of both hardware and software resources. Both the client and database server can be dedicated to the tasks for which they are best suited. This architecture also provides an opportunity for both horizontal (i.e., more servers) and vertical (i.e., larger servers) scaling of resources to do the job.

Today's database servers are generally general-purpose computers running database management software, typically a relational DBMS. These servers employ essentially the same hardware technology used for the client workstations. This approach offers the most cost-effective computing environment for a wide range of applications by leveraging the advances in commodity hardware. A potential pitfall of this approach is that the many equally powerful workstations may saturate the server. The situation is aggravated for applications which involve very large databases and complex queries. To address this problem, designers have relied on parallel processing technologies to build the more powerful database servers (1–4). This solution enables servers to be configured in a variety of ways to support various needs.

PARALLEL DATABASE SERVER ARCHITECTURES

The problem faced by database applications has long been known as I/O limited. The I/O bottleneck sets a hard limitation on the performance of a database server. To address this problem, all parallel database approaches distribute the data across a large number of disks in order to take advantage of their aggregate bandwidth. The different types of parallel database servers are characterized by the way their processors are allowed to share the storage devices. Most existing systems employ one of the three basic parallel architectures (5): shared everything (SE), shared disk (SD), and shared nothing (SN). None emerges as the undisputed winner. Each has its own advantages as well as disadvantages.

Shared Everything

All disks and memory modules are shared by the processors [Fig. 1(a)]. Examples of this architecture include IBM mainframes, HP T500, SGI Challenge, and the symmetric-multiprocessor (SMP) systems available from PC manufacturers. A major advantage of this approach is that interprocessor communication is fast as the processors can cooperate via the shared memory. This system architecture, however, does not scale well to support very large databases. For an SE system with more than 32 processors, the shared memory would have to be a physically distributed memory to accommodate the aggregate demand on the shared memory from the large number of processors. An interconnection network (e.g., multistage network) is needed, in this case, to allow the processors to access the different memory modules simultaneously. As the number of the processors increases, the size of the interconnection network grows accordingly rendering a longer memory access latency. The performance of microprocessors is very sensitive to this factor. If the memory-access latency exceeds one instruction time, the processor may idle until the storage cycle completes. A popular solution to this problem is to have cache memory with each processor. However, the use of caches requires a mechanism to ensure cache coherency. As we increase the number of processors, the number of messages due to cache coherency control (i.e., cross interrogation) increases. Unless this problem can be solved, scaling an SE database server into the range of 64 or more processors will be impractical. Commercial DBMSs designed for this architecture include Informix 7.2 Online Dynamic Server, Oracle 7.3 Parallel Query Option, and IBM DB2/MVS.

Shared Disk

To address the memory-access-latency problem encountered in SE systems, each processor is coupled with its private memory in an SD system [Fig. 1(b)]. The disks are still shared by all processors as in SE. Intel Paragon, nCUBE/2, and Tandem's ServerNet-based machines typify this design. Since each processor may cache data pages in its private memory, SD also suffers the high cost of cache coherency control. In fact the interference among processors is even more severe than in SE. As an example, let us consider a disk page containing 32 cache lines of data. There is no interference in an SE system as long as the processors update different cache lines of this page. In contrast, an update to any of these cache lines in an SD system will interfere with all the processors currently having a copy of this page even when they are actually using different cache lines of the page. Commercial DBMSs designed for this architecture include IBM IMS/VS Data Sharing Product, DEC VAX DBMS and Rdb products, and Oracle on DEC's VAXcluster and Ncube Computers.

Shared Nothing

To improve scalability, SN systems are designed to overcome the drawbacks of SE and SD systems [Fig. 1(c)]. In this configuration, a message-passing network is used to interconnect a large number of processing nodes (PN). Each PN is an autonomous computer consisting of a processor, local private memory, and dedicated disk drives. Memory access latency is no longer a problem. Furthermore, since each processor is only allowed to read and write its local partition of the database, cache coherency is much easier to maintain. SN is not a performance panacea, however. Message passing is significantly more expensive than sharing data through a centralized shared memory as in SE systems. Some examples of this architecture are Teradata's DBC, Tandem NonStopSQL, and IBM 6000 SP. Commercial DBMSs designed for this architecture include Teradata's DBC, Tandem NonStopSQL and IBM DB2 Parallel Edition.

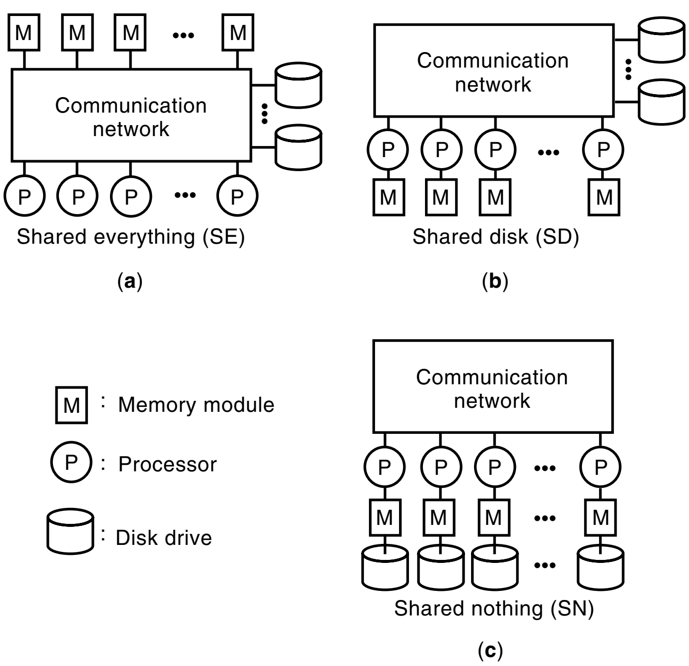

Figure 1. Basic architectures for parallel database servers. Both disks and memory modules are shared by all the processors in SE. Only disks are shared in SD. Neither disks nor memory modules are shared by the processors in SN.

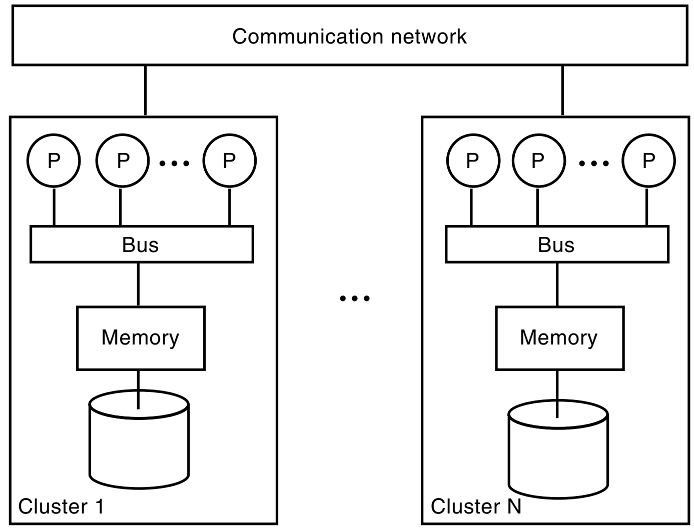

Figure 2. A hybrid architecture for parallel database servers. SE clusters are interconnected to form an SN structure at the intercluster level.

To combine the advantages of the previously discussed architectures and compensate for their respective disadvantages, new parallel database servers are converging toward a hybrid architecture (6), in which SE clusters are interconnected through a communication network to form an SN structure at the intercluster level (Fig. 2). The motivation is to minimize the communication overhead associated with the SN structure, and yet each cluster size is kept small within the limitation of the local memory and I/O bandwidth. Examples of this architecture include new Sequent computers, IBM RS/6000 SP, NCR 5100M and Bull PowerCluster. Some of the commercial DBMSs designed for this structure are the Teradata Database System for the NCR WorldMark 5100 computer, Sybase MPP, and Informix-Online Extended Parallel Server.

DATA PARTITIONING TECHNIQUES

Traditional use of parallel computers is to speed up the complex computation of scientific and engineering applications. In contrast, database applications use parallelism primarily to increase the disk-I/O bandwidth. The level of I/O concurrency achievable determines the degree of parallelism that can be attained. If each relation (i.e., data set) is divided into partitions each stored on a distinct disk, a database operator can often be decomposed into many independent operators each working on one of the partitions. To maximize parallelism, several data partitioning techniques have been used (7).

Round-Robin Partitioning

The tuples (i.e., data records) of a relation are distributed among the disks in a round-robin fashion. The advantages of this approach are simplicity and the balanced data load among the disks. The drawback of this scheme is that it does not support associative search. Any search operations would require searching all the disks in the system. Typically, local indices must be created for each data partition to speed up the local search operations.

Hash Partitioning

A randomizing hash function is applied to the partitioning attribute (i.e., key field) of each tuple in order to determine its disk. Like round-robin partitioning, hash partitioning usually provides an even distribution of data across the disks. However, unlike round-robin partitioning, the same hash function can be employed at runtime to support associative searches. A drawback of hash partitioning is its inability to support range queries. A range query retrieves tuples which have the value of the specified attribute falling within a given range. This type of query is common in many applications.

Range Partitioning

This approach maps contiguous key ranges of a relation to various disks. This strategy is useful for range queries because it helps to identify data partitions relevant to the query, skipping all of the uninvolved partitions. The disadvantage of this scheme is that data processing can be concentrated on a few disks leaving most computing resources underutilized, a phenomenon known as *access skew.* To minimize this effect, the relation can be divided into a large number of fragments using very small ranges. These fragments are distributed among the disks in a round-robin fashion.

Multidimensional Partitioning

Range partitioning cannot support range queries expressed on nonpartitioning attributes. To address this problem, multidimensional partitioning techniques allow a relation to be declustered based on multiple attributes. As an example, let us consider the case of partitioning a relation using two attributes, say age and salary. Each data fragment is characterized by a unique combination of age range and salary range. For instance, [2,4] denotes the data fragment whose tuples have the age and salary values falling within the second age range and fourth salary range, respectively. These data fragments can be allocated among the disks in different ways (8–11). As an example, the following function can be used to map a fragment $[X_1, X_2, \ldots, X_n]$ to a disk:

$$DISK\_ID(X_1, X_2, \cdots, X_n) = \left[\sum_{i=2}^{d} \left\lfloor \frac{X_i \cdot GCD_i}{N} \right\rfloor + \sum_{i=1}^{d} (X_i \cdot Shf\_dist_i)\right] \bmod N \tag{1}$$

where N is the number of disks, d is the number of partitioning attributes, $Shf\_dist_i = \lfloor \sqrt[d]{N} \rfloor^{i-1}$, and $GCD_i = gcd(Shf\_dist_i, N)$. A data placement example using this mapping function is illustrated in Fig. 3. Visually, the data fragments represented by the two-dimensional grid are assigned to the nine disks as follows.

1. Compute the shift distance, $shf\_dist = \lfloor \sqrt[d]{N} \rfloor = 3$.
2. Mark the top-most row as the check row.
3. Disks 0, 1, . . ., 8 are assigned to the nine fragments in this row from left to right. Make the next row the current row.
4. The allocation pattern for the current row is determined by circularly left-shifting the pattern of the row above it by three (i.e., *shf_dist*) positions.
5. If the allocation pattern of the current row is identical to that of the check row, we perform a circular left-shift on the current row one more position and mark it as the new check row.
6. If there are more rows to consider, make the next row the current row and repeat steps 4, 5 and 6.

Assuming that nine had been determined to be the optimal degree of I/O-parallelism for the given relation, this data placement scheme allows as many types of range queries to take full advantage of the I/O concurrency as possible. Range queries expressed on either age or salary or both can be supported effectively. The optimal degree of I/O parallelism is known as the degree of declustering (DoD), which defines the number of partitions a relation should have. For clarity, we assumed in this example that the number of intervals on each dimension is the same as the DoD. The mapping function [Eq. (1)], however, can be used without this restriction.

Many studies have observed that linear speedup for smaller numbers of processors could not always be extrapolated to larger numbers of processors. Although increasing the DoD improves the performance of a system, excessive declustering will reduce throughput due to overhead associated with parallel execution (12). Full declustering should not be used for very large parallel systems. The DoDs should be carefully determined to maximize the system throughput. A good approach is to evenly divide the disks into a number of groups, and assign relations which are frequently used together as operands of database operators (e.g., join) to the same disk group. Having different DoDs for various relations is not a good approach because the set of disks used by each relation would usually overlap with many sets of disks used for other relations. Under the circumstances, scheduling one operator for execution will cause most of the other concurrent queries to wait due to disk contention. This approach generally results in very poor system utilization.

PARALLEL EXECUTION

Today, essentially all parallel database servers support the relational data model and its standard query language: SQL (structured query language). SQL applications written for uniprocessor systems can be executed in these parallel servers without needing to modify the code. In a multi-user environ-

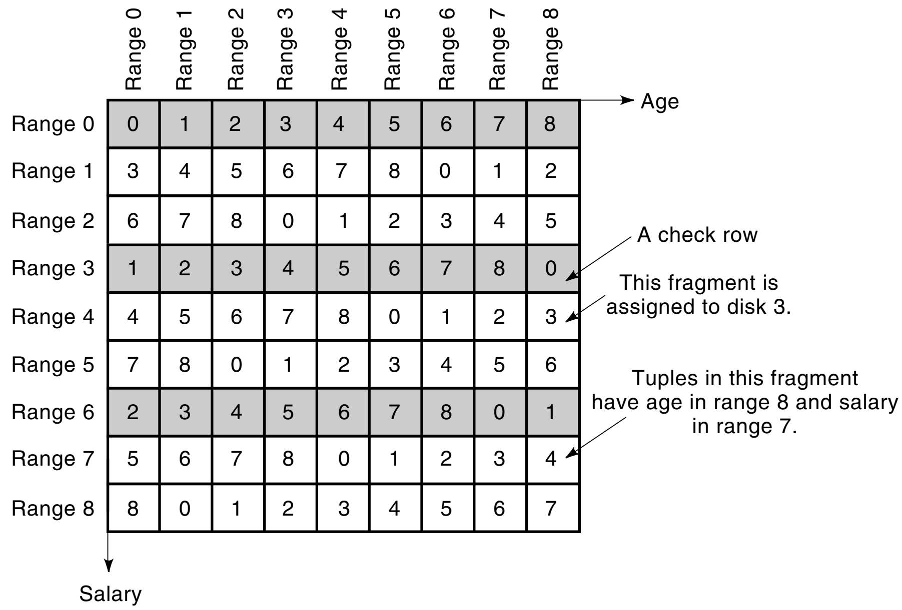

Figure 3. Two-dimensional data partitioning based on age and salary. The 9 × 9 data fragments are assigned to nine processing nodes. Range queries based on age, salary, or both can be supported effectively.

ment, queries submitted to the server are queued up and are processed in two steps:

- *Compile Time.* Each query is translated into a query tree which specifies the optimized order for executing the necessary database operators.
- *Execution Time.* The operators on these query trees are scheduled to execute in such a way to maximize system throughput while ensuring good response times.

Three types of parallelism can be exploited: interquery parallelism, intraquery parallelism, and intra-operator parallelism.

Intra-operator parallelism is achieved by executing a single database operator using several processors. This is possible if the operand relations are already partitioned and distributed across multiple disks. For instance, a scan process can be precreated in each processor at system startup time. To use a number of processors to scan a relation in parallel, we need only request the scan processes residing in these processors to carry out the local scans in parallel. In order to effectively support various types of queries, it is desirable to create at least one process in each processor for each type of primitive database operator. These processes are referred to as *operator servers.* They behave as a logical server specializing in a particular database operation. Once an operator server completes its work for a query, the logical server is returned to the free pool awaiting another service request to come from some pending query. By having queries share the operator servers, this approach avoids the overhead associated with process creation.

Interquery parallelism is realized by scheduling database operators from different queries for concurrent execution. Two scheduling approaches have been used:

Competition-Based Scheduling

In this scheme, a set of coordinator processes is precreated at system startup time. They are assigned to the queries by a dispatcher process according to some queuing discipline, say first come first serve (FCFS). When a coordinator is assigned to a query, it becomes responsible for scheduling the operators in the corresponding query tree. For each operator in the tree, the coordinator competes with other coordinators for the required operator servers. When the coordinator has successfully acquired all the operator servers needed for the task, it coordinates these servers to execute the operation in parallel. An obvious advantage of this approach is its simplicity. It assumes that the number of coordinators has been optimally set by the system administrator, and deals only with ways to reduce service times. The scheduling strategy is fair in the sense that each query is given the same opportunity to compete for the computing resources.

Planning-Based Scheduling

In this approach, all active queries share a single scheduler. Since this scheduler knows the resource requirements of all the active queries, it can schedule the operators of these queries based on how well their requirements match the current condition of the parallel system. For instance, a best-fit strategy can be used to select from among the pending operators the one that can make the maximum use of currently available operator servers to execute first. The motivation is to maximize the resource utilization. This approach, however, is not as fair as the competition-based technique. Queries which involve very small or very large relations can experience starvation. The scheduler can also become a bottleneck. To ameliorate the latter problem, a parallel search algorithm can be used to determine the best fit.

We note that the scheduling techniques discussed previously do not preclude the possibility of executing two or more operators of the same query simultaneously. This form of parallelism is referred to as intraquery parallelism. Both of these scheduling techniques try to maximize the system performance by strategically mixing all three forms of parallelism discussed herein.

LOAD BALANCING

Since each PN in an SN system processes the portion of the database on its local disks, the degree of parallelism is dictated by the placement of the data across the PNs. When the distribution is seriously skewed, balancing the load on these PNs is essential to good system performance (12,13). Although SE systems allow the collaborating processors to share the workload more easily, load balancing is still needed in such systems to maximize processor utilization (14). More specifically, the load balancing task should equalize the load on each disk, in addition to evenly dividing the data-processing tasks among the processors. As an example, let us consider an extreme scenario in which a large portion of the data which needs to be processed happens to reside on a single disk. Since little I/O parallelism can be exploited in this case, the storage subsystem cannot deliver a level of I/O performance commensurate with the computational capabilities of the SE system. Although the data processing tasks can still be perfectly balanced among the processors by sharing the workload stored on that one disk, the overall performance of the system is deteriorated due to poor utilization of the available I/O bandwidth. Similarly, balancing the data load among the disks is essential to the performance of SD systems. In summary, none of the architectures is immune to the skew effect. We shall see shortly that similar techniques can be used to address this problem in all three types of systems.

SE and SD systems, however, do have the advantage under the following circumstances. Let us consider a transaction-processing environment in which frequently accessed data are localized to only a few disks. Furthermore, the system memory is large enough to keep these frequently used data in the memory buffer most of the time. In this case, it is very easy for the processors of an SE or SD system to share the workload since each processor is allowed to access the shared disks. In contrast, when an SN system is faced with this situation, only a couple of the PNs which own the disks with the frequently used data are overly busy. The remaining PNs are idle most of the time. This phenomenon, however, is most likely due to bad data placement and usually can be rectified by redistributing the tuples.

Many load-balancing techniques have been developed for parallel database systems. Let us first examine techniques designed for the SN environment. Several parallel join algorithms have been proposed. Among them, hash-based algorithms are particularly suitable for SN systems. In these strategies, the operand relations are partitioned into buckets by applying the same randomizing hash function to the join key value, e.g., the join key value modulo the desired number of buckets. The buckets of the two relations, which correspond to the same hash value, are assigned to the same PN. These matching bucket pairs are evenly distributed among the PNs. Once the buckets have been assigned, each processor joins its local matching bucket pairs independently of the other PNs. This strategy is very effective unless there is a skew in the tuple distribution, i.e., the sizes of some buckets are substantially larger than the remaining buckets. When severe fluctuations occur among the bucket sizes, some processors are assigned significantly more tuples on which to perform the local join operation. Since the computation time of the join operation is determined by the slowest PN, skew in the tuple distribution seriously affects the overall performance of the system.

To minimize the skew effect, the buckets can be redistributed among the PNs as follows. At the end of the hashing stage, each PN keeps as many of the larger local buckets as possible; however, the total number of tuples retained should not exceed the ideal size each PN would have if the load were uniformly distributed. The excessive buckets are made available for redistribution among the PNs, using some bin-packing technique (e.g., largest processing time first), so as to balance the workload. This strategy is referred to as *partition tuning* (12). It handles severe skew conditions very well. However, when the skew condition is mild, the overhead associated with load balancing outweighs its benefits causing this technique to perform slightly worse than methods which do not perform load balancing at all. This is because this load balancing scheme scans the entire operand relations in order to determine the redistribution strategy. To reduce this overhead, the distribution of the tuples among the buckets can be estimated in the early stage of the bucket formation process as follows:

- *Sampling Phase.* Each PN independently takes a sample of both operand relations by retrieving the leading consecutive pages from it own disk. The size of the sample is chosen such that the entire sample can fit in the memory capacity. As the sampling tuples are brought into memory, they are declustered into a number of in-memory buckets by hashing on the join attributes.
- *Partition Tuning Phase.* A predetermined coordinating processor computes the sizes of the sampling buckets by adding up the sizes of the corresponding local buckets. It then determines how the sampling buckets should be assigned among the PNs, using some bin-packing technique, so as to evenly distribute the sampling tuples among the PNs.
- *Split Phase.* Each processor collects the assigned local sampling buckets to form the corresponding sampling join buckets on its own disk. When all the sampling tuples have been stored to disks, each PN continues to load the remaining tuples from the relations and redistribute them among the same buckets on disks. We note that tuples are not written to disk one at a time. Instead, each processor maintains a page buffer for each hash value. Tuples having the same hash values are piggybacked to the same page buffer, and the buffer is sent to its disk destination when it is full.
- *Join Phase.* Each PN performs the local joins of respectively matching buckets.

The sampling-based load balancing technique has the following advantages. First, the sampling and load balancing processes are blended with the normal join operation. As a result, the sampling phase incurs essentially no overhead. Second, since the sample is a byproduct of the normal join operation and therefore is free, the system can afford to use a large sample whose size is limited only by the memory capacity. Although the technique must rely on page-level sampling to keep the I/O cost low, studies show that a sample size as small as 5% of the size of the two operand relations is sufficient to accurately estimate the tuple distribution under practical conditions. With the capacity of today's memory technology, this scheme is effective for a wide range of database applications.

We note that although we focus our discussion on the join operation, the same technique can also be used for other relational operators. For instance, load balancing for the union operation can be implemented as follows. First, each PN hashes its portion of each operand relation (using an attribute with a large number of distinct values) into local buckets and stores them back on the local disks. A predetermined coordinating PN then assigns the respectively matching bucket-pairs to the PNs using the partition tuning technique. Once the distribution of the bucket pairs has been completed, each PN independently processes its local bucket pairs as follows. For each bucket pair, one bucket is first loaded to build an in-memory hash table. The tuples of the other bucket are then brought into memory to probe the hash table. When a match is found for a given tuple, it is discarded; otherwise, it is inserted into the hash table. At the end of this process, the hash tables located across the PNs contain the results of the union operation. Obviously, the sampling-based technique can also be adapted for this and other relational operators.

Partition tuning can also be used to balance workload in SE and SD systems. Let us consider an SE system, in which the operand relations are evenly distributed among n disks. A parallel join algorithm which uses n processors is given below.

- *Sampling Phase.* Each processor is associated with a distinct disk. Each processor independently takes a local sample of both operand relations by reading the leading consecutive pages from its disk unit. The size of the local samples is chosen such that the entire sample can fit in the available memory. As the sampling tuples are brought into memory, they are declustered into a number of in-memory local buckets by hashing on the join attributes. Each processor also counts the number of tuples in each of its local buckets.
- *Partition Tuning Phase.* A predetermined coordinating processor computes the sizes of the sampling buckets by adding up the sizes of the corresponding local buckets. It then determines how the sampling buckets should be assigned among the disks, using some bin-packing technique, so as to distribute the sampling tuples evenly among the disks.
- *Split Phase.* Each processor collects the assigned local sampling buckets to form the corresponding sampling join buckets on its disk. When all the sampling tuples have been collected to disks, each PN continues to load from its disk the remaining tuples of the two relations and redistribute them among the same buckets.
- *Join Phase.* Each PN joins the matching buckets located on its disk independently of the other PNs.

We observe in this algorithm that each disk performs the same number of read and write operations assuming the operand relations were evenly distributed across the disks. Furthermore, each processor processes the same number of tuples. The workload is perfectly balanced among the computing resources. An important advantage of associating a processor with a distinct disk unit is to avoid contention and to allow sequential access of the local partitions. Alternatively, the load can be evenly distributed by spreading each bucket across all the disks. This approach, however, requires each disk to serve all the processors at once during the join phase causing the read head to move in an anarchic way. On another issue, each processor using its own local buckets and page buffers during the sampling phase and split phase, respectively, also avoids contention. If the processors were allowed to write to a set of shared buckets as determined by the hash values, some mechanism would have been necessary to synchronize the write conflicts. This is not a good approach since the contention for some of the buckets would be very severe under a skew condition.

FUTURE DIRECTIONS AND RESEARCH PROBLEMS

Traditional parallel computers were designed to support computation-intensive scientific and engineering applications. As the processing power of inexpensive workstations has doubled every two years over the past decade, it has become feasible to run many of these applications on workstations. As a result, the market for parallel scientific and engineering applications has shrunk rapidly over the same period. A few major parallel computer manufacturers having financial difficulties in recent years is evidence of this phenomenon. Fortunately, a new and much stronger market has emerged for those manufacturers that could make the transition to adapt their machines to database applications. This time, business is much more profitable for following reasons. Firstly, the database market is a lot larger than that of scientific and engineering applications. In fact, significantly more than half of the computing resources in the world today are used for data-processing related tasks. Secondly, advances in microprocessor technology do not make workstations more suitable for handling database management tasks which are known to be I/O intensive. It would be impractical to pack a workstation with a very large number of disks. This is not even desirable because most data should be centralized in a repository to allow data sharing. Thirdly, managing a large amount of multimedia data has become a necessity for many business sectors. Only parallel database servers can have the scalable bandwidth to support such applications.

As parallel database systems displaced scientific and engineering applications as the primary applications for parallel computers, manufacturers put a great deal of attention in improving the I/O capabilities of their machines. With the emergence of multimedia applications, however, a new hurdle, the network-I/O bottleneck (15,16), is facing the database community. Essentially all of today's parallel database servers are designed for conventional database applications. They are not suitable for applications that involve multimedia data. For conventional database applications, the server requires a lot of storage-I/O bandwidth to support query processing. On the other hand, the demand on the network-I/O bandwidth is minimal since the results returned to the clients are typically a very small fraction of the data examined by the query. In contrast, the database server must deliver very large multimedia objects as query results to the clients in a multimedia application. As an example, the network-I/O bottleneck is encountered in Time Warner Cable's *Full Service Network* project in Orlando. Although each of the SGI Challenge servers used in this project can sustain thousands of storage-I/O streams, the network-I/O bottleneck limits its performance to less than 120 MPEG-1 video streams. This is reminiscent of a large crowd funneling out of the gates after a football match. To address this bottleneck, eight servers had to be used at Time Warner Cable to serve the 4,000 homes significantly increasing the hardware cost and the costs of hiring additional system administrators. It is essential that futuregeneration servers have sufficient network-I/O bandwidth to make their storage bandwidth available to clients for retrieving large multimedia data.

Today's parallel database systems use only sequential algorithms to perform query optimization despite the large number of processors available in the system. Under time constraints, no optimizer can consider all the parallel algorithms for each operator and all the possible query tree organizations. A parallel parallelizing query optimizer is highly desirable. It would have the leeway to examine many more possibilities. A potential solution is to divide the possible plans among a number of optimizer instances running on different processors. The costs of various plans can be estimated in parallel. At the end, a coordinating optimizer compares the best candidates nominated by the participating optimizers and selects the best plan. With the additional resources, it also becomes feasible to optimize multiple queries together to allow sharing of intermediate results. Considering the fact that most applications access 20% of their data 80% of the time, this approach could be a major improvement. More work is needed in this area.

Parallel database systems offer parallelism within the database system. On the other hand, existing parallel programming languages are not designed to take advantage of parallel database systems. There is a mismatch between the two technologies. To address this issue, two strategies can be considered. One approach is to introduce new constructs in the parallel programming language to allow computer programs to be structured in a way to exploit database parallelism. Alternatively, one can consider implementing a persistent parallel programming language by extending SQL with general-purpose parallel programming functionality. Several companies have extended SQL with procedural programming constructs such as sequencing, conditionals, and loops. However, no parallel processing constructs have been proposed. Such a language is critical to applications that are both I/O intensive and computationally intensive.

As the object-oriented paradigm becomes a new standard for software development, SQL is being extended with object functionality. The ability to process rules is also being incorporated to support a wider range of applications. How to enhance existing parallel database server technology to support the extended data model is a great challenge facing the database community. For instance, SQL3 supports sequence and graph structures. We need new data placement techniques and parallel algorithms for these nonrelational data objects. Perhaps, techniques developed in the parallel programming language community can be adapted for this purpose.

BIBLIOGRAPHY

1. H. Borak et al., Prototyping bubba, a highly parallel database system, *IEEE Trans. Knowl. Data Eng.,* **2**: 4–24, 1990.
2. D. DeWitt et al., The gamma database machine project, *IEEE Trans. Knowl. Data Eng.,* **2**: 44–62, 1990.
3. K. A. Hua and H. Young, Designing a highly parallel database server using off-the-shelf components, *Proc. Int. Comp. Symp.,* pp. 17–19, 1990.
4. M. Kitsuregawa, H. Tanaka, and T. Moto-oka, Application of hash to data base machine and its architecture, *New Gen. Comp.,* **1** (1): 63–74, 1983.
5. M. Stonebraker, The case for shared nothing, *Database Eng.,* **9** (1): 1986.
6. K. A. Hua, C. Lee, and J. Peir, Interconnecting shared-nothing systems for efficient parallel query processing, *Proc. Int. Conf. Parallel Distrib. Info. Sys.,* 1991, pp. 262–270.
7. D. DeWitt and J. Gray, Parallel database systems: The future of high performance database systems, *Commun. ACM,* **35** (6): 85–98, 1992.
8. L. Chen and D. Rotem, Declustering objects for visualization, *Proc. Int. Conf. Very Large Data Bases,* 1993, pp. 85–96.
9. H. C. Du and J. S. Sobolewski, Disk allocation for Cartesian product files on multiple disk systems, *ACM Trans. Database Sys.,* **7** (1): 82–101, 1982.
10. C. Fabursos and P. Bhagwat, Declustering using fracals, *Proc. Int. Conf. Parallel Distrib. Inf. Sys.,* 1993, pp. 18–25.
11. K. A. Hua and C. Lee, An adaptive data placement scheme for parallel database computer systems, *Proc. Int. Conf. Very Large Data Bases,* 1990, pp. 493–506.
12. K. A. Hua and C. Lee, Handling data skew in multicomputer database systems using partitioning tuning, *Proc. Int. Conf. Very Large Data Bases,* 1991, pp. 525–535.
13. J. Wolf, D. Dias, and P. Yu, An effective algorithm for parallelizing hash joins in the presence of data skew, *Proc. Int. Conf. Data Eng.,* 1991, pp. 200–209.
14. E. Omiecinski, Performance analysis of a load balancing hash-join algorithm for shared memory multiprocessor, *Proc. Int. Conf. Very Large Data Bases,* 1991, pp. 375–385.
15. K. Hua and S. Sheu, Skyscraper broadcasting: A new broadcasting scheme for metropolitan video-on-demand systems, *Proc. ACM SIGCOMM'97 Conf.,* 1997.
16. S. Sheu, K. Hua, and W. Tavanapong, Chaining: A generalized batching technique for video-on-demand systems, *Proc. IEEE Int. Conf. Multimedia Com. Sys.,* 1997.

KIEN A. HUA
University of Central Florida

PARALLEL DATABASES. See DISTRIBUTED DATABASES.