Parallel Database Systems: The Future of High Performance Database Systems

David DeWitt and Jim Gray

Highly parallel database systems are beginning to displace traditional mainframe computers for the largest database and transaction processing tasks. The success of these systems refutes a 1983 paper predicting the demise of database machines [3]. Ten years ago the future of highly parallel database machines seemed gloomy, even to their staunchest advocates. Most database machine research had focused on specialized, often trendy, hardware such as CCD memories, bubble memories, head-per-track disks, and optical disks. None of these technologies fulfilled their promises; so there was a sense that conventional CPUs, electronic RAM, and moving-head magnetic disks would dominate the scene for many years to come. At that time, disk throughput was predicted to double while processor speeds were predicted to increase by much larger factors. Consequently, critics predicted that multiprocessor systems would soon be I/O limited unless a solution to the I/O bottleneck was found.

While these predictions were fairly accurate about the future of hardware, the critics were certainly wrong about the overall future of parallel database systems. Over the last decade Teradata, Tandem, and a host of startup companies have successfully developed and marketed highly parallel machines.

Database Systems

Parallel

Why have parallel database systems become more than a research curiosity? One explanation is the widespread adoption of the relational data model. In 1983 relational database systems were just appearing in the marketplace; today they dominate it. Relational queries are ideally suited to parallel execution; they consist of uniform operations applied to uniform streams of data. Each operator produces a new relation, so the operators can be composed into highly parallel dataflow graphs. By streaming the output of one operator into the input of another operator, the two operators can work in series giving *pipelined parallelism*. By partitioning the input data among multiple processors and memories, an operator can often be split into many independent operators each working on a part of the data. This partitioned data and execution gives *partitioned parallelism* (Figure 1).

The dataflow approach to database system design needs a message-based client-server operating system to interconnect the parallel processes executing the relational operators. This in turn requires a high-speed network to interconnect the parallel processors. Such facilities seemed exotic a decade ago, but now they are the mainstream of computer architecture. The client-server paradigm using high-speed LANs is the basis for most PC, workstation, and workgroup software. Those same client-server mechanisms are an excellent basis for distributed database technology.

Mainframe designers have found it difficult to build machines powerful enough to meet the CPU and I/O demands of relational databases serving large numbers of simultaneous users or searching terabyte databases. Meanwhile, multiprocessors based on fast and inexpensive microprocessors have become widely available from vendors including Encore, Intel, NCR, nCUBE, Sequent, Tandem, Teradata, and Thinking Machines. These machines provide more total power than their mainframe coun-

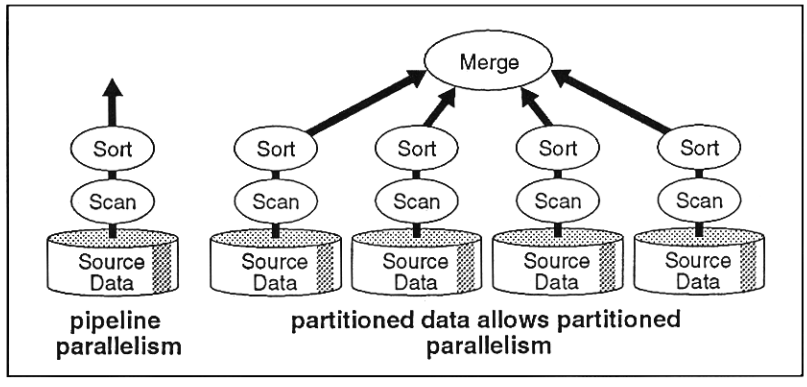

Figure 1.
The dataflow approach to relational operators gives both pipelined and partitioned parallelism. Relational data operators take relations (uniform sets of records) as input and produce relations as outputs. This allows them to be composed in dataflow graphs that allow *pipeline parallelism* (left) in which the computation of one operator proceeds in parallel with another, and *partitioned parallelism* in which operators (sort and scan in the diagram at the right) are replicated for each data source, and the replicas execute in parallel.

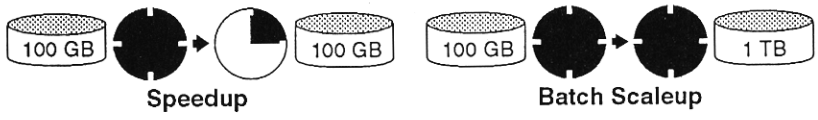

Figure 2.
Speedup and Scaleup. A speedup design performs a one-hour job four times faster when run on a four-times larger system. A scaleup design runs a ten-times bigger job is done in the same time by a ten-times bigger system.

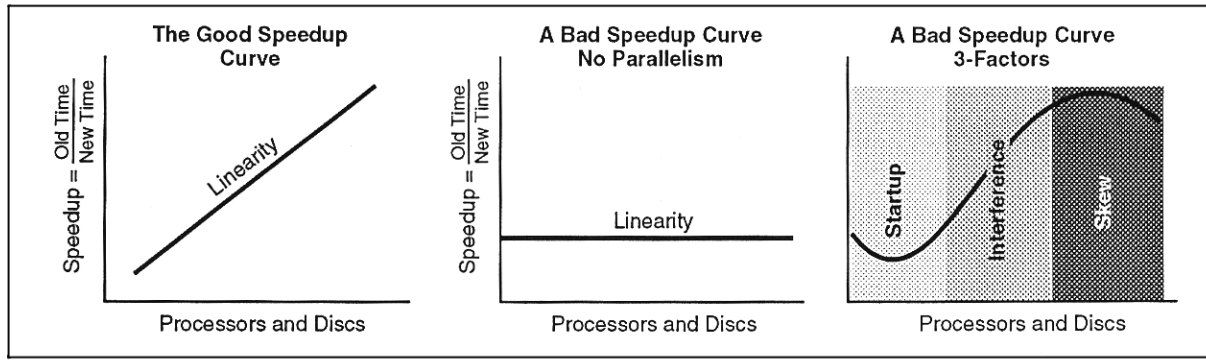

Figure 3.
Good and bad speedup curves. The standard speedup curves. The left curve is the ideal. The middle graph shows no speedup as hardware is added. The right curve shows the three threats to parallelism. Initial startup costs may dominate at first. As the number of processes increase, interference can increase. Ultimately, the job is divided so finely, that the variance in service times (skew) causes a slowdown.

terparts at a lower price. Their modular architectures enable systems to grow incrementally, adding MIPS, memory, and disks either to speedup the processing of a given job, or to scaleup the system to process a larger job in the same time.

In retrospect, special-purpose database machines have indeed failed; but parallel database systems are a big success. The successful parallel database systems are built from conventional processors, memories, and disks. They have emerged as major consumers of highly parallel architectures, and are in an excellent position to exploit massive numbers of fast-cheap commodity disks, processors, and memories promised by current technology forecasts.

A consensus on parallel and distributed database system architecture has emerged. This architecture is based on a *shared-nothing* hardware design [29] in which processors communicate with one another only by sending messages via an interconnection network. In such systems, tuples of each relation in the database are *partitioned (declustered)* across disk storage units[1] attached directly to each processor. Partitioning allows multiple processors to scan large relations in parallel without needing any exotic I/O devices. Such architectures were pioneered by Teradata in the late 1970s and by several research projects. This design is now used by Teradata, Tandem, NCR, OracleNCUBE, and several other products currently under development. The research community has also embraced this shared-nothing dataflow architecture in systems like Arbre, Bubba, and Gamma.

The remainder of this article is organized as follows: The next section describes the basic architectural concepts used in these parallel database systems. This is followed by a brief presentation of the unique features of the Teradata, Tandem, Bubba, and Gamma systems in the following section, entitled "The State of the Art." Several areas for future research are described in "Future Directions and Research Problems" prior to the conclusion of this article.

Basic Techniques for Parallel Database Machine Implementation

Parallelism Goals and Metrics: Speedup and Scaleup

The ideal parallel system demonstrates two key properties: (1) *linear speedup:* Twice as much hardware can perform the task in half the elapsed time, and (2) *linear scaleup:* Twice as much hardware can perform twice as large a task in the same elapsed time (see Figures 2 and 3).

More formally, given a fixed job run on a small system, and then run on a larger system, the *speedup* given by the larger system is measured as:

$$Speedup = \frac{small\_system\_elapsed\_time}{big\_system\_elapsed\_time}$$

Speedup is said to be linear, if an N-times large or more expensive system yields a speedup of N.

Speedup holds the problem size constant, and grows the system. Scaleup measures the ability to grow both the system and the problem. *Scaleup* is defined as the ability of an N-times larger system to perform an N-times larger job in the same elapsed time as the original system. The scaleup metric is:

$$Scaleup = \frac{small\_system\_elapsed\_time\_on\_small\_problem}{big\_system\_elapsed\_time\_on\_big\_problem}$$

If this scaleup equation evaluates to 1, then the scaleup is said to be linear[2]. There are two distinct kinds of scaleup, batch and transactional. If the job consists of performing many small independent requests submitted by many clients and operating on a shared database, then scaleup consists of N-times as many clients, submitting N-times as many requests against an N-times larger database. This is the scaleup typically found in transaction processing systems and timesharing systems. This form of scaleup is used by the Transaction Processing Performance Council to scaleup their transaction processing benchmarks [36]. Consequently, it is called *transaction scaleup*. Transaction scaleup is ideally suited to parallel systems since each transaction is typically a small independent job that can be run on a separate processor.

A second form of scaleup, called *batch scaleup*, arises when the scaleup task is presented as a single large job. This is typical of database queries and is also typical of scientific simulations. In these cases, scaleup consists of using an N-times larger computer to solve an N-times larger problem. For database systems batch scaleup translates to the same query on an N-times larger database; for scientific problems, batch scaleup translates to the same calculation on an N-times finer grid or on an N-times longer simulation.

The generic barriers to linear speedup and linear scaleup are the triple threats of:

startup: The time needed to start a parallel operation. If thousands of processes must be started, this can easily dominate the actual computation time.

interference: The slowdown each new process imposes on all others when accessing shared resources.

skew: As the number of parallel steps increases, the average size of each step decreases, but the variance can well exceed the mean. The service time of a job is the service time of the slowest step of the job. When the variance dominates the mean, increased parallelism improves elapsed time only slightly.

The subsection "A Parallel

[1]The term disk is used here as a shorthand for disk or other nonvolatile storage media. As the decade proceeds, nonvolatile electronic storage or some other media may replace or augment disks.

[2]The execution cost of some operators increases super-linearly. For example, the cost of sorting n-tuples increases as $nlog(n)$. When n is in the billions, scaling up by a factor of a thousand, causes $nlog(n)$ to increase by 3,000. This 30% deviation from linearity in a three-orders-of-magnitude scaleup justifies the use of the term *near-linear* scaleup.

Dataflow Approach to SQL Software" describes several basic techniques widely used in the design of shared-nothing parallel database machines to overcome these barriers. These techniques often achieve linear speedup and scaleup on relational operators.

Hardware Architecture, the Trend to Shared-Nothing Machines

The ideal database machine would have a single infinitely fast processor with an infinite memory with infinite bandwidth—and it would be infinitely cheap (free). Given such a machine, there would be no need for speedup, scaleup, or parallelism. Unfortunately, technology is not delivering such machines—but it is coming close. Technology is promising to deliver fast one-chip processors, fast high-capacity disks, and high-capacity electronic RAM. It also promises that each of these devices will be very inexpensive by today's standards, costing only hundreds of dollars each.

So, the challenge is to build an infinitely fast processor out of infinitely many processors of finite speed, and to build an infinitely large memory with infinite memory bandwidth from infinitely many storage units of finite speed. This sounds trivial mathematically; but in practice, when a new processor is added to most computer designs, it slows every other computer down just a little bit. If this slowdown (interference) is 1%, then the maximum speedup is 37 and a 1,000-processor system has 4% of the effective power of a single-processor system.

How can we build scaleable multiprocessor systems? Stonebraker suggested the following simple taxonomy for the spectrum of designs (see Figures 4 and 5) [29][3]:

shared-memory: All processors share direct access to a common global memory and to all disks. The IBM/370, Digital VAX, and Sequent Symmetry multiprocessors typify this design.

shared-disks: Each processor has a private memory but has direct access to all disks. The IBM Sysplex and original Digital VAXcluster typify this design.

shared-nothing: Each memory and disk is owned by some processor that acts as a server for that data. Mass storage in such an architecture is distributed among the processors by connecting one or more disks. The Teradata, Tandem, and nCUBE machines typify this design.

Shared-nothing architectures minimize interference by minimizing resource sharing. They also exploit commodity processors and memory without needing an incredibly powerful interconnection network. As Figure 5 suggests, the other architectures move large quantities of data through the interconnection network. The shared-nothing design moves only questions and answers through the network. Raw memory accesses and raw disk accesses are performed locally in a processor, and only the filtered (reduced) data is passed to the client program. This allows a more scaleable design by minimizing traffic on the interconnection network.

Shared-nothing characterizes the database systems being used by Teradata [33], Gamma [8, 9], Tandem [32], Bubba [1], Arbre [21], and nCUBE [13]. Significantly, Digital's VAXcluster has evolved to this design. DOS and UNIX workgroup systems from 3com, Borland, Digital, HP, Novell, Microsoft, and Sun also adopt a shared-nothing client-server architecture.

The actual interconnection networks used by these systems vary enormously. Teradata employs a redundant tree-structured communication network. Tandem uses a three-level duplexed network, two levels within a cluster, and rings connecting the clusters. Arbre, Bubba, and Gamma are independent of the underlying interconnection network, requiring only that the network allow any two nodes to communicate with one another. Gamma operates on an Intel Hypercube. The Arbre prototype was implemented using IBM 4381 processors connected to one another in a point-to-point network. Workgroup systems are currently making a transition from Ethernet to higher speed local networks.

The main advantage of shared-nothing multiprocessors is that they can be scaled up to hundreds and probably thousands of processors that do not interfere with one another. Teradata, Tandem, and Intel have each shipped systems with more than 200 processors. Intel is implementing a 2,000-node hypercube. The largest shared-memory multiprocessors currently available are limited to about 32 processors.

These shared-nothing architectures achieve near-linear speedups and scaleups on complex relational queries and on on-line transaction processing workloads [9, 10, 32]. Given such results, database machine designers see little justification for the hardware and software complexity associated with shared-memory and shared-disk designs.

Shared-memory and shared-disk systems do not scale well on database applications. Interference is a major problem for shared-memory multiprocessors. The interconnection network must have the bandwidth of the sum of the processors and disks. It is difficult to build such networks that can scale to thousands of nodes. To reduce network traffic and to minimize latency, each processor is given a large private cache. Measurements of shared-memory multiprocessors running database workloads show that loading and flushing these caches considerably degrades processor performance [35]. As parallelism increases, interference on shared resources limits performance. Multiprocessor systems

[3]Single Instruction stream, Multiple Data stream (SIMD) machines such as ILLIAC IV and its derivatives like MASSPAR and the "old" Connection Machine are ignored here because to date they have few successes in the database area. SIMD machines seem to have application in simulation, pattern matching, and mathematical search, but they do not seem to be appropriate for the multiuser, I/O intensive, and dataflow paradigm of database systems.

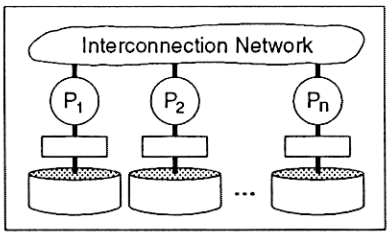

Figure 4.
The basic shared-nothing design. Each processor has a private memory and one or more disks. Processors communicate via a high-speed interconnect network. Teradata, Tandem, nCUBE, and the newer VAXclusters typify this design.

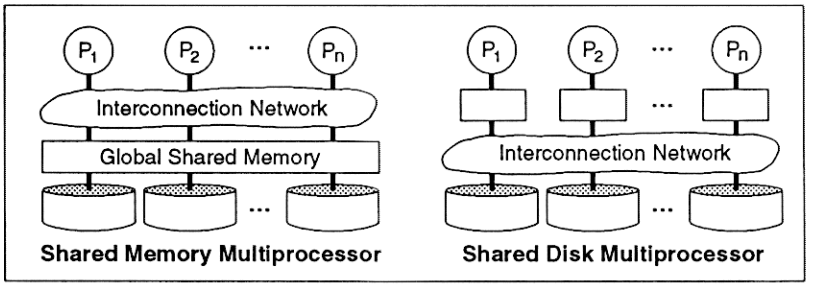

Figure 5.
The shared-memory and shared-disk designs. A shared-memory multiprocessor connects all processors to a globally shared memory. Multiprocessor IBM/370, VAX, and Sequent computers are typical examples of shared-memory designs. Shared-disk systems give each processor a private memory, but all the processors can directly address all the disks. Digital's VAXcluster and IBM's Sysplex typify this design.

Figure 6.
Example of a scan of a telephone relation to find the phone numbers of all people named Smith.

```
SELECT  telephone_number      /* the output attribute(s) */
FROM    telephone_book        /* the input relation      */
WHERE   last_name = 'Smith';  /* the predicate           */
```

often use an affinity scheduling mechanism to reduce this interference; giving each process an affinity to a particular processor. This is a form of data partitioning; it represents an evolutionary step toward the shared-nothing design. Partitioning a shared-memory system creates many of the skew and load balancing problems faced by a shared-nothing machine; but reaps none of the simpler hardware interconnect benefits. Based on this experience, we believe high-performance shared-memory machines will not economically scale beyond a few processors when running database applications.

To ameliorate the interference problem, most shared-memory multiprocessors have adopted a shared-disk architecture. This is the logical consequence of affinity scheduling. If the disk interconnection network can scale to thousands of disks and processors, then a shared-disk design is adequate for large read-only databases and for databases where there is no concurrent sharing. The shared-disk architecture is not very effective for database applications that read and write a shared database. A processor wanting to update some data must first obtain the current copy of that data. Since others might be updating the same data concurrently, the processor must declare its intention to update the data. Once this declaration has been honored and acknowledged by all the other processors, the updator can read the shared data from disk and update it. The processor must then write the shared data to disk so that subsequent readers and writers will be aware of the update. There are many optimizations of this protocol, but they all end up exchanging reservation messages and exchanging large physical data pages. This creates processor interference and delays. It creates heavy traffic on the shared interconnection network.

For shared database applications, the shared-disk approach is much more expensive than the shared-nothing approach of exchanging small high-level logical questions and answers among clients and servers. One solution to this interference has been to give data a processor affinity; other processors wanting to access the data send messages to the server managing the data. This has emerged as a major application of transaction processing monitors that partition the load among partitioned servers, and is also a major application for remote procedure calls. Again, this trend toward the partitioned data model and shared-nothing architecture on a shared-disk system reduces interference. Since the shared-disk system interconnection network is difficult to scale to thousands of processors and disks, many conclude that it would be better to adopt the shared-nothing architecture from the start.

Given the shortcomings of shared-disk and shared-memory architectures, why have computer architects been slow to adopt the shared-nothing approach? The first answer is simple, high-performance, low-cost commodity components have only recently become available. Traditionally, commodity components provided relatively low performance and low quality.

Today, old software is the most significant barrier to the use of parallelism. Old software written for uniprocessors gets no speedup or scaleup when put on any kind of multiprocessor. It must be rewritten to benefit from parallel processing and multiple disks. Database applications are a unique exception to this. Today, most database programs are written in the relational language SQL that has been standardized by both ANSI and ISO. It is possible to take standard SQL applications written for uniprocessor systems and execute them in parallel on shared-nothing database

machines. Database systems can automatically distribute data among multiple processors. Teradata and Tandem routinely port SQL applications to their system and demonstrate near-linear speedups and scaleups. The following subsection explains the basic techniques used by such parallel database systems.

A Parallel Dataflow Approach to SQL Software

Terabyte on-line databases, consisting of billions of records, are becoming common as the price of on-line storage decreases. These databases are often represented and manipulated using the SQL relational model. The next few paragraphs give a rudimentary introduction to relational model concepts needed to understand the remainder of this article.

A relational database consists of *relations* (*files* in COBOL terminology) that in turn contain *tuples* (*records* in COBOL terminology). All the tuples in a relation have the same set of *attributes* (*fields* in COBOL terminology).

Relations are created, updated, and queried by writing SQL statements. These statements are syntactic sugar for a simple set of operators chosen from the relational algebra. *Select-project,* here called *scan,* is the simplest and most common operator—it produces a row-and-column subset of a relational table. A scan of relation R using predicate P and attribute list L produces a relational data stream as output. The scan reads each tuple, t, of R and applies the predicate P to it. If $P(t)$ is true, the scan discards any attributes of t not in L and inserts the resulting tuple in the scan output stream. Expressed in SQL, a scan of a telephone book relation to find the phone numbers of all people named Smith would be written as shown in Figure 6. A scan's output stream can be sent to another relational operator, returned to an application, displayed on a terminal, or printed in a report. Therein lies the beauty and utility of the relational model. The uniformity of the data and operators allow them to be arbitrarily composed into dataflow graphs.

The output of a scan may be sent to a *sort* operator that will reorder the tuples based on an attribute sort criteria, optionally eliminating duplicates. SQL defines several *aggregate* operators to summarize attributes into a single value, for example, taking the sum, min, or max of an attribute, or counting the number of distinct values of the attribute. The *insert* operator adds tuples from a stream to an existing relation. The *update* and *delete* operators alter and delete tuples in a relation matching a scan stream.

The relational model defines several operators to combine and compare two or more relations. It provides the usual set operators *union, intersection, difference,* and some more exotic ones like *join* and *division.* Discussion here will focus on the *equi-join* operator (here called *join*). The join operator composes two relations, A and B, on some attribute to produce a third relation. For each tuple, *ta*, in A, the join finds all tuples, *tb*, in B whose attribute values are equal to that of *ta*. For each matching pair of tuples, the join operator inserts into the output stream a tuple built by concatenating the pair.

Codd, in a classic paper, showed that the relational data model can represent any form of data, and that these operators are complete [5]. Today, SQL applications are typically a combination of conventional programs and SQL statements. The programs interact with clients, perform data display, and provide high-level direction of the SQL dataflow.

The SQL data model was originally proposed to improve programmer productivity by offering a nonprocedural database language. Data independence was an additional benefit; since the programs do not specify how the query is to be executed, SQL programs continue to operate as the logical and physical database schema evolves.

Parallelism is an unanticipated benefit of the relational model. Since relational queries are really just relational operators applied to very large collections of data, they offer many opportunities for parallelism. Since the queries are presented in a nonprocedural language, they offer considerable latitude in executing the queries.

Relational queries can be executed as a dataflow graph. As mentioned in the first section of this article, these graphs can use both pipelined parallelism and partitioned parallelism. If one operator sends its output to another, the two operators can execute in parallel giving potential speedup of two.

The benefits of pipeline parallelism are limited because of three factors: (1) Relational pipelines are rarely very long—a chain of length ten is unusual. (2) Some relational operators do not emit their first output until they have consumed all their inputs. Aggregate and sort operators have this property. One cannot pipeline these operators. (3) Often, the execution cost of one operator is much greater than the others (this is an example of skew). In such cases, the speedup obtained by pipelining will be very limited.

Partitioned execution offers much better opportunities for speedup and scaleup. By taking the large relational operators and partitioning their inputs and outputs, it is possible to use divide-and-conquer to turn one big job into many independent little ones. This is an ideal situation for speedup and scaleup. Partitioned data is the key to partitioned execution.

Data Partitioning. Partitioning a relation involves distributing its tuples over several disks. Data partitioning has its origins in centralized systems that had to partition files, either because the file was too big for one disk, or because the file access rate could not be supported by a single disk. Distributed databases use data partitioning when they place relation fragments at different network sites [23]. Data par-

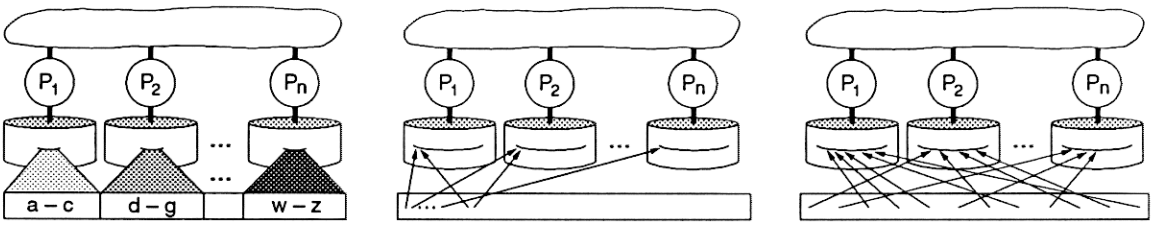

Figure 7.
The three basic partitioning schemes. Range partitioning maps contiguous attribute ranges of a relation to various disks. Round-robin partitioning maps the i'th tuple to disk i **mod** n. Hashed partitioning maps each tuple to a disk location based on a hash function. Each of these schemes spreads data among a collection of disks, allowing parallel disk access and parallel processing.

titioning allows parallel database systems to exploit the I/O bandwidth of multiple disks by reading and writing them in parallel. This approach provides I/O bandwidth superior to RAID-style systems without needing any specialized hardware [22, 24].

The simplest partitioning strategy distributes tuples among the fragments in a *round-robin* fashion. This is the partitioned version of the classic entry-sequence file. Round-robin partitioning is excellent if all applications want to access the relation by sequentially scanning all of it on each query. The problem with round-robin partitioning is that applications frequently want to associatively access tuples, meaning that the application wants to find all the tuples having a particular attribute value. The SQL query looking for the Smiths in the phone book shown in Figure 6 is an example of an associative search.

Hash partitioning is ideally suited for applications that want only sequential and associative access to the data. Tuples are placed by applying a *hashing* function to an attribute of each tuple. The function specifies the placement of the tuple on a particular disk. Associative access to the tuples with a specific attribute value can be directed to a single disk, avoiding the overhead of starting queries on multiple disks. Hash partitioning mechanisms are provided by Arbre, Bubba, Gamma, and Teradata.

Database systems pay considerable attention to clustering related data together in physical storage. If a set of tuples is routinely accessed together, the database system attempts to store them on the same physical page. For example, if the Smiths of the phone book are routinely accessed in alphabetical order, then they should be stored on pages in that order, these pages should be clustered together on disk to allow sequential prefetching and other optimizations. Clustering is very application-specific. For example, tuples describing nearby streets should be clustered together in geographic databases, tuples describing the line items of an invoice should be clustered with the invoice tuple in an inventory control application.

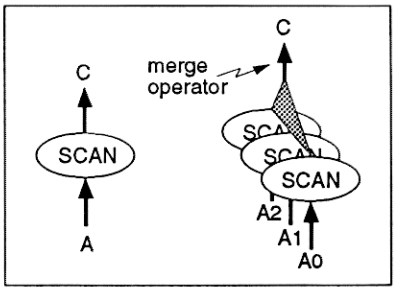

Figure 8.
Partitioned data parallelism. A simple relational dataflow graph showing a relational scan (project and select) decomposed into three scans on three partitions of the input stream or relation. These three scans send their output to a merge node that produces a single data stream.

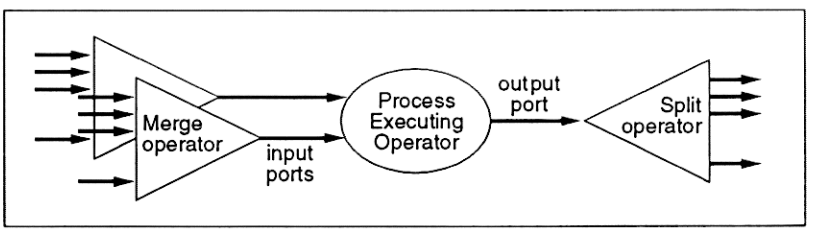

Figure 9.
Merging the inputs and partitioning the output of an operator. A relational dataflow graph showing a relational operator's inputs being merged to a sequential stream per port. The operator's output is being decomposed by a split operator in several independent streams. Each stream may be a duplicate or a partitioning of the operator output stream into many disjoint streams. With the split and merge operators, a web of simple sequential dataflow nodes can be connected to form a parallel execution plan.

Hashing tends to randomize data rather than cluster it. *Range partitioning* clusters tuples with similar attributes together in the same partition. It is good for sequential and associative access, and is also good for clustering data. Figure 7 shows range partitioning based on lexicographic order, but any clustering algorithm is possible. Range partitioning derives its name from the typical SQL range queries such as

```
latitude BETWEEN 38° AND 39°
```

Arbre, Bubba, Gamma, Oracle, and Tandem provide range partitioning.

The problem with range partitioning is that it risks *data skew*, where all the data is placed in one partition, and *execution skew* in which all the execution occurs in one partition. Hashing and round-robin are less susceptible to these skew problems. Range partitioning can minimize skew by picking non-uniformly-distributed partitioning criteria. Bubba uses this concept by considering the access frequency *(heat)* of each tuple when creating partitions of a relation; the goal being to balance the frequency with which each partition is accessed (its *temperature*) rather than the actual number of tuples on each disk (its volume) [6].

Table 1.
Sample Split Operators.
Each split operator maps tuples to a set of output streams (ports of other processes) depending on the range value (predicate) of the input tuple. The split operator on the left is for the relation A scan in Figure 10, while the table on the right is for the relation B scan. The tables partition the tuples among three data streams.

Relation A Scan Split Operator		Relation B Scan Split Operator	
Predicate	**Destination Process**	**Predicate**	**Destination Process**
"A-H"	(CPU #5, Process #3, Port #0)	"A-H"	(CPU #5, Process #3, Port #1)
"I-Q"	(CPU #7, Process #8, Port #0)	"I-Q"	(CPU #7, Process #8, Port #1)
"R-Z"	(CPU #2, Process #2, Port #0)	"R-Z"	(CPU #2, Process #2, Port #1)

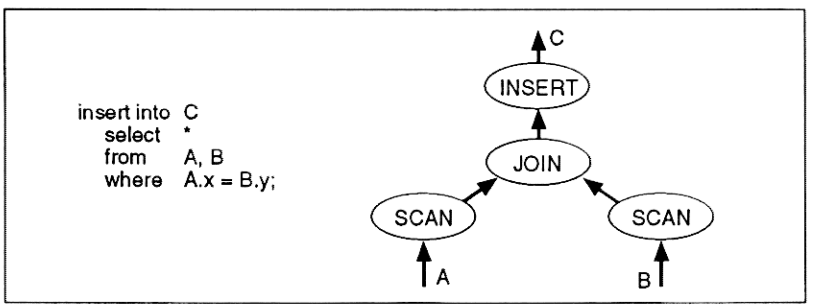

Figure 10.
A simple SQL query and the associated relational query graph. The query specifies that a join is to be performed between relations A and B by comparing the x attribute of each tuple from the A relation with the y attribute value of each tuple of the B relation. For each pair of tuples that satisfy the predicate, a result tuple is formed from all the attributes of both tuples. This result tuple is then added to the result relation C. The associated logical query graph (as might be produced by a query optimizer) shows a tree of operators, one for the join, one for the insert, and one for scanning each input relation.

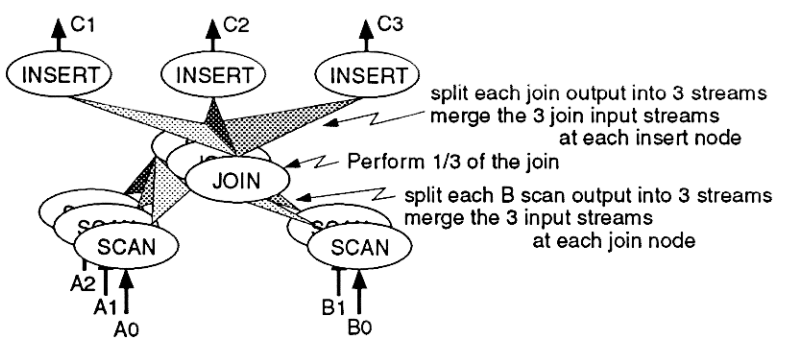

Figure 11.
A simple relational dataflow graph. It shows two relational scans (project and select) consuming two input relations, A and B and feeding their outputs to a join operator that in turn produces a data stream C.

While partitioning is a simple concept that is easy to implement, it raises several new physical database design issues. Each relation must now have a partitioning strategy and a set of disk fragments. Increasing the degree of partitioning usually reduces the response time for an individual query and increases the overall throughput of the system. For sequential scans, the response time decreases because more processors and disks are used to execute the query. For associative scans, the response time improves because fewer tuples are stored at each node and hence the size of the index that must be searched decreases.

There is a point beyond which further partitioning actually increases the response time of a query. This point occurs when the cost of starting a query on a node becomes a significant fraction of the actual execution time [6, 11].

Parallelism Within Relational Operators. Data partitioning is the first step in partitioned execution of relational dataflow graphs. The basic idea is to use parallel data streams instead of writing new parallel operators (programs). This approach enables the use of unmodified, existing sequential routines to execute the relational operators in parallel. Each relational operator has a set of *input ports* on which input tuples arrive and an *output* port to which the operator's output stream is sent. The parallel dataflow works by partitioning and merging data streams into these sequential ports. This approach allows the use of existing sequential relational operators to execute in parallel.

Consider a scan of a relation, A, that has been partitioned across three disks into fragments A0, A1, and A2. This scan can be imple-

mented as three scan operators that send their output to a common merge operator. The merge operator produces a single output data stream to the application or to the next relational operator. The parallel query executor creates the three scan processes shown in Figure 8 and directs them to take their inputs from three different sequential input streams (A0, A1, A2). It also directs them to send their outputs to a common merge node. Each scan can run on an independent processor and disk. So the first basic parallelizing operator is a *merge* that can combine several parallel data streams into a single sequential stream.

The merge operator tends to focus data on one spot. If a multistage parallel operation is to be done in parallel, a single data stream must be split into several independent streams. A *split operator* is used to partition or replicate the stream of tuples produced by a relational operator. A split operator defines a mapping from one or more attribute values of the output tuples to a set of destination processes (see Figure 9).

As an example, consider the two split operators shown in Table 1 in conjunction with the SQL query shown in Figure 10. Assume that three processes are used to execute the join operator, and that five other processes execute the two scan operators—three scanning partitions of relation A while two scan partitions of relation B. Each of the three relation A scan nodes will have the same split operator, sending all tuples between "A-H" to port 1 of join process 0, all between "I-Q" to port 1 of join process 1, and all between "R-Z" to port 1 of join process 2. Similarly the two relation B scan nodes have the same split operator except that their outputs are merged by port 1 (not port 0) of each join process. Each join process sees a sequential input stream of A tuples from the port 0 merge (the left-scan nodes) and another sequential stream of B tuples from the port 1 merge (the right-scan nodes). The outputs of each join are, in turn, split into three streams based on the partitioning criterion of relation C.

To clarify this example, consider the first join process in Figure 11 (processor 5, process 3, ports 0 and 1 in Table 1). It will receive all the relation A "A-H" tuples from the three relation A scan operators merged as a single stream on port 0, and will get all the "A-H" tuples from relation B merged as a single stream on port 1. It will join them using a hash-join, sort-merge join, or even a nested join if the tuples arrive in the proper order.

If each of these processes is on an independent processor with an independent disk, there will be little interference among them. Such dataflow designs are a natural application for shared-nothing machine architectures.

The split operator in Table 1 is just an example. Other split operators might duplicate the input stream, or partition it round-robin, or partition it by hash. The partitioning function can be an arbitrary program. Gamma, Volcano, and Tandem use this approach [14]. It has several advantages including the automatic parallelism of any new operator added to the system, plus support for many kinds of parallelism.

The split and merge operators have flow control and buffering built into them. This prevents one operator from getting too far ahead in the computation. When a split-operator's output buffers fill, it stalls the relational operator until the data target requests more output.

For simplicity, these examples have been stated in terms of an operator per process. But it is entirely possible to place several operators within a process to get coarser grained parallelism. The fundamental idea though is to build a self-pacing dataflow graph and distribute it in a shared-nothing machine in a way that minimizes interference.

Specialized Parallel Relational Operators. Some algorithms for relational operators are especially appropriate for parallel execution, either because they minimize data flow, or because they better tolerate data and execution skew. Improved algorithms have been found for most of the relational operators. The evolution of join operator algorithms is sketched here as an example of these improved algorithms.

Recall that the join operator combines two relations, A and B, to produce a third relation containing all tuple pairs from A and B with matching attribute values. The conventional way of computing the join is to sort both A and B into new relations ordered by the join attribute. These two intermediate relations are then compared in sorted order, and matching tuples are inserted in the output stream. This algorithm is called *sort-merge* join.

Many optimizations of sort-merge join are possible, but since sort has execution cost *nlog(n)*, sort-merge join has an *nlog(n)* execution cost. Sort-merge join works well in a parallel dataflow environment unless there is data skew. In case of data skew, some sort partitions may be much larger than others. This in turn creates execution skew and limits speedup and scaleup. These skew problems do not appear in centralized sort-merge joins.

Hash-join is an alternative to sort-merge join. It has linear execution cost rather than *nlog(n)* execution cost, and it is more resistant to data skew. It is superior to sort-merge join unless the input streams are already in sorted order. Hash join works as follows. Each of the relations A and B are first hash partitioned on the join attribute. A hash partition of relation A is hashed into memory. The corresponding partition of table relation B is scanned, and each tuple is compared against the main-memory hash table for the A partition. If there is a match, the pair of tuples are sent to the output stream. Each pair of hash partitions is compared

in this way.

The hash join algorithm breaks a big join into many little joins. If the hash function is good and if the data skew is not too bad, then there will be little variance in the hash bucket size. In these cases hash-join is a linear-time join algorithm with linear speedup and scaleup. Many optimizations of the parallel hash-join algorithm have been discovered over the last decade. In pathological skew cases, when many or all tuples have the same attribute value, one bucket may contain all the tuples. In these cases no algorithm is known to speedup or scaleup.

The hash-join example shows that new parallel algorithms can improve the performance of relational operators. This is a fruitful research area [4, 8, 18, 20, 25, 26, 38, 39]. Although parallelism can be obtained from conventional sequential relational algorithms by using split and merge operators, we expect that many new algorithms will be discovered in the future.

The State of the Art

Teradata

Teradata quietly pioneered many of the ideas presented in this article. Since 1978 they have been building shared-nothing highly-parallel SQL systems based on commodity microprocessors, disks, and memories. Teradata systems act as SQL servers to client programs operating on conventional computers.

Teradata systems may have over 1,000 processors and many thousands of disks. The Teradata processors are functionally divided into two groups: Interface Processors (IFPs) and Access Module Processors (AMPs). The IFPs handle communication with the host, query parsing and optimization, and coordination of AMPs during query execution. The AMPs are responsible for executing queries. Each AMP typically has several disks and a large memory cache. IFPs and AMPs are interconnected by a dual redundant, tree-shaped interconnect called the Y-net [33].

Each relation is hash partitioned over a subset of the AMPs. When a tuple is inserted into a relation, a hash function is applied to the primary key of the tuple to select an AMP for storage. Once a tuple arrives at an AMP, a second hash function determines the tuple's placement in its fragment of the relation. The tuples in each fragment are in hash-key order. Given a value for the key attribute, it is possible to locate the tuple in a single AMP. The AMP examines its cache, and if the tuple is not present, fetches it in a single disk read. Hash secondary indices are also supported.

Hashing is used to split the outputs of relational operators into intermediate relations. Join operators are executed using a parallel sort-merge algorithm. Rather than using pipelined parallel execution, during the execution of a query, each operator is run to completion on all participating nodes before the next operator is initiated.

Teradata has installed many systems containing over 100 processors and hundreds of disks. These systems demonstrate near-linear speedup and scaleup on relational queries, and far exceed the speed of traditional mainframes in their ability to process large (terabyte) databases.

Tandem NonStop SQL

The Tandem NonStop SQL system is composed of processor clusters interconnected via 4-plexed fiber-optic rings. Unlike most other systems discussed in this article, the Tandem systems run the applications on the same processors and operating system as the database servers. There is no front-end/back-end distinction between programs and machines. The systems are configured at a disk per MIPS, so each 10-MIPS processor has about 10 disks. Disks are typically duplexed [2]. Each disk is served by a set of processes managing a large shared RAM cache, a set of locks, and log records for the data on that disk pair. Considerable effort is spent on optimizing sequential scans by prefetching large units, and by filtering and manipulating the tuples with SQL predicates at these disk servers. This minimizes traffic on the shared interconnection network.

Relations may be range partitioned across multiple disks. Entry-sequenced, relative, and B-tree organizations are supported. Only B-tree secondary indices are supported. Nested join, sort-merge join, and hash join algorithms are provided. Parallelization of operators in a query plan is achieved by inserting split and merge operators between operator nodes in the query tree. Scans, aggregates, joins, updates, and deletes are executed in parallel. In addition, several utilities use parallelism (e.g., load, reorganize, . . .) [31, 39].

Tandem systems are primary designed for on-line transaction processing (OLTP)—running many simple transactions against a large shared database. Beyond the parallelism inherent in running many independent transactions in parallel, the main parallelism feature for OLTP is parallel index update. SQL relations typically have five indices on them, although it is not uncommon to see 10 indices on a relation. These indices speed reads, but slow down inserts, updates, and deletes. By doing the index maintenance in parallel, the maintenance time for multiple indices can be held almost constant if the indices are spread among many processors and disks.

Overall, the Tandem systems demonstrate near-linear scaleup on transaction processing workloads, and near-linear speedup and scaleup on large relational queries [10, 31].

Gamma

The current version of Gamma runs on a 32-node Intel iPSC/2 Hypercube with a disk attached to each node. In addition to round-robin, range and hash partitioning, Gamma also provides hybrid-range

partitioning that combines the best features of the hash and range partitioning strategies [12]. Once a relation has been partitioned, Gamma provides both clustered and nonclustered indices on either the partitioning or nonpartitioning attributes. The indices are implemented as B-trees or hash tables.

Gamma uses split and merge operators to execute relational algebra operators using both parallelism and pipelining [9]. Sort-merge and three different hash join methods are supported [7]. Near-linear speedup and scaleup for relational queries has been measured on this architecture [9, 25, 26].

The Super Database Computer

The Super Database Computer (SDC) project at the University of Tokyo presents an interesting contrast to other database systems [16, 20]. SDC takes a combined hardware and software approach to the performance problem. The basic unit, called a processing module (PM), consists of one or more processors on a shared memory. These processors are augmented by a special-purpose sorting engine that sorts at high speed (3MB/second at present), and by a disk subsystem [19]. Clusters of processing modules are connected via an omega network that provides both nonblocking NxN interconnect and some dynamic routing minimize skewed data distribution during hash joins. The SDC is designed to scale to thousands of PMs, and so considerable attention is paid to the problem of data skew.

Data is partitioned among the PMs by hashing. The SDC software includes a unique operating system, and a relational database query executor. The SDC is a shared-nothing design with a software dataflow architecture. This is consistent with our assertion that current parallel database machines systems use conventional hardware. But the special-purpose design of the omega network and of the hardware sorter clearly contradict the thesis that special-purpose hardware is not a good investment of development resources. Time will tell whether these special-purpose components offer better price performance or peak performance than shared-nothing designs built of conventional hardware.

Bubba

The Bubba prototype was implemented using a 40-node FLEX/32 multiprocessor with 40 disks [4]. Although this is a shared-memory multiprocessor, Bubba was designed as a shared-nothing system and the shared-memory is only used for message passing. Nodes are divided into three groups: Interface Processors for communicating with external host processors and coordinating query execution; Intelligent Repositories for data storage and query execution; and Checkpoint/Logging Repositories. While Bubba also uses partitioning as a storage mechanism (both range and hash partitioning mechanisms are provided) and dataflow processing mechanisms, Bubba is unique in several ways. First, Bubba uses FAD rather than SQL as its interface language. FAD is an extended-relational persistent programming language. FAD provides support for complex objects via several type constructors including shared subobjects, set-oriented data manipulation primitives, and more traditional language constructs. The FAD compiler is responsible for detecting operations that can be executed in parallel according to how the data objects being accessed are partitioned. Program execution is performed using a dataflow execution paradigm. The task of compiling and parallelizing a FAD program is significantly more difficult than parallelizing a relational query. Another Bubba feature is its use of a single-level store mechanism in which the persistent database at each node is mapped to the virtual memory address space of each process executing at the node. This is in contrast to the traditional approach of files and pages. Similar mechanisms are used in IBM's AS400 mapping of SQL databases into virtual memory, HP's mapping of the Image Database into the operating system virtual address space, and Mach's mapped file [34] mechanism. This approach simplified the implementation of the upper levels of the Bubba software.

Other Systems

Other parallel database system prototypes include XPRS [30], Volcano [14], Arbre [21], and the PERSIST project under development at IBM Research Labs in Hawthorne and Almaden. While both Volcano and XPRS are implemented on shared-memory multiprocessors, XPRS is unique in its exploitation of the availability of massive shared-memory in its design. In addition, XPRS is based on several innovative techniques for obtaining extremely high performance and availability.

Recently, the Oracle database system has been implemented atop a 64-node nCUBE shared-nothing system. The resulting system is the first to demonstrate more than 1,000 transactions per second on the industry-standard TPC-B benchmark. This is far in excess of Oracle's performance on conventional mainframe systems—both in peak performance and in price/performance [13].

The NCR Corporation has announced the 3600 and 3700 product lines that employ shared-nothing architectures running System V R4 of Unix on Intel 486 and 586 processors. The interconnection network for the 3600 product line uses an enhanced Y-Net licensed from Teradata while the 3700 is based on a new multistage interconnection network being developed jointly by NCR and Teradata. Two software offerings have been announced. The first, a port of the Teradata software to a Unix environment, is targeted toward the decision-support marketplace. The second, based on a parallelization of the Sybase DBMS, is intended primarily for transaction processing workloads.

Database Machines and Grosch's Law

Today shared-nothing database machines have the best peak performance and best price performance available. When compared to traditional mainframes, the Tandem system scales linearly well beyond the largest reported mainframes on the TPC-A transaction processing benchmark. Its price/performance on these benchmarks is three times cheaper than the comparable mainframe numbers. Oracle on an nCUBE has the highest reported TPC-B numbers, and has very competitive price performance [13, 36]. These benchmarks demonstrate linear scaleup on transaction processing benchmarks.

Gamma, Tandem, and Teradata have demonstrated linear speedup and scaleup on complex relational database benchmarks. They scale well beyond the size of the largest mainframes. Their performance and price performance is generally superior to mainframe systems.

These observations defy Grosch's law. In the 1960s, Herb Grosch observed that there is an economy-of-scale in computing. At that time, expensive computers were much more powerful than inexpensive computers. This gave rise to super-linear speedups and scaleups. The current pricing of mainframes at $25,000/MIPS and $1,000/MB of RAM reflects this view. Meanwhile, microprocessors are selling for $250/MIPS and $100/MB of RAM.

By combining hundreds or thousands of these small systems, one can build an incredibly powerful database machine for much less money than the cost of a modest mainframe. For database problems, the near-linear speedup and scaleup of these shared-nothing machines allows them to outperform current shared-memory and shared disk mainframes.

Grosch's law no longer applies to database and transaction processing problems. There is no economy of scale. At best, one can expect linear speedup and scaleup of performance and price/performance. Fortunately, shared-nothing database architectures achieve this near-linear performance.

Future Directions and Research Problems

Mixing Batch and OLTP Queries

The second section of this article, "Basic Techniques for Parallel Database Machine Implementation", concentrated on the basic techniques used for processing complex relational queries in a parallel database system. Concurrently running a mix of both simple and complex queries concurrently presents several unsolved problems.

One problem is that large relational queries tend to acquire many locks and tend to hold them for a relatively long time. This prevents concurrent updates of the data by simple on-line transactions. Two solutions are currently offered: give the ad-hoc queries a fuzzy picture of the database, not locking any data as they browse it. Such a "dirty-read" solution is not acceptable for some applications. Several systems offer a versioning mechanism that gives readers a consistent (old) version of the database while updators are allowed to create newer versions of objects. Other, perhaps better, solutions for this problem may also exist.

Priority scheduling is another mixed-workload problem. Batch jobs have a tendency to monopolize the processor, flood the memory cache, and make large demands on the I/O subsystem. It is up to the underlying operating system to quantize and limit the resources used by such batch jobs to ensure short response times and low variance in response times for short transactions. A particularly difficult problem is the *priority inversion problem,* in which a low-priority client makes a request to a high-priority server. The server must run at high priority because it is managing critical resources. Given this, the work of the low-priority client is effectively promoted to high priority when the low-priority request is serviced by the high-priority server. There have been several ad-hoc attempts at solving this problem, but considerably more work is needed.

Parallel Query Optimization

Current database query optimizers do not consider all possible plans when optimizing a relational query. While cost models for relational queries running on a single processor are now well-understood [27] they still depend on cost estimators that are a guess at best. Some dynamically select from among several plans at run time depending on, for example, the amount of physical memory actually available and the cardinalities of the intermediate results [15]. To date, no query optimizers consider all the parallel algorithms for each operator and all the query tree organizations. More work is needed in this area.

Another optimization problem relates to highly skewed value distributions. Data skew can lead to high variance in the size of intermediate relations, leading to both poor query plan cost estimates and sub-linear speedup. Solutions to this problem are an area of active research [17, 20, 37, 38].

Application Program Parallelism

The parallel database systems offer parallelism within the database system. Missing are tools to structure application programs to take advantage of parallelism inherent in these parallel systems. While automatic parallelization of applications programs written in COBOL may not be feasible, library packages to facilitate explicitly parallel application programs are needed. Ideally the SPLIT and MERGE operators could be packaged so that applications could benefit from them.

Physical Database Design

For a given database and workload there are many possible indexing and partitioning combinations. Database design tools are needed to help the database administrator select among these many design options. Such tools might accept as input a description of the queries comprising the workload, their frequency of execution, statistical information about the relations in the database, and a description of the

processors and disks. The resulting output would suggest a partitioning strategy for each relation plus the indices to be created on each relation. Steps in this direction are beginning to appear.

Current algorithms partition relations using the values of a single attribute. For example, geographic records could be partitioned by longitude or latitude. Partitioning on longitude allows selections for a longitude range to be localized to a limited number of nodes, selections on latitude must be sent to all the nodes. While this is acceptable in a small configuration, it is not acceptable in a system with thousands of processors. Additional research is needed on multidimensional partitioning and search algorithms.

On-line Data Reorganization and Utilities

Loading, reorganizing, or dumping a terabyte database at a megabyte per second takes over 12 days and nights. Clearly parallelism is needed if utilities are to complete within a few hours or days. Even then, it will be essential that the data be available while the utilities are operating. In the SQL world, typical utilities create indices, add or drop attributes, add constraints, and physically reorganize the data, changing its clustering.

One unexplored and difficult problem is how to process database utility commands while the system remains operational and the data remains available for concurrent reads and writes by others. The fundamental properties of such algorithms are that they must be *online* (operate without making data unavailable), *incremental* (operate on parts of a large database), *parallel* (exploit parallel processors), and *recoverable* (allow the operation to be canceled and return to the old state).

Summary and Conclusions

Like most applications, database systems want cheap, fast hardware. Today that means commodity processors, memories, and disks. Consequently, the hardware concept of a *database machine* built of exotic hardware is inappropriate for current technology. On the other hand, the availability of fast microprocessors, and small inexpensive disks packaged as standard inexpensive but fast computers is an ideal platform for *parallel database systems.* A shared-nothing architecture is relatively straightforward to implement and, more importantly, has demonstrated both speedup and scaleup to hundreds of processors. Furthermore, shared-nothing architectures actually simplify the software implementation. If the software techniques of data partitioning, dataflow, and intra-operator parallelism are employed, the task of converting an existing database management system to a highly parallel one becomes relatively straightforward. Finally, there are certain applications (e.g., data mining in terabyte databases) that require the computational and I/O resources available only from a parallel architecture.

While the successes of both commercial products and prototypes demonstrate the viability of highly parallel database machines, several research issues remain unsolved including techniques for mixing ad-hoc queries with on-line transaction processing without seriously limiting transaction throughput, improved optimizers for parallel queries, tools for physical database design, on-line database reorganization, and algorithms for handling relations with highly skewed data distributions. Some application domains are not well supported by the relational data model. It appears that a new class of database systems based on an object-oriented data model is needed. Such systems pose a host of interesting research problems that require further examination. ■

References

1. Alexander, W., et al. Process and dataflow control in distributed data-intensive systems. In *Proceedings of ACM SIGMOD Conference* (Chicago, Ill., June 1988) ACM, NY, 1988.
2. Bitton, D. and Gray, J. Disk shadowing. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases* (Los Angeles, Calif., August, 1988).
3. Boral, H. and DeWitt, D. Database machines: An idea whose time has passed? A critique of the future of database machines. In *Proceedings of the 1983 Workshop on Database Machines.* H.-O. Leilich and M. Missikoff, Eds., Springer-Verlag, 1983.
4. Boral, H. et al. Prototyping Bubba: A highly parallel database system. *IEEE Knowl. Data Eng. 2,* 1, (Mar. 1990).
5. Codd, E.F. A relational model of data for large shared databanks. *Commun. ACM 13,* 6 (June 1970).
6. Copeland, G., Alexander, W., Boughter, E., and Keller, T. Data placement in Bubba. In *Proceedings of ACM-SIGMOD International Conference on Management of Data* (Chicago, May 1988).
7. DeWitt, D.J., Katz, R., Olken, F., Shapiro, D., Stonebraker, M. and Wood, D. Implementation techniques for main memory database systems. In *Proceedings of the 1984 SIGMOD Conference,* (Boston, Mass., June, 1984).
8. DeWitt, D., et al. GAMMA—A high performance dataflow database machine. In *Proceedings of the 1986 VLDB Conference* (Japan, August 1986).
9. DeWitt, D., et al. The Gamma database machine project. *IEEE Knowl. Data Eng. 2,* 1 (Mar. 1990).
10. Engelbert, S, Gray, J., Kocher, T., and Stah, P. A benchmark of nonstop SQL Release 2 demonstrating near-linear speedup and scaleup on large databases. Tandem Computers, Technical Report 89.4, Tandem Part No. 27469, May 1989.
11. Ghandeharizadeh, S., and DeWitt, D.J. Performance analysis of alternative declustering strategies. In *Proceedings of the Sixth International Conference on Data Engineering* (Feb. 1990).
12. Ghandeharizadeh, S., and Dewitt, D.J. Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. In *Proceedings of the Sixth International Conference on Very Large Data Bases,* (Melbourne, Australia, Aug. 1990).
13. Gibbs, J. Massively parallel systems, rethinking computing for business and science. *Oracle 6,* 1 (Dec. 1991).
14. Graefe, G. Encapsulation of parallelism in the Volcano query processing system. In *Proceedings of 1990*

ACM-SIGMOD International Conference on Management of Data (May 1990).

15. Graefe, G., and Ward, K. Dynamic query evaluation plans. In *Proceedings of the 1989 SIGMOD Conference*, (Portland, Ore., June 1989).
16. Hirano, M.S. et al. Architecture of SDC, the super database computer. In *Proceedings of JSPP '90*. 1990.
17. Hua, K.A. and Lee, C. Handling data skew in multiprocessor database computers using partition tuning. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*. (Barcelona, Spain, Sept. 1991).
18. Kitsuregawa, M., Tanaka, H., and Moto-oka, T. Application of hash to data base machine and its architecture. *New Generation Computing 1*, 1 (1983).
19. Kitsuregawa, M., Yang, W., and Fushimi, S. Evaluation of 18-stage pipeline hardware sorter. In *Proceedings of the Third International Conference on Data Engineering* (Feb. 1987).
20. Kitsuregawa, M., and Ogawa, Y. A new parallel hash join method with robustness for data skew in super database computer (SDC). In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*. (Melbourne, Australia, Aug. 1990).
21. Lorie, R., Daudenarde, J., Hallmark, G., Stamos, J., and Young, H. Adding intra-transaction parallelism to an existing DBMS: Early experience. *IEEE Data Engineering Newsletter 12*, 1 (Mar. 1989).
22. Patterson, D. A., Gibson, G. and Katz, R. H. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*. (Chicago, May 1988).
23. Ries, D. and Epstein, R. Evaluation of distribution criteria for distributed database systems. UBC/ERL Technical Report M78/22, UC Berkeley, May, 1978.
24. Salem, K. and Garcia-Molina, H. Disk-striping. Department of Computer Science, Princeton University Technical Report EEDS-TR-322-84, Princeton, N.J., Dec. 1984.
25. Schneider, D. and DeWitt, D. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the 1989 SIGMOD Conference* (Portland, Ore., June 1989).
26. Schneider, D. and DeWitt, D. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*. (Melbourne, Australia, Aug., 1990).
27. Selinger P. G., et al. Access path selection in a relational database management system. In *Proceedings of the 1979 SIGMOD Conference* (Boston, Mass., May 1979).
28. Stonebraker, M. Muffin: A distributed database machine. ERL Technical Report UCB/ERL M79/28, University of California at Berkeley, May 1979.
29. Stonebraker, M. The case for shared nothing. *Database Eng. 9*, 1 (1986).
30. Stonebraker, M., Katz, R., Patterson, D., and Ousterhout, J. The design of XPRS. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases*. (Los Angeles, Calif., Aug. 1988).
31. Tandem Database Group. NonStop SQL, a distributed, high-performance, high-reliability implementation of SQL. Workshop on High Performance Transaction Systems, Asilomar, CA, Sept. 1987.
32. Tandem Performance Group. A benchmark of non-stop SQL on the debit credit transaction. In *Proceedings of the 1988 SIGMOD Conference* (Chicago, Ill., June 1988).
33. Teradata Corporation. DBC/1012 Data Base Computer Concepts & Facilities. Document No. C02-0001-00, 1983.
34. Tevanian, A., et al. A Unix interface for shared memory and memory mapped files under Mach. Dept. of Computer Science Technical Report, Carnegie Mellon University, July, 1987.
35. Thakkar, S.S. and Sweiger, M. Performance of an OLTP application on symmetry multiprocessor system. In *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture*. (Seattle, Wash., May, 1990).
36. *The Performance Handbook for Database and Transaction Processing Systems*. J. Gray, Ed., Morgan Kaufmann, San Mateo, Ca., 1991.
37. Walton, C.B., Dale, A.G., and Jenevein, R.M. A taxonomy and performance model of data skew effects in parallel joins. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*. (Barcelona, Spain, Sept. 1991).
38. Wolf, J.L., Dias, D.M., and Yu, P.S. An effective algorithm for parallelizing sort-merge joins in the presence of data skew. In *Proceedings of the Second International Symposium on Parallel and Distributed Systems*. (Dublin, Ireland, July, 1990).
39. Zeller, H.J. and Gray, J. Adaptive hash joins for a multiprogramming environment. In *Proceedings of the 1990 VLDB Conference* (Australia, Aug. 1990).

CR Categories and Subject Descriptors: B.5.1 [**Register-Transfer-Level Implementation**]: Design-style (e.g., parallel, pipelined, special-purpose); C.1.2 [**Computer Systems Organization**]: Processor Architectures—*Multiple Data Stream Architectures (Multiprocessors)*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*Parallelism*; H.2.1 [**Information Systems**]: Database Management—*Logical design*; H.2.8 [**Information Systems**]: Database Management—*Database Applications*; H.3 [**Information Systems**]: Information Storage and Retrieval

General Terms: Design, Measurement

Additional Keywords and Phrases: Parallelism, parallel database systems, parallel processing systems.

About the Authors:

DAVID DEWITT is a professor in the Computer Sciences Department at the University of Wisconsin. His current research interests include parallel database systems, object-oriented database systems, and database performance evaluation. **Author's Present Address:** Computer Sciences Department, University of Wisconsin, 1210 West Dayton Street, Madison, WI 53706; email: dewitt@cs.wisc.edu

JIM GRAY is a staff member with the Digital Equipment Corporation. His current research interests include databases, transaction processing, and computer architecture. **Author's Present Address:** San Francisco Systems Center, Digital Equipment Corporation, 455 Market Street—7th Floor, San Francisco, CA 94105-2403; email: gray@sfbay.enet.dec.com

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00039-86-C-0578, by the National Science Foundation under grant DCR-8512862, and by research grants from Digital Equipment Corporation, IBM, NCR, Tandem, and Intel Scientific Computers.