Parallelism in relational database management systems

by C. Mohan
H. Pirahesh
W. G. Tang
Y. Wang

In order to provide real-time responses to complex queries involving large volumes of data, it has become necessary to exploit parallelism in query processing. This paper addresses the issues and solutions relating to intraquery parallelism in a relational database management system (DBMS). We provide a broad framework for the study of the numerous issues that need to be addressed in supporting parallelism efficiently and flexibly. The alternatives for a parallel architecture system are discussed, followed by the focus on how a query can be parallelized and how that affects load balancing of the different tasks created. The final part of the paper contains information about how the IBM DATABASE 2™ (DB2®) Version 3 product provides support for I/O parallelism to reduce response time for data-intensive queries.

The widespread adoption of the easy-to-use products of relational database technology has led to the expectation that responses to queries should be received faster than before, especially because the queries may be posed by a user at a terminal rather than by a batch program, as in the past. Although high-level ad hoc query languages like SQL (Structured Query Language) are used to access the database management system (DBMS) to generate complex reports, volumes of data have grown rapidly, resulting in queries becoming data-intensive and complex.

Solutions to reduce the complexity of query processing and improve the response time of queries include moving additional function into the query languages and exploiting parallelism of both the hardware and the software processing.

This paper explores two main topics in this environment. First, in sections on overall system architecture options and parallel algorithms, the use of parallelism is discussed in the architecture, in processing queries, and in various relational operators. Second, the implementation of I/O parallelism in the IBM DATABASE 2* (DB2*) Version 3 product to reduce response time for data-intensive and I/O bound queries is described in the section on I/O parallelism in DB2 Version 3. Readers of this paper who only want to understand how DB2 Version 3 exploits parallelism may skip directly to this section.

Background

Trends in the environment and where some of the solutions may be found are important to understand as parallelism is exploited.

Volumes of data. Today there are customers who would like to store more than 100 gigabytes of

data in a single table and keep the data on line and readily available.[1] The amount of data kept in a single large relational database is expected to be in the terabyte range in the coming decade, causing queries to become extremely data-intensive. Furthermore, there is growing emphasis on supporting newer, nontraditional database applications, such as Computer-Aided Software Engineering (CASE), geographical information systems (GIS), and multimedia applications, where the volumes of data are enormous compared to those in traditional business data processing.

Complexity of queries. The complexity of the queries that are being posed is also growing as a result of competition intensifying in various sectors of the economy and direct-mail marketing becoming more and more common. Ad hoc interactions with the new generation DBMSs are commonly performed through high-level user interfaces, allowing complex queries to be specified very easily by users, where the users may not even be aware of the complexity of their requests. Often, a high-level interface query results in many complex DBMS queries, which must have a short response time due to the interactive nature of the user interface. This increases both the complexity and the traffic rate of DBMS queries. The same phenomenon occurs in interfaces between high-level programming languages, such as visual query generators and visual fourth-generation languages (4GLs). These programming environments allow programmers to write applications that initiate many complex DBMS queries where those queries become logic-intensive.

Even though the processing power of affordable parallel computers is expected to be over 1000 million instructions per second (MIPS) shortly, the combination of massive amounts of data plus enormous processing power still creates the environment for much more complex queries. Hence, we expect that future DBMSs will have to deal with applications that are both data-intensive and logic-intensive.

Solution areas. We expect the functionality provided by such query languages to grow considerably. Today's relational query languages typically do not have the functions for statistical analysis and structural (complex objects, record structures, etc.) expressibility, which are crucial for data summation and engineering databases, respectively. More of the application logic will be moved inside the DBMS, both for better performance (bringing function to data) and for better sharing of data among applications (better protection of data by encapsulation).[2] Given that applications tend to be sequential, applying the complex search predicates (record selection criteria) within the DBMS would allow parallelism to be exploited in evaluating those predicates also, thereby potentially reducing the response time tremendously.

DBMSs will also have to deal with a much larger set of data types and operations. From the application performance viewpoint, this is valuable since it allows more type-specific operations to be specified in search predicates, so that massive amounts of irrelevant data do not have to pass through the different layers of the DBMS to the applications. This is particularly significant since the data rate of the output from DBMSs is typically much less than the data rate of storage devices from which data are retrieved. Operations such as outer join, recursion, and sampling should be handled by DBMSs for the same reason.

Exploiting parallelism. The limitations to the improvement of response time via faster processors and larger memories alone lead us to believe that in most cases, one can hope to get real-time responses to data- and logic-intensive queries only by exploiting parallelism.

Limitations in not using parallelism. The following are observations that support our premise:

- Based on the trends of the recent past, it is expected that the growth in the processing capacity of a uniprocessor or a closely-coupled multiprocessor is not going to be sufficient to provide real-time responses to certain types of complex queries using such systems. At least today, it appears that the dollars per MIPS ($/MIPS) cost of the very powerful machines is much higher than the $/MIPS cost of smaller, microprocessor-based machines.
- Even though the price of main memory keeps declining rapidly and the sizes of the memories that are attachable to a single processor keep growing, the volume of data to be handled keeps growing also. Further, with some architectures, there are limits on the amount of main memory that may be attached to a single machine (e.g., 2 gigabytes of real memory due to

the 31-bit real memory addressing used on the IBM System/370*).

- As the processors become more and more powerful (even in the smaller microprocessor-based machines), the gap between the CPU processing speed and the I/O capacity of a single device becomes wider and wider. This is at present necessitating the use of techniques like disk striping[3] (spreading a single file across multiple disks) and disk arrays[4] to improve the I/O bandwidth. For a long time, systems like the IBM Transaction Processing Facility (TPF)[5] used disk striping in software to improve intertransaction parallelism. But now, striping is needed to support intratransaction and query parallelism as well. Disk striping, if done in software, already demands parallelism at least at the I/O level to access the multiple disk in parallel.

Support for the use of parallelism. The problems that the query optimization and the query execution logic must handle are expanding because the nature of the queries that DBMSs must handle is expanding. Exploiting parallelism will provide solutions that will overcome the limitations previously mentioned. This may come as a surprise to some people who might be led to think that the way to address the response time requirement is to stay with the simpler strategy of no intraquery parallelism, faster processors, and larger and larger amounts of memory. But in order to gain price-performance advantages and response time improvements, the trend is toward building a system consisting of many machines and exploiting intraquery parallelism.

Overall system architecture options

In building a parallel system, many objects exist with respect to how different components are interconnected. In this section we discuss some of the system architecture possibilities.

Shared data, nothing, or everything. One approach to improving the capacity and availability characteristics of a single-system DBMS is to use multiple systems. There are three major architectures in use in the multisystem environment as shown in Figure 1:[6] (1) *shared disks* or data sharing,[7–12] (2) *shared nothing* or partitioned data,[13,14] and (3) *shared everything*.

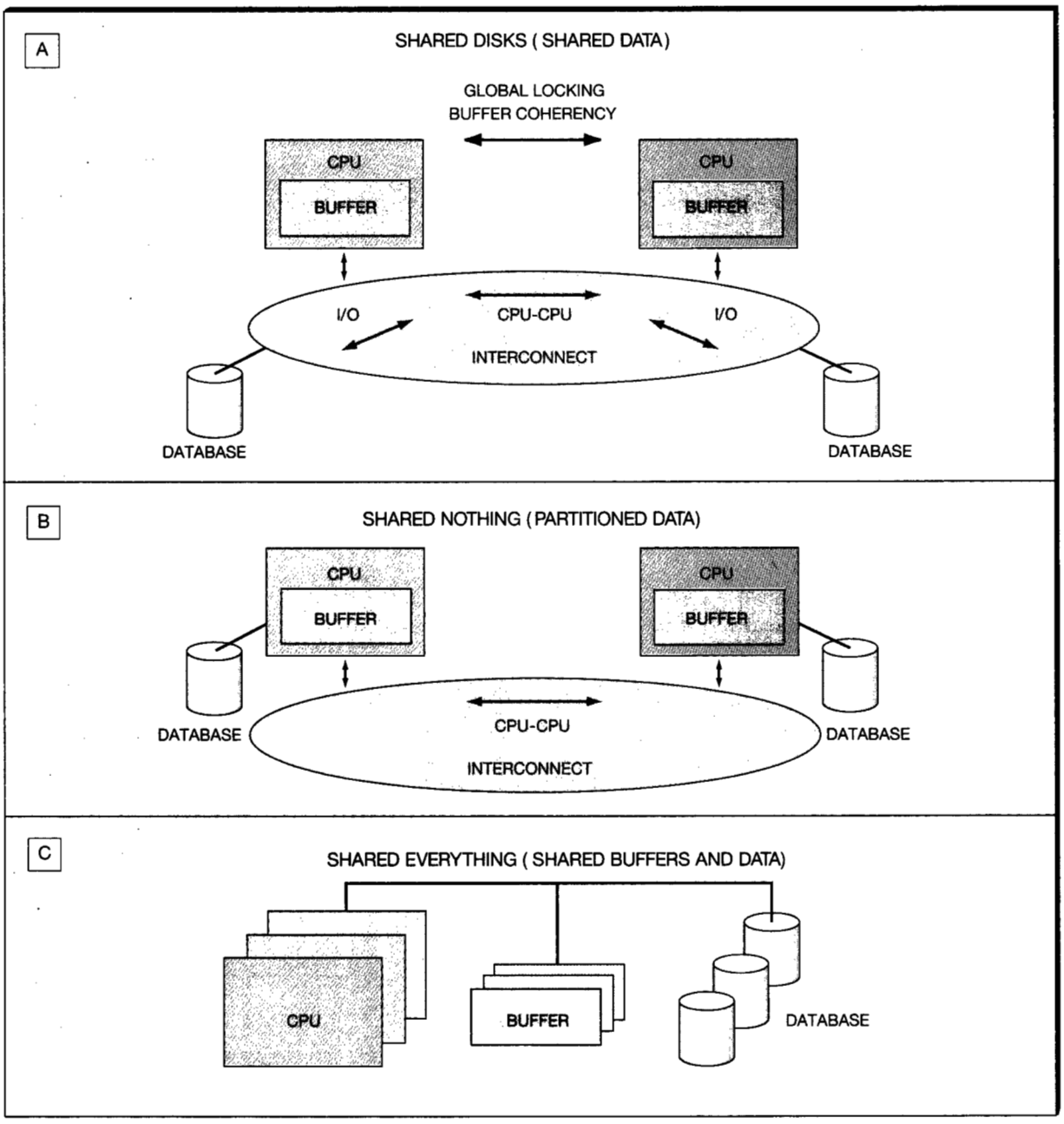
With shared disks (SD) all the disks containing the databases are shared among the different systems and each system has its own buffer pool (see Figure 1A). Every system that has an instance of the DBMS executing on it may access and modify any portion of the database on the shared disks. Since each instance has its own buffer pool and because conflicting accesses to the same data may be made from different systems, the interactions among the systems must be controlled via various synchronization protocols. This necessitates global locking and protocols for the maintenance of buffer coherency. SD is the approach used in the IBM IMS/VS[15,16] (Information Management System/Virtual Storage), TPF,[5] and DB2,[8–10,17] in the Digital Equipment Corp. RDB/VMS**,[11] and in the Oracle Corp. Oracle** Parallel Server. These systems are using the SD architecture for *inter*-transaction parallelism rather than *intra*-transaction parallelism.

With shared nothing (SN), each system owns a portion of the database and only that portion may be directly read or modified by that system (see Figure 1B). That is, the database is partitioned among the multiple systems. The kind of synchronization protocols mentioned before for SD are not needed for SN. But a transaction accessing data in multiple systems would need a form of two-phase commit protocol (e.g., the Presumed Abort protocol[18,19]) to coordinate its activities. This is the approach taken in the Tandem NonStop SQL**,[20,21] the Teradata Corp. DBC/1012**,[22] the Microelectronics and Computer Technology Corp. (MCC) Bubba,[23–25] the IBM Almaden Research ARBRE project,[26] and the University of Wisconsin Gamma.[3,27]

In the shared everything (SE) approach, memory, in addition to disks, is also shared across the processors (see Figure 1C). The University of California at Berkeley XPRS system has adopted this approach.[28,29] It has been pointed out[29] that SE has scalability problems. But it is attractive *within* a node of an SD or SN system. It helps reduce the number of nodes, making system management and load balancing easier. DB2,[30] for example, is able to very nicely exploit an SE machine like an IBM ES/9000* Model 900 in the 9021 family, which has six processors. Further, this architecture provides an effective basis for the implementation of DB2 parallelism, as discussed in the section on I/O parallelism in DB2.

Arguments in favor of SD are given[31] in the context of complex objects and parallelism. For com-

Figure 1 Architectural alternatives



plex objects, it is said that partitioning the data, as is required with SN, is a big problem.

Transaction monitors. In discussing an overall architecture, the role of data communications[32,33] and the transaction monitor (like IMS/DC[34] [Information Management System/Data Communications] or CICS*[35] [Customer Information Control System]) cannot be ignored. Most online transactions are executed in the environment of a

transaction monitor. The monitors provide support for terminal interactions, message queue management, logging, program libraries, etc.[36] They are in essence an extension of the base operating system.

Supporting the transaction monitor and the environment that it needs is essential even in a parallel architecture system. Any existing large application base that relies on such an environment must be accounted for. Resources (CPU, I/O, communication) used in the non-DBMS part of transactions (i.e., in transaction monitors and applications) are very significant. Hence, it is important to provide a parallel environment for both applications and transaction monitors. The Tandem NonStop SQL provides such an environment. This is the so-called *peer-peer* configuration.

If the adopted approach is one in which the monitor would run on one or more front-end machines (machines running the application code) and the actual data management would be done in a back-end (database) machine (the so-called *front-end–back-end* configuration) where parallelism would be exploited using machines of a different nature from the front-end machines, then two issues must be addressed. First, the cost of the interactions between the front end and the back end must be considered when evaluating the performance implications of this approach on the transaction workload. This division of labor between the front end and the back end is bound to increase the overall path length of a transaction. This increase will be felt especially in the case of the short transactions of the transaction workload. One way to address this problem is to support the notion of stored procedures and make the front-end issue a single call to the back end to execute a sequence of SQL statements.

The second issue with the front-end–back-end configuration is related to pushing more application functions down into the lower layers of the DBMS, either in the form of operations on abstract data types, function libraries (for scientific routines, statistical routines, etc.), methods on objects stored in the database (as in the object-oriented DBMSs), or rules (as in rule-based systems). This trend essentially fosters a more uniform runtime environment for applications and DBMSs, thereby allowing functions to move from applications into DBMS more easily. As a result, it may not be a good idea to have a very special-purpose operating system in the back end.

Interconnection technologies and requirements. The technology used for interconnecting the processors and the storage devices plays a crucial role in determining the communication bandwidth that can be sustained between the processors themselves, and between the processors and the storage devices. While fiber-optic[37] switches can sustain high bandwidths and cover more distances compared to copper interconnects, the costs of fiber-optic interface and switching devices are still rather high.

In the case of the SD approach, the storage devices must be attached through a switch since any processor must be capable of accessing any of the devices. This means that the switch should support high bandwidth communication. The processor-to-processor communications will be less in this environment, if parallelism for a given transaction is going to be handled within a system by utilizing a multiprocessor like the six-way IBM ES/9000 Model 900 (in the 9021 family). Most of the processor-to-processor communication is likely to be messages relating to global locking and buffer coherency protocols.[8–11,15]

With SN, the devices may be locally attached to the *owning* processors, perhaps using less expensive technologies. In this case, the processor-to-processor communications can be significant if a given complex query is accessing data owned by multiple systems, and the extracted data must be sent to other processors to perform operations such as join. The Teradata database machine uses a specially designed interconnection network called the Ynet, which can connect up to 1024 microprocessors.[22] To provide fault tolerance, the system actually includes two completely independent Ynet structures. When both Ynets are operational, message traffic is equally divided between the two. The Ynet is also able to sort the data as they flow through it.

Short transactions and complex queries. It is very important that the system architecture chosen can accommodate complex queries as well as short transactions against the *same* data. That is, it should be possible to pose ad hoc queries against the same data on which the "bread and butter" applications of the customers [the applications that financially support the business] are

also performing on-line, short transactions that may be updating as well as reading the data. The former is called the *query workload* and the latter is called the *transaction workload*. In modern applications, the transaction workload transfers most of the new data from the real world into databases. Hence, it is the *producer* of the data from the database viewpoint. Examples are transactions originating from automated teller machines (ATMs), point-of-sale transactions, and stock exchange transactions. Complex queries are usually *consumers* of data. Sharing between producers and consumers of data is a fundamental phenomenon. Good performance for the transaction workload must be guaranteed since those transactions have more stringent response time constraints.

Traditionally, users have been forced to deal with this problem of handling the transaction and query workloads properly by maintaining two different databases on two different systems. One of the databases is the one most up-to-date and it is against that one that the transaction workload is run. The other database is an extracted version of the first one and it is on this extracted database that the complex queries are executed. Not all users are happy with this solution. In addition to the problems of having to maintain two different systems, the disk storage requirements are doubled for the data that are replicated.[38] Additionally, there is the expensive extraction process that needs to be performed periodically and that only gives out-of-date data to the ad hoc query users. Some of the advantages of this two-database strategy are: (1) the two types of workloads are on different machines and hence could hopefully be more easily managed, and (2) since the second database is a read-only one, different access paths and buffer management policies (or even a different DBMS) may be defined for it to improve the performance of complex queries. Some of these users with dual databases may have an IMS system that is running the older transaction workload and from which they are unable to migrate away quickly, due to performance and application rewrite cost. They may extract data from such a system and put it into a DB2 or Teradata system for the benefit of their newer decision support applications.

When both sets of workloads are brought into the same system, great care must be exercised to ensure that the exploitation of parallelism by the complex queries does not consume too much resource (CPU, I/O, and memory) at the expense of the short transactions. This requires that the system, at the least, support a priority concept for treating different users or database requests differently. Some server-based systems do not have such a concept, which leads to very unpredictable response times and wide variances. DB2 Version 3 has added a capability to control the amount of resources used by parallel queries. We discuss this in the section on I/O parallelism in DB2 Version 3. A resource governor would also be essential to control "runaway" queries. DB2 Version 2, Release 1 for example, introduced such a governor for controlling the resource consumption of dynamic SQL queries.

There is also a concurrency versus locking overhead dilemma with respect to mixing these workloads with very different characteristics. In order to maximize concurrency for the transaction workload, the developers of the application would be highly tempted to choose fine-granularity (e.g., record) locking.[39] But this will make the query workload incur significant locking overhead since queries in general access large numbers of records. Apart from the overhead concern, the major problem may be that the locks held by the complex queries will delay the transaction workload from performing updates. Typically, this problem is dealt with by executing the complex queries with the isolation level of *cursor stability*. That is, the read locks are given up as soon as the cursor moves from one record to the next. Even though many DBMSs (like DB2, DB2/2*[17] and NonStop SQL) support cursor stability, the research literature has concentrated only on repeatable read. More implications of cursor stability on data accesses have been discussed in References 40–42.

The locking path length overhead problem is normally addressed using different solutions, with each one compromising on some functionality or the other. Two of the solutions are: *unlocked reads* and *transient versioning*.

Unlocked reads. With unlocked reads, the queries are run without locking and use latches[39,40] to assure physical consistency of the pages being read. IMS supports this type of access via what is called *GO processing*. Relational systems like the Tandem NonStop SQL, and the IBM Application System/400* (AS/400*) and DB2/2 also support such

accesses. This solution avoids not only the locking overhead but also the undesirable lock conflicts between the two types of workloads. This approach has the disadvantage that uncommitted data may be exposed to the transactions that are not obtaining locks. In particular, integrity constraint violations may be noticed by the unlocked readers. For statistical queries (e.g., market analysis queries), this exposure usually causes little or no problem. But there is a concern regarding queries dealing with structured (e.g., computer-aided design/computer-aided manufacturing, or CAD/CAM) objects, where inconsistent data close to the root of the object may result in retrieving a very different, and possibly invalid set of objects close to the leaves (as in a tree structure). In fact, this problem, to a lesser degree, also occurs with cursor stability. Retrieval of the children (as in a parent-child relationship for the set of objects) at two different times during the course of a query may result in two different sets since the read data are locked only briefly and the data might have been updated in between the two retrievals.

Transient versioning. In the transient versioning approach, for data that are being modified, one or more older versions of the data may be maintained.[43] With this support, the query workload would be able to read without locking. Just for data that are being modified, a slightly older but committed version of that data will be exposed to such transactions. The advantage is that the database that is being exposed will be internally consistent. Concerns may be that not all the exposed data are up to date and that there is a slight increase in storage consumption and complexity to keep multiple copies of some of the data. But the major problem may be that typically in such schemes the transactions that are not locking are not allowed to do any updates and such transactions must declare themselves to be read-only.

In References 44 and 45, a technique called *Commit_LSN* is presented for eliminating, most of the time, the need for locking when cursor stability accesses are made. This technique takes advantage of some information (e.g., the log sequence number[39]) that is tracked, for recovery purposes, on every page to conclude (without locking) that all the data in a page are in the committed state. It helps in reducing the locking overhead for update transactions also, when record locking is in effect. Concurrency is also improved in conjunction with index concurrency control methods like ARIES/IM.[42] Many applications of the Commit_LSN technique are described in detail in Reference 45. Commit_LSN has been implemented in DB2 Version 3.

I/O versus CPU parallelism. Query processing in a parallel environment requires four major resource types: CPU, I/O, memory, and communication. Some form of parallelism is needed for large-scale use of any of these resources. Disk arrays[4] provide large amounts of storage as well as many read/write arms for higher bandwidth. (They may also improve availability by striping different bits of a byte on different devices and by storing some parity bits in a similar fashion.) Main memory subsystems with many ports and many memory modules provide similar features. Likewise, communication systems with switches at different levels and many ports provide high bandwidth. The degree of parallelism needed in each resource type (e.g., CPU) depends on the load on that resource type and the speed of a component of that resource type. As a result, different degrees of parallelism are needed for different resource types. Next, we discuss the relationship between parallelism of two major resource types in DBMSs: CPU and I/O.

Different degrees of parallelism are needed for different types.

Our objective is: Minimize the response time (up to a threshold), where the constraint is the amount of given resources. *Threshold* is defined as that response time below which minimization is not significant. In other words, we want to maximize use of the given limited resources to minimize the response time up to a threshold.[46] Different degrees of parallelism may satisfy this objective. Suppose we can fully utilize the CPU resource with 100 tasks or with 1000 tasks. One question is what the degree of parallelism should be. We argue that it is important to find the min-

imal degree of parallelism, while satisfying our objective. The higher the degree of parallelism, the harder the load balancing would be. By increasing the number of tasks across which work is being distributed, we are decreasing the number of records that each task handles. In other words, we have fragmented the processing and made it less *set oriented*, hence potentially compromising one of the major benefits that the relational model provides us. As a result, the processing may become less efficient. For example, we may lose the efficiency of sequential prefetch[47] because each task does not access enough pages to take full advantage of sequential prefetch in terms of amortizing the cost of an I/O call across a large number of pages.

Inefficiency can also arise in accessing data through nonclustered indices. In sequential processing, we extract the *RIDs* (or record identifiers) of qualified records from the index, sort the RIDs by page identification, and then perform the I/O.[40] Hence, each relevant data page is retrieved only once. If many tasks do this in parallel, often the same page may be retrieved many times, because, for a given page, more than one task may be interested in different records in it. Each task has a certain fixed cost associated with operations such as opening and closing scans, and sort initialization (e.g., initialization of the tournament trees when tournament sorts are used). This cost is multiplied by the degree of task parallelism. In addition to the wastage of CPU cycles, other resources like memory and channel capacity may also be wasted. Contention for disk arms and channels may also be increased.

The relationship between CPU and I/O parallelism raises a concern that often there is a significant mismatch between the degree of parallelism needed for CPU and that needed for the I/O subsystem. One reason for this is that the speed of I/O devices has not increased over time as fast as that of CPUs. To study the relationship between I/O and CPU parallelism, consider the problem of accessing the base tables directly or through indices. If all the data fit in main memory, then each task is CPU bound, and we need only one task per CPU. Hence, the degree of parallelism is the number of available CPUs. If data are on disks, the tasks can be I/O bound if one disk arm at a time is used. This causes a significant mismatch between the degrees of parallelism needed for CPU and I/O. The reason is that the speeds of the available disks are too low compared to the power of the currently available CPUs. Therefore, we need to have numerous disk arms, as in disk arrays, to keep up with each CPU.

Let us discuss an example. We assume that the processing capacity of each CPU is 30 MIPS. Consider two types of disks: (1) Slower disks with 3MBPS (megabytes per second) bandwidth, and 20 ms (millisecond) average seek plus search (i.e., rotational latency) time, and (2) Faster disks with higher bandwidth and moderately lower seek plus search time. Let us assume that these disks are an order of magnitude better in bandwidth (30MBPS) and half the order of magnitude better in average seek plus search time (7 ms).

Consider two types of queries: (1) Type 1 that are complex queries with numerous sequential table scans, and (2) Type 2 that are complex queries with numerous RID list data accesses, as previously explained (mostly doing random I/O).

The second type of query is chosen when the table is very big and the predicates are very selective. Hence, we may be heavily using even nonclustered indices (one index, or several, with index ANDing or ORing[40]). The queries of the first type mainly do sequential I/O. Hence, for each I/O, the seek/search cost is incurred once for a set of pages (e.g., 64 pages) and the limiting factor is mostly the data transfer bandwidth of the disk. The second type of queries mainly do random I/O, hence the seek/search time delay is usually incurred for every page. In this second case, the seek/search time is the limiting factor.

As the I/O speed increases, we need less parallelism in the I/O subsystem. There are two interesting cases: Case 1 is where the degrees of parallelism for CPU and I/O are close to each other, and Case 2 is where the degree of parallelism for I/O is much more than that for CPU (more than an order of magnitude for the Type 2 complex queries previously explained).

In Case 1, the system is not significantly CPU or I/O bound. Each task spends roughly equal time using CPU or I/O resources. Suppose each task does asynchronous disk page prefetch, where the task starts the I/O for the next set of pages at the time it starts working on the current set of pages. Under these conditions, each task becomes CPU bound, and it is sufficient to have as many tasks

as CPUs. I/O parallelism follows from (happens as a result of) CPU parallelism, and no special mechanism is needed for I/O parallelism.

For Case 2, we have two possibilities:

1. Use the same approach as in Case 1, where I/O parallelism follows from CPU parallelism. In this case, each task is now mostly I/O bound (even with I/O overlap). We need to increase the degree of CPU parallelism to that of I/O, hence allowing better utilization of resources, such as the CPU. The problem with this approach is that it artificially increases the CPU parallelism significantly (an order of magnitude in the previous Type 2 example). This may not be acceptable because as we argued before, we want to decrease the degree of parallelism in CPU as much as possible for better load balancing and reduction of overheads.
2. Decouple parallelism of CPU and I/O subsystems. Allow I/O to have more parallelism than CPU. This is the desired approach. An example of such an approach is the use of disk arrays where different blocks of data are scattered on different disks. Note that we mostly need this for random I/O, allowing different disk arms to work on different blocks of data. As explained before (and in References 40 and 48), a CPU task accesses the index and forms a list of pages to be retrieved. This list can be given to the I/O subsystem (via a START I/O instruction). Suppose these pages are stored in a disk array. The control unit of the disk array is responsible to initiate I/O (tasks) on different disk devices in parallel to retrieve the pages.

So far, we have discussed the three main architectures for parallelism: shared everything, shared disks, and shared nothing. Further, we showed that in addition to CPU, parallel use of I/O resources is important. In the following section, we concentrate on how to parallelize queries. As we will see, parallelization of queries and load balancing typically become harder as we move from the shared everything to shared disks to shared nothing architecture.

Parallel algorithms

In this section, we discuss the ways of parallelizing a query, load balancing issues, what impact parallelization has on different components of a relational DBMS, and how parallelism is enhanced with some relational operators.

Targets of parallelization. There are two ways of parallelizing complex queries: *Program parallelism* where the execution of multiple operations of a given program occurs in parallel, and *data parallelism* where the execution of a single operation occurs by operating on its input data (possibly, different pieces) in parallel.

Program parallelism (PP) and data parallelism (DP) are possible in all the three architectures, SD, SN and SE, defined earlier, and can be mixed. DP is the key to supporting a high degree of parallelism, whereas the degree of parallelism obtainable by PP is often much less than that of DP.

Program parallelism. Let us consider an example that involves joining the four tables T0, T1, T2, and T3. A possible execution strategy is one in which the join of T0 and T1 is performed in parallel with the join of T2 and T3. We call this style of execution *independent task execution*. Another possible execution strategy is one in which the join of T0 and T1 is performed by task S1, which then sends the result records incrementally to task S2 to perform the join with T2. S2 then sends its result records incrementally to S3 to do the join with T3. We say that this style of execution makes use of *asynchronous pipelines*. The reasoning behind the name has to do with the fact that the records are *piped* between tasks. But, unlike the synchronous pipelining used in sequential plans (e.g., as in System R[49]), here different stages of the pipeline are not executed in a lockstep fashion.

The queue between the producer and consumer tasks is called a *table queue* since its contents are records in (composite) tables. Obviously, some sort of flow control is needed between the producer(s) and consumer(s) of a table queue in order to reduce the overflow of the queue to disk, if the queue gets too large due to a slow consumer. This kind of asynchronous pipelining is also proposed in References 50 and 51, and is also useful in distributed DBMSs. It was also used in the R* prototype.[52] In the latter, the communication network protocols provided the pacing between the producer and the consumer.

An execution plan is a partially ordered set of operators.[53] Examples of operators are index or

data access and predicate evaluators, sort, join, aggregation, etc. The number of operators depends on the complexity of queries. Obviously, the degree of parallelism obtainable by PP is limited by the number of operators used in a query. In fact, the actual degree of parallelism attainable is usually much less than this upper bound due to the dependencies between operators. For example, the merge join of T1 and T2 cannot start until the access and sort of T1 and T2 are completed. In most of the cases, PP is not sufficient to provide a degree of parallelism in the 100s or 1000s. However, PP is more useful in conjunction with DP, which is discussed more later. The cost of intertask communication between operators in two different tasks is considerably higher than that between operators within the same task and, in fact, in systems like DB2, the records are not copied (in most of the cases) when they go through synchronous pipelines between operators. This cost is particularly high if tasks are in different processors that are not sharing memory. Analysis of queries in the context of a model based on projected path lengths of MVS (the IBM Multiple Virtual Storage operating system) and DB2 shows that the path length more than doubles if all synchronous pipelines are replaced by asynchronous pipelines.[51] The extra path lengths are mostly due to the costs of forming records, inserting them into and retrieving them from table queues.

Data parallelism. DP is the key to supporting a high degree of parallelism. Currently, a table may be divided up into a number of partitions (one such system, DB2, allows up to 64 partitions, for example). This is true even in a system that does not employ parallelism within a query (beyond doing sequential prefetching of data using system tasks in anticipation of future requests from the user's query processing task[47]). Each partition may be stored on a different device (possibly of a different type) and reorganized independently. DB2 partitioning is based on nonoverlapping key ranges, as specified by the creator of the table. In contrast, systems like TPF, Bubba, DBC/1012,[22] and Grace[54] use hashing to assign records to different partitions.

A hybrid approach is one that combines DP and PP. The extreme case of the hybrid approach is the one where we associate one task with each operator for each data partition. That is, we employ full DP and full PP and name this approach *parallel asynchronous pipelines*. This approach is undesirable from the viewpoint of the tremendous increase in path length that it would cause. Hence, if DP provides the desired parallelism, then use *synchronous* pipeline as much as possible for each partition and run different partitions in parallel. This scheme is named the *parallel synchronous pipeline* approach.

Pipelining helps reduce peaks in data communications and disk I/O. If pipelining is not used, in an SN architecture, the data from the producer are transferred across the network and put on the disk at the consumer's system. This may cause a peak in communication if the producer does not have much work to do (e.g., it is reading the local work files and distributing them across the network). But, if the data are piped to the consumer, then usually it is the consumer who is the bottleneck due to the processing (e.g., join) that needs to be performed on the incoming data, and also may be due to the lower priority assigned to it. As a result, the data transfer is spread over a longer period of time, thereby reducing the peak in the communication traffic. An asynchronous approach and a synchronous approach are presented in Reference 55 for controlling and managing query pipelines.

Load-balancing issues. As discussed in the previous section on parallelism, the key elements of parallelism are data and computation partitioning. Different methods of data partitioning (e.g., key range partitioning) and computation partitioning (e.g., program and data parallelism) were mentioned before. Computation partitioning must be done such that the load is spread as evenly as possible among the different tasks and different physical resources involved in the computation. Two kinds of load balancing are important: (1) physical resource level (e.g., load balancing of CPU nodes across many simultaneous applications), and (2) logical resource level (e.g., load balancing of different tasks accessing a table in parallel). Other discussions on load balancing can be found in Reference 56.

Physical load balancing. In this paper, we discuss only physical load balancing involving CPU nodes, not the load balancing of I/O and communication resources. In the partitioned architecture (SN), the data from a disk must be retrieved through the CPU node where the disk is attached. This node controls I/O and locking, applies the local predicates, extracts the qualified records, and sends

them to the next stage of computation, which may be in another node. A node may become overloaded if there is too much demand for the data under its control. To reduce the load on this node, we might consider the following alternatives:

1. Off-loading predicate evaluation and record extraction. This requires sending raw pages to the destination nodes. Locking is essentially at the page level because the source node cannot distinguish between qualified and unqualified records. Another alternative is for the destination node to do the locking, but this requires a global locking mechanism, as in the SD architecture. In a partitioned architecture (SN), usually all the locking is done locally. The effective communication bandwidth required may also increase considerably because the records are not filtered at the source. With this, the SN architecture comes closer to the SD architecture, where raw pages are shared among the nodes. But unlike in SD, no buffer coherency protocols are needed.
2. Data redistribution. We can redistribute the data to avoid the overloading of the node. This is possible if different pages, or different records within pages are demanded from the different nodes. Also, it requires *a priori* knowledge of the data usage pattern. Further, the usage pattern must not change too often (e.g., between day and night times). Note that the SD architecture can handle such pattern changes very well.
3. Orthogonal data distribution. In this approach, the correlation between distribution of data placement and data usage is minimized. An example of this is random data placement. This approach is the best for avoiding skews, however, it does not allow the clustering of data to minimize I/O and locking costs. This is a drawback particularly for handling of complex objects. Also, the overhead may be too much for small queries. (For example, if a request results in 15 records, then it probably will involve at least 15 tasks on 15 nodes in the SN architecture.)

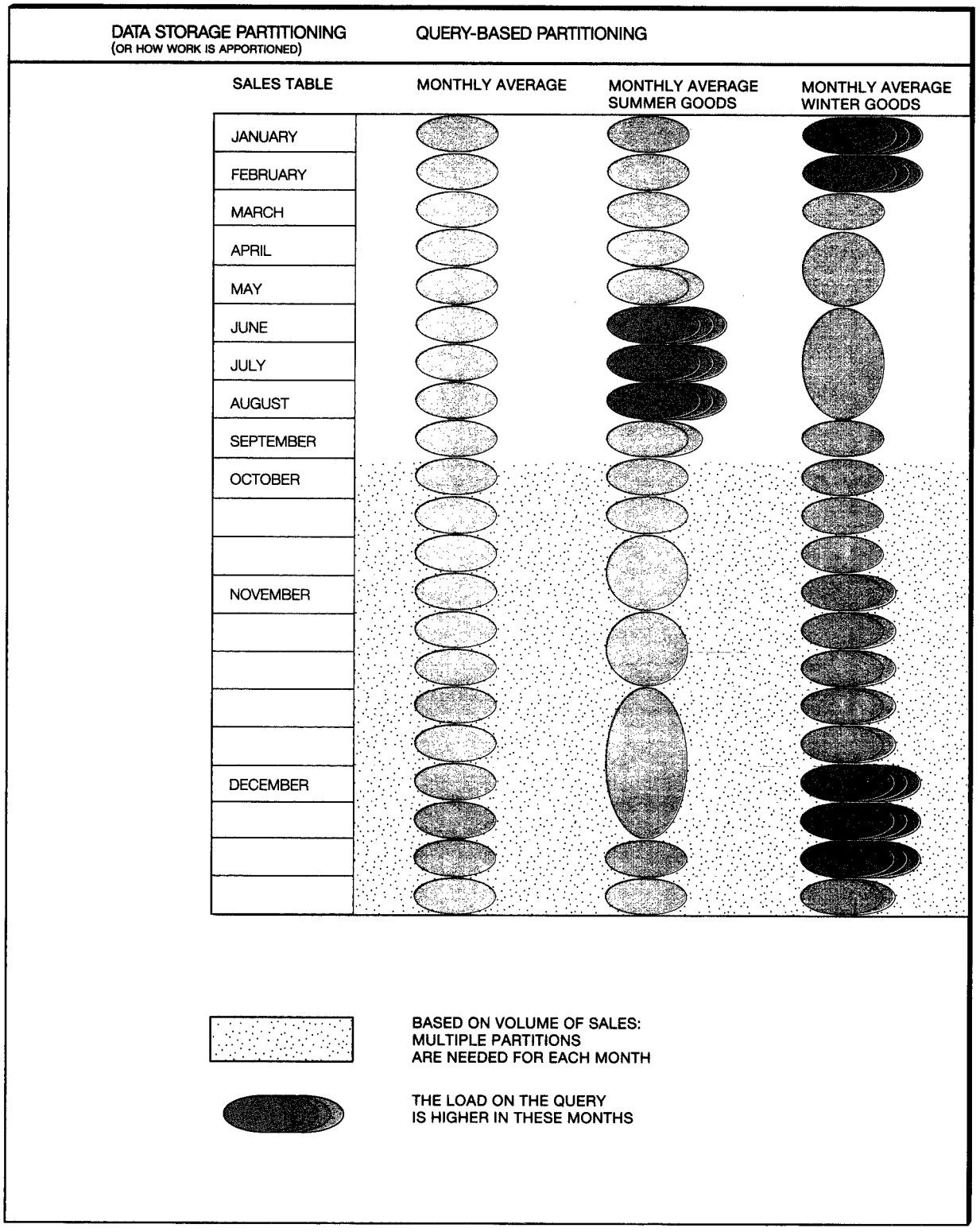
It is possible that the same set of records is demanded from different nodes. If the data are only read most of the time, then data replication can reduce contention. Otherwise, the data must be granularized more through schema changes, or new lock modes (such as increment/decrement locks) must be introduced to reduce contention. Also, the Commit_LSN technique[44,45] may be used to reduce contention and to avoid a significant amount of locking overhead.

Logical load balancing. With data parallelism, each operator is assigned a portion of the work. One way of partitioning the work is to assign a task to each physical data partition. For example, in Figure 2 the Sales Table is partitioned into data for sales in January, February, etc. Since the volume of sales records is higher during the months of October, November, and December, more partitions are assigned to these months. The query for computing average sales for each kind of item sold per month can easily be computed by running one task per partition to aggregate the data of each partition and consolidate the aggregates at the end. This load is balanced across different tasks assuming uniform sizes for the partitions. We call this physical partitioning.

Physical and logical resource level load balancing are important.

Now consider another example where the query is for computing the monthly average for the summer goods. For this, the load of the query will be higher for the physical partitions associated with the summer months. Therefore we need to run more tasks for the summer partitions and fewer tasks for the winter partitions to balance the load of tasks. As shown in Figure 2, suppose we need to run four tasks for the month of June. To achieve this, we may logically partition this physical partition into four logical partitions, each containing roughly one week of data and running a task for each logical partition. Likewise, consider another example where the query is for computing the monthly average for the winter goods. For this, the load of the query will be higher for the physical partitions associated with the winter months. Therefore we need to run more tasks for the winter partitions and fewer tasks for the sum-

Figure 2 Example of logical partitioning



mer partitions to balance the load of tasks. This example shows that physical partitioning is not sufficient to achieve load balancing. Physical partitioning of data helps load balancing, but load balancing must be done for each query. In an extreme case, the query might be only for the month of September, which has only one partition. For this, logical partitioning is a must to achieve parallelism.

Although the above example might apply to the Access and the Group By types of operators, load balancing must be done for all the operators, such as join and sort. References 57 and 58 address the load balancing issues for the sort operator.

Impact of parallelism on DBMS components. In this section we give an overview of different components of a typical DBMS, then explain the features of these components that are important for parallelism. For ease of exposition, we present this in terms of enhancements needed to parallelize a sequential DBMS.

We call the upper part of the system that deals with query optimization and plan generation the relational data system (RDS). The lower part of the system that deals with buffer management—concurrency control, recovery, record management, and space management—is called the data manager (DM). We discuss the enhancements that need to be done at the RDS and DM levels. These levels correspond to the System R RDS and Research Storage System (RSS) levels.[59]

Enhancements to the SQL language are necessary to make more parallelism possible within the DBMS. Although the 1992 (SQL2 SQL) standard allows an insert statement to provide multiple records, most implementations of SQL allow the application to provide only one record at a time to the DBMS for insertion. As a result, there is little chance for parallelism for such insert operations.[60] The same is true for the SQL statements *update* or *delete where current of cursor*. Only one record at a time can be updated or deleted in this case. We must allow the application to specify update or delete for a set of records in one SQL command.[61] Enhancements to the application program interface are also required to interface to DBMS applications that have many parallel pieces.

The optimizer may be designed to deal with the questions relating to the degree of parallelism and the assignment of work to the different tasks solely at the time of query compilation. This would be what we call *compile-time* or *static optimization*. Another possible approach is where, in addition to doing the compile-time determination of the number of tasks, enough intelligence is built into the run-time support and the plans themselves to dynamically adapt the execution plan. This would be based on information about the loads on the different processors, characteristics (size, data distribution, etc.) of the intermediate results, the availability of memory, etc. Typically, this *dynamic optimization* is more difficult to accomplish than the static optimization approach.

Complexity of optimization is already a major problem in relational DBMSs, and parallelism makes this problem even bigger. One question is whether there is a compromise approach to optimization that does not increase complexity too much. One idea is a two-phase static optimizer. In the first phase, it optimizes the query, ignoring parallelism (i.e., acts as if the query will be run in a sequential DBMS). In the second phase, it takes the query plan chosen in the first phase and optimizes it further for parallelism. This is the approach chosen in XPRS,[29] mentioned earlier, and DB2 Version 3 (discussed in a later section). This approach is particularly attractive if we want to parallelize an existing DBMS. The two-phase optimization approach also reduces the search space of optimization, hence reducing optimization time.

Obviously, there will be situations for which this approach does not produce an optimal plan due to the fragmentation of optimization. Let us consider an example. Suppose the first phase chooses plan alternative P1 and rejects plan alternative P2. If P2 is much more expensive than P1, then usually we are not interested in the parallelized version of P2 either. The reason is that it wastes a significant amount of resource and such wastage is not acceptable in a multiuser environment. Furthermore, there is a good chance that its response time will be worse than that of P1. If the resource consumption of P1 is not very different and the parallelized version of P2 is better than the parallelized version of P1 then this original choice of P1 over P2 was a bad choice.

Let us assume that in the previous example P1 uses the nested loop and P2 uses the sort-merge

join methods; suppose the sequential optimizer chooses P1 since it has a lower cost, but not by a wide margin. In the SN architecture, there is a major difference in performance of nested loop versus sort merge. This problem arises when the corresponding partitions of the outer and inner tables are not co-located. Then, for each record of the outer, we have to visit other nodes to get the matching inner records. If the system has N nodes and each node accesses the records of the outer, we need N tasks for this. Each of these tasks has to execute N-1 tasks in N-1 other nodes to get the matching inner records. Therefore, the number of tasks is in the order of N^2, and the computation is fragmented in the order of N^2. The sort-merge alternative (P2) can run this query with N tasks for the outer, N tasks for the inner, and N tasks for the join. Therefore, it is in the order of N. N^2 number of tasks is not acceptable, particularly when N is large (for $N = 40$, number of tasks is more than 1000). Also, the efficiency of computation may be drastically reduced when it is fragmented N^2 times. Therefore, the sort-merge alternative (P2) is preferable, and the two-phase optimization will miss this better alternative. Such situations typically arise in the SN architecture.

Parallelism is enhanced in some of the key relational operations.

As argued above, the parallel optimization of a query depends on how much resource is available. This is usually known at run time. However, one source of complexity of dynamic optimization is the need for the modification of the plan at run time, particularly during the execution of the plan. One way to avoid this is for the static optimizer to optimize the query based on the maximum amount of available resources. Then, at run time, the degree of parallelism of each part of the plan can be changed based on the actual amount of available resource. This usually does not require complex plan changes at run time. This is the approach chosen for DB2 Version 3, as we will see later.

Another change that would be desirable to the RDS component is the consideration of bushy joins (i.e., composite inner tables) in addition to the System R approach of considering only non-composite inners. Otherwise, we would be restricted to pipelining as the only means of getting program parallelism among the join operations. Join pipelining is feasible only if no sort needs to be performed on the result of one join before the next join can be performed.

The run-time component of a parallel DBMS must support starting and stopping of tasks, monitoring their progress, and communicating run-time errors. A paper on Gamma[27] discusses such support. In a multiuser environment, only a portion of the resources is allocated to a given query. Parallel queries have the potential of using the entire system resources, such as CPU, I/O, etc. Therefore, there is a need for a run-time mechanism to limit resource usage rate of queries to the assigned values. This usually requires cooperation with the (operating) system resource manager.

Enhancements may be necessary at the DM level for the handling of large buffer sizes, and possibly multiple buffer pools,[47] I/O parallelism, task structure, and locking support for parallelized update queries.

Parallelism and relational operators. Parallelism is enhanced if we (1) reduce dependencies between operations (see the discussion that follows about the join operation), and (2) make even the lower level DBMS functions more set-oriented (e.g., by performing aggregation during sort). In this section we discuss the functions that must be considered in parallelization of some of the key relational operations.

Access. Access refers to accessing permanent tables or work files. This operation also includes acquiring any locks, applying the eligible predicates and performing projections. An access may be via indices or by table scans. When access via index is chosen, more than one index may be used for accessing the records of a single table.[40]

Join. The *join* operation involves taking the results of accessing two or more tables and applying

the join predicates. If parallelism among accesses and the join is being maximized, then, during the access of the inner table, we cannot take advantage of those predicates on the inner table that involve columns of the outer table. The effect is as if the nested loop join method is not being used and the accesses are more like those performed in the sort-merge join method.

Group By. Ideally, aggregation should be combined with the sort and merge phases of the SQL *Group By* operation, instead of being performed as a separate operation after the merge is completed and we have a completely sorted stream. Combining aggregation with other operations will cut down the number of passes through the data by one. More importantly, in most cases, it will reduce the number of records to be dealt with in the merge phases, thereby potentially reducing the CPU and I/O overheads.

Duplicate elimination. Duplicate elimination also should be combined with sort and merge operations. Of course, if the columns of interest in the result constitute the key of a nonunique index, then it would be highly preferable to make the index manager itself return only nonduplicate values. Under these conditions, retrieving all the keys and then eliminating duplicates in the index manager's caller would be much more expensive, since indices typically store duplicate keys in a compressed form that can drastically reduce the number of comparisons required to select only one instance of each duplicate key. It may also reduce the number of locks that are acquired, depending on what the objects of locking are (see References 41, 42, and 45 for more discussions). This will be especially beneficial in the SD environment where the locks are global locks. While the above points are applicable even without parallelism coming into the picture, they become extremely important in the context of a database machine, since adopting them could lead to a drastic reduction in communication traffic also.

Selection. In the case of the set-oriented insert, delete, and update operations, the selection of the records can go on in parallel with the insert, delete, or update operation. Depending on the consistency level used (repeatable read or cursor stability) during the retrievals, by the time the delete or update operation is executed, the records that previously qualified may no longer qualify for the delete or update due to the activities of other transactions. This has to be dealt with carefully to avoid inconsistencies. This kind of problem arises even when there is no intratransaction parallelism, but the data access is postponed until the index accesses are finished and cursor stability is used during the index access (see References 40, 45, and 62 for more discussions).

Union. The *union* operation will not have much work to do, unless (1) duplicates are to be eliminated, or (2) an Order By clause exists. If duplicate elimination is required, then, unless multiquery optimization is going to be performed and somehow the queries constituting the operands of the union are going to be combined, the elimination of the postprocessing required to remove duplicates would not be possible in most cases.

I/O parallelism in DB2 Version 3

DB2 Version 3 (DB2 V3) provides support for exploiting I/O parallelism (IOP) to reduce response time for data-intensive, I/O-bound queries. For a table with large amounts of data, DB2 users may partition the table (*physical partitioning*) onto separate I/O devices.

In prior releases, DB2 always read each table in a single stream manner, one partition after another. Therefore, a large percentage of the query response time might have been spent on waiting for I/O operations to complete. DB2 V3 provides the capability to fully utilize the I/O bandwidth that is made possible by the use of a partitioned table. DB2 will initiate multiple concurrent I/O requests for a single-user query and perform parallel I/O processing on multiple data partitions. The query elapsed time can thus be significantly reduced for I/O-bound queries.

If a table is created as a partitioned table, DB2 can exploit the possibility of IOP when this table needs access. When a table is not created as a partitioned table, DB2 may consider partitioning such a table into multiple work files to benefit from IOP in subsequent operations. For example, when a table needs to be sorted for performing a sort-merge join, DB2 may consider partitioning the sort output into multiple work files and then executing the join in parallel.

DB2 uses IOP for both dynamic and static SQL queries. In the distributed context, it is used for both local and remote queries (with a DB2 requestor

and a DB2 server). Users have the ability to tell DB2 whether or not it should consider using IOP. Even if the user asks for IOP, it will be used only if the optimizer determines that it is likely to be useful. Using the EXPLAIN statement, a user can find out whether DB2 has planned to exploit IOP for a given query.

IOP is designed to support read-only I/O-bound queries. It can be exploited for an embedded SELECT, a read-only cursor, or a subquery since their results are read only and never updated. I/O-bound queries are those queries whose response time is dominated by the time for completing I/O operations. For a given query, when the user has requested IOP, if DB2 estimates that its I/O processing time is much higher than its CPU processing time, then it will choose to activate multiple parallel I/O streams. IOP can be applied to both single-table accesses and multiple-table joins. Typical types of queries that can take advantage of IOP include:

- Queries with intensive data scans and high selectivity: These queries involve large volumes of data to be scanned, but relatively few rows meet the search criteria and are returned.
- Queries containing aggregate functions: Similar to the first type of queries, column functions (MIN, MAX, SUM, AVG, COUNT) usually involve large amounts of data to be scanned, but few aggregate results are returned.
- Queries accessing long data rows: These queries access tables with long data rows, and the ratio of rows per page is very low (e.g., each page contains only one row).

For these types of queries, DB2 spends most of the time fetching pages (high I/O cost), and relatively little time processing rows (low CPU cost). Therefore, IOP can be used to reduce the I/O elapsed time. When a query is I/O bound, the I/O bandwidth determines the query elapsed time. Query elapsed time decreases when I/O bandwidth increases.

The fundamental strategy of IOP is to process portions of a single user query in parallel through the use of horizontal data partitioning. DB2 will attempt to fully utilize the I/O bandwidth of the partitioned table by triggering parallel sequential prefetches on multiple partitions. The number of parallel I/O streams activated in a query is called the *degree of parallelism.* One or more consecutive operations (each operation can be a table access, a join, or a sort) can be executed in parallel as a group, such that all the operations in this group have the same degree of parallelism. Such a group is referred to as a *parallel group.*

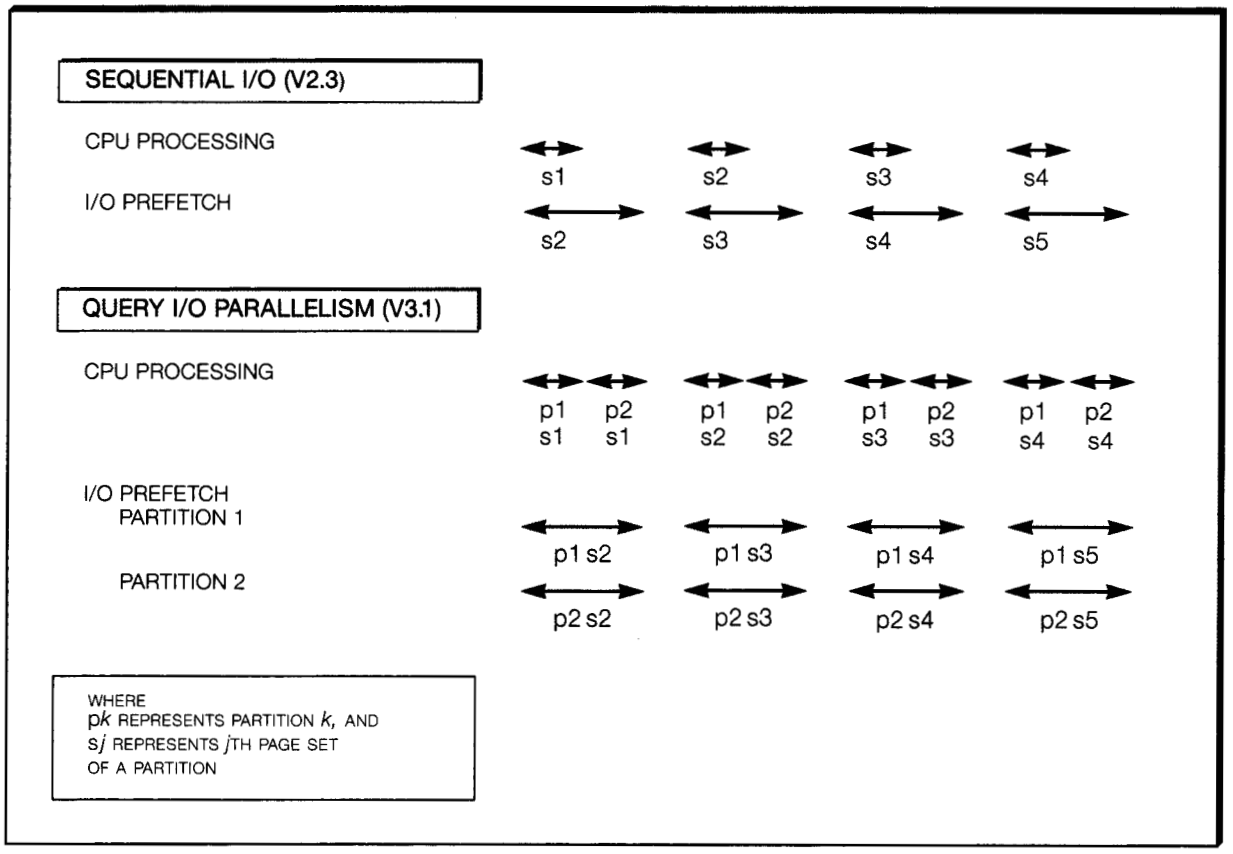
The sequential prefetch support that has existed from DB2 Version 1, Release 1[47] is essential to the implementation of IOP. Sequential prefetch is a mechanism to trigger asynchronous I/O operations. Pages are fetched into the buffer pool before they are required, and several consecutive pages (e.g., 32 pages) are read with a single I/O operation. This technique allows CPU and I/O processing to overlap, and reduces the query elapsed time.

Figure 3 illustrates sequential prefetch. When a CPU is processing a set of pages fetched into the buffer pool (e.g., set s1), the I/O subsystem is fetching the next group of pages (e.g., set s2). Therefore, the CPU and I/O processing are overlapped, but the I/O processing itself is still in sequential mode in DB2 Version 2. As processors become faster, more queries become I/O bound. The figure shows a case where the CPU speed is faster than that of I/O, which results in the CPU waiting for I/O operations to complete. Query performance can be improved further by overlapping the I/O operations with one another. With IOP, multiple instances of the same operation are executed on many disjoint partition key ranges or page ranges.

Figure 3 also shows an example of parallel I/O processing on a table with two partitions. It assumes that the CPU processing speed is two times faster than the I/O processing speed. DB2 invokes two parallel I/O streams to fetch data from the two partitions concurrently. Except during the initial transitive stage, the CPU is kept busy all the time processing data, while sequential prefetches on different partitions continue to be triggered in a round robin fashion until the query is complete. During the initial transitive stage, CPU processing will be delayed while waiting for the first few sequential prefetches to be activated for all the partitions.

The placement of files on the disk is critical to the performance of IOP. To maximize the performance improvement, I/O contention should be kept to a minimal amount by spreading the partitions over different disks, and allowing each I/O

Figure 3 Illustration of I/O parallelism



stream to operate on a separate I/O channel (bus) path. The DB2 V3 IOP approach is extensible to any possible future support for CPU parallelism within a machine and even across machines in a shared disks environment.[8]

Planning for parallelism. The DB2 V3 optimizer is used to group similar degrees of parallelism.

Parallel groups. A parallel plan is produced by the optimizer based on the best sequential plan by parallelizing its I/O bottlenecks, thereby reducing its elapsed time. In DB2 V2, the optimizer produces a sequential query plan that consists of a chain of operations. Each operation can be a table access, a join, or a sort, and is represented by a *miniplan* data structure. To support IOP in DB2 V3, the pre-existing optimizer has been enhanced with a *postoptimization* phase that examines the best sequential plan produced by the pre-existing logic and determines which operations are I/O bound and can benefit from IOP. Some operations with the same degree of parallelism are grouped together. Each of those groups is called a *parallel group*.

The general rule-of-thumb to determine the groupings of parallelism is: A parallel group starts with an access and continues to cascade until there is a sort operation. Since CPU parallelism is not being exploited in DB2 V3, a sort operation will have to be performed sequentially on a single processor. Therefore, a sort operation stops a parallel group in the miniplan chain. There can potentially be two different parallel groups that a table can be in: one for accessing that table and another for joining that table with a composite table. This usually means that the table is sorted with parallel table access and then partitioned for the next parallel join.

Degree of parallelism. The degree of parallelism is the number of parallel I/O streams activated for a table.[63] The optimizer estimates the degree of parallelism for each parallel group (at compile time or at run time). However, since some important factors are not known until run time (such as amount of buffer pool resources available and values of host variables), the actual degree of parallelism may be adjusted at run time. The run-time logic goes through a process called *buffer pool negotiation* to determine whether enough buffers are available to stay with the degree of parallelism determined by the optimizer at compile time. As a result of (automated) negotiation, a parallel plan may even fall back to the original sequential plan at run time. To be able to do the run-time adjustment, the compiled plan is augmented to include partitioning information for all the referenced tables.

The goal of IOP is to reduce elapsed time with optimal resource utilization. Therefore, the degree of parallelism can be determined in two steps. The first step is to estimate the best possible elapsed time for a given parallel group in the query. To do this, we pretend that every physical partition is going to be accessed by a separate I/O stream. The second step is to find the minimum degree of parallelism that can achieve the best possible elapsed time (see the earlier section on I/O versus CPU parallelism for further discussion).

Step 1: Estimate the best possible elapsed time.

```
Elapsed time = max (CPU processing time,
                    I/O elapsed time)
I/O elapsed time = maximum partition I/O time
```

Yields:

```
Elapsed time = max (CPU processing time,
                    maximum partition I/O time)
```

Thus:

- The total elapsed time is the maximum of CPU processing time and I/O elapsed time, assuming CPU processing and I/O are totally overlapped with each other.
- The CPU processing time is equivalent to the CPU resource utilization time estimated for sequential plan, assuming the CPU overhead for activating parallelism is negligible.
- The I/O elapsed time is determined by the maximum partition I/O time (i.e., the partition that takes the longest time to finish its I/O). This is due to the fact that, with IOP, all the partitions of all the tables are read in parallel. Therefore, the slowest partition becomes the bottleneck. Having multiple I/O streams accessing a single partition cannot reduce the elapsed time of partition I/O time since those multiple I/O streams have to share the same physical device.
- By substituting CPU processing time with sequential CPU time and I/O elapsed time with maximum partition I/O time, we obtain a formula for the best possible elapsed time. It is the maximum of the sequential CPU time and the largest partition I/O time.

Step 2: Find the *minimum* degree of parallelism needed to achieve the elapsed time estimated in Step 1.

A table can be accessed in parallel either through key partitioning (for an index scan) or page partitioning (for a table scan). To determine the degree of parallelism, we derive appropriate key ranges or page ranges to achieve the best possible elapsed time based on the following assumptions:

- Data are uniformly distributed within a partition.
- I/O contention can be made negligible when multiple I/O streams access a single physical partition by making those multiple streams access that partition at different times.

The key ranges or page ranges being derived are called *logical partitions*. Each logical partition is accessed by an I/O stream. Logical partitions do not necessarily correspond to physical partitions and may not fall into physical partition boundaries. The degree of parallelism can be found after all the logical partitions are derived. It is important that when more than one table is present in a parallel group, the corresponding logical partitions of the different tables have matching key ranges. This is necessary to make sure that a set of records of one table is read into the buffer pool around the same time the corresponding joining records of the other tables are also read in.

Impact of host variables on degree of parallelism. The optimizer determines the degree of parallelism at compilation time if there are no host variables in the query that may influence the degree of parallelism. If a query references a host vari-

able that involves the partitioning key, which may influence the I/O time of the query, then the ideal degree of parallelism cannot be determined at compile time. When host variables are present, the compiled plan has been augmented to include the necessary CPU and I/O time information so that the run-time logic can evaluate the values of host variables, adjust the I/O time of the query, and determine the actual degree of parallelism. The run-time logic does the latter by invoking a procedure in the optimizer component.

The values of host variables can be used to determine which part of a table is qualified to be accessed. The partition I/O time can be adjusted accordingly.

Buffer manager extensions. A few extensions to the buffer manager were needed to support IOP. Those extensions are described in this section.

Conditional page fixing. Until DB2 V3, whenever a fix_page request was invoked, buffer manager (BM) tried to find the requested page in the buffer pool. If BM could not find the page, it allocated a buffer slot for the page and scheduled a read I/O operation. BM suspended the invoker unconditionally whenever a read I/O was in progress. The read I/O could be triggered either by the invoker or by an asynchronous prefetch task. With IOP, since a query process could be initiating many concurrent I/O streams, it was beneficial for the process to switch to a different I/O stream if the current I/O stream ran into any wait state.

In order to allow the process to switch to a different I/O stream, BM in DB2 V3 provides a conditional fix_page request. When this option is used, the BM invoker is not forced to wait for any read I/O to be completed for the requested page, but an asynchronous I/O will be initiated on behalf of the requester. In the absence of CPU parallelism, since a single process has to process multiple I/O streams, such processing is made possible by making the process go as far as it can with processing the pages of a particular I/O stream that are in the buffer pool before switching to pages from another I/O stream. Anytime the process decides to process a particular stream, the process issues an *unconditional* fix_page for the next page to be processed (i.e., the page for which a *conditional* fix_page performed during the previous round of processing of this stream resulted in a *page not found in buffer pool* response from BM). For any subsequent pages of that stream, *conditional* fix_pages are issued during the current round.

Buffer pool availability. Each parallel stream requires at least 16 pages, and up to a maximum of 64 pages from the buffer pool for sequential prefetch. (One study has shown that each I/O stream should maintain at least 16 buffer pages for sequential prefetch in order to obtain a reasonable reduction in response time with IOP.) The prefetch quantity can be dynamically adjusted by DB2 based on the system-wide buffer pool usage. If the sequential prefetch quantity is Q pages, then the number of buffer pages required per I/O stream is 2Q pages (Q for pages currently being processed and Q for the next set of pages being prefetched).

At run time, DB2 will look at the number of buffer pages available and determine the maximum number of parallel I/O streams that it can support without jeopardizing the performance of other queries running in the system. To derive the actual degree of parallelism at run time for a parallel group, DB2 divides this maximum number by the number of tables that need to be accessed by this group. This new degree may be lower than the planned degree if the current buffer pool usage is high.

Performance monitoring and tuning. New trace and statistics records are produced by DB2 V3 to help in performance monitoring and tuning. They provide the following information:

- Description of how the tables within a parallel group are partitioned by specifying the key range and page range for each partition
- Elapsed time statistics for each parallel operation
- Number of times the buffer manager was unable to allocate the desired number of buffers to support the planned degree of parallelism for a parallel group
- Number of times the sequential prefetch quantity had to be reduced in order to allow multiple queries to continue to execute concurrently with IOP
- How often the page had already been prefetched into the buffer pool, when DB2 needed a page during the execution of a parallel query

The system administrator is allowed to specify what percentage of the buffer pages can be used to support prefetching for IOP.

Possible extensions. The following are some areas that will potentially improve performance, if implemented.

- Nonuniform distribution of partitioning key values—The current implementation assumes a uniform distribution of partitioning key values. As a matter of fact, for some time, DB2 has been supporting the efficient handling of those situations where there is nonuniform distribution of key values by tracking frequently occurring key values.[64] But this latter information can be exploited further to do a better job of logical partitioning.
- Better modeling of CPU availability—To encourage IOP, DB2 assumes that the CPU is 100 percent available to the query in order to obtain the best estimates for the I/O and CPU times. This CPU available percentage may need to be adjusted to reflect the situation where CPU utilization is high when the query is executed.
- Parallel I/Os for write operations—SQL insert, delete, and update statements also can benefit from IOP. This can be done by parallelizing at least the retrieval portions of those statements (e.g., the evaluation of the subquery in the case of an insert from subquery statements). A further step would be to update any indexes also in parallel.

Conclusions

In this paper we discuss some architectural alternatives and design approaches for introducing intraquery parallelism in a relational DBMS. We discuss the pros and cons of the shared nothing (SN) and shared disks (SD) architectures. While scalability might be a problem for SD, it has many advantages with respect to load balancing and database design. Further research is needed to clarify these points. A possibility is using the SD architecture in the nodes of an SN system. This gives us more powerful nodes that are easier to manage and whose load is easier to balance. Availability in case of failures of some processors within a node is also enhanced with this hybrid approach. We also discuss the increasingly important role that disk arrays will play in improving the performance of the I/O subsystem to match the latter with that of the CPUs. In addition, we discuss asynchronous pipelining using table queues and the overheads that they impose compared to synchronous pipelining. Parallel synchronous pipelining is pointed out as the preferred method of accomplishing parallelism, whenever possible. With respect to load balancing, we discuss some of the major issues. We may not foresee some skew problems at compile time due to the absence of knowledge about the values of bindings of host variables, correlations, etc. This requires doing some work at run time. This is a major research problem.

Other major topics that must be considered in the study of a parallel DBMS include system management, utilities (database reload, unload, reorganization, etc.), performance of transaction workload on a large number of small CPUs, and the mix of transaction and query workloads. We are continuing work on the research topics that we have identified in this paper.

We also present in detail the implementation of I/O parallelism in DB2 V3.

Acknowledgments

We would like to acknowledge the contributions of our IBM colleagues in Almaden, Santa Teresa, Toronto, and at the T. J. Watson Research Center in Yorktown Heights, New York. In particular, we wish to thank Annie Tsang and Allen Yang for their contributions to some sections of this paper.

Cited references and notes

1. C. Mohan, "A Survey of DBMS Research Issues in Supporting Very Large Tables," *Proceedings of the Fourth International Conference on Foundations of Data Organization and Algorithms*, Chicago, IL (October 1993).
2. R. Ananthanarayanan, V. Gottemukkala, W. Kaefer, T. J. Lehman, and H. Pirahesh, "SMRC: Incorporating Object-Orientation into Starburst," *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Washington, DC (May 1993).
3. H. Hsiao and D. DeWitt, "Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines," *Proceedings of the Sixth International Conference on Data Engineering*, Los Angeles, CA; IEEE Computer Society, Washington, DC (February 1990).
4. D. Patterson, G. Gibson, and R. Katz, "A Case for Re-

dundant Arrays of Inexpensive Disks (RAID)," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Chicago, IL; Association for Computing Machinery, New York (May 1988).
5. S. Hobson, "ALCS—A High-Performance, High-Availability DB/DC Monitor," *Lecture Notes in Computer Science* **359**, D. Gawlick, M. Haynie, A. Reuter, Editors, Springer-Verlag (1989).
6. A. Bhide, "An Analysis of Three Transaction Processing Architectures," *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, Los Angeles, CA; Morgan Kaufmann Publishers, Incorporated, Palo Alto, CA (August 1988).
7. D. Dias, B. Iyer, J. Robinson, and P. Yu, "Integrated Concurrency-Coherency Controls for Multisystem Data Sharing," *IEEE Transactions on Software Engineering* **15**, No. 4 (April 1989).
8. C. Mohan and I. Narang, "Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment," *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, Barcelona; Morgan Kaufmann Publishers, Incorporated, San Mateo, CA (September 1991). A longer version is available as Research Report RJ-8017, IBM Almaden Research Center, San Jose, CA 95120 (March 1991).
9. C. Mohan and I. Narang, "Efficient Locking and Caching of Data in the Multisystem Shared Disks Transaction Environment," *Proceedings of the Third International Conference on Extending Database Technology*, Vienna (March 1992). Also available as Research Report RJ-8301, IBM Almaden Research Center, San Jose, CA 95120 (August 1991).
10. C. Mohan and I. Narang, "Data Base Recovery in Shared Disks and Client-Server Architectures," *Proceedings of the Twelfth International Conference on Distributed Computing Systems*, Yokohama, Japan; IEEE Computer Society Press, Los Alamitos, CA (June 1992).
11. T. K. Rengarajan, P. Spiro, and W. Wright, "High Availability Mechanisms of VAX DBMS Software," *Digital Technical Journal*, No. 8 (February 1989).
12. K. Shoens, "Data Sharing vs. Partitioning for Capacity and Availability," *Database Engineering* **9**, No. 1, (March 1986).
13. H. Boral, "Parallelism and Data Management," *Proceedings of the Third International Conference on Data and Knowledge Bases, Improving Usability and Responsiveness*, Jerusalem; Morgan Kaufmann Publishers, Incorporated, San Mateo, CA (June 1988).
14. M. Stonebraker, "The Case for Shared Nothing," *IEEE Database Engineering* **9**, No. 1 (1986).
15. C. Carr, R. L. Huddleston, J. Strickland, *Method and Means for the Retention of Locks Across System, Subsystem, and Communication Failures in a Multiprocessing, Multiprogramming, Shared Data Environment*, U.S. Patent No. 4,480,304 (1985).
16. J. P. Strickland, P. P. Uhrowczik, and V. L. Watts, "IMS/VS: An Evolving System," *IBM Systems Journal* **21**, No. 4, 490–510 (1982).
17. C. Mohan, "IBM's Relational DBMS Products: Features and Technologies," *Proceedings of the 1993 ACM SIGMOD International Conference on the Management of Data*, Washington, DC; Association for Computing Machinery, New York (May 1993).
18. C. Mohan, K. Britton, A. Citron, and G. Samaras, "Generalized Presumed Abort: Marrying Presumed Abort and SNA's LU 6.2 Commit Protocols," *Proceedings of the Fifth International Workshop on High Performance Transaction Systems*, Asilomar (September 1993). Also available as Research Report RJ-8684, IBM Almaden Research Center, San Jose, CA 95120 (March 1992).
19. C. Mohan, B. Lindsay, and R. Obermarck, "Transaction Management in the R* Distributed Data Base Management System," *ACM Transactions on Database Systems* **11**, No. 4 (December 1986).
20. The Tandem Database Group, "NonStop SQL: A Distributed, High-Performance, High-Availability Implementation of SQL," *Lecture Notes in Computer Science* **359**, D. Gawlick, M. Haynie, A. Reuter, Editors, Springer-Verlag (1989).
21. H. Zeller, "Parallel Query Execution in NonStop SQL," *Proceedings of IEEE Compcon Spring '90* (March 1990).
22. F. Carino and P. Kostamaa, "Exegesis of DBC/1012 and P-90—Industrial Supercomputer Database Machines," *Proceedings of the Fourth International PARLE Conference*, Paris; Springer-Verlag (June 1992).
23. W. Alexander and G. Copeland, "Process and Dataflow Control in Distributed Data-Intensive Systems," *Proceedings of the 1988 ACM SIGMOD International Conference on the Management of Data*, Chicago, IL; Association for Computing Machinery, New York (May 1988).
24. H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez, "Prototyping Bubba, a Highly Parallel Database System," *IEEE Transactions on Knowledge and Data Engineering* **2**, No. 1 (March 1990).
25. H. Boral, "Parallelism in Bubba," *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, Austin, TX; IEEE Computer Society, Washington, DC (December 1988).
26. R. A. Lorie, J.-J. Daudenarde, J. W. Stamos, and H. C. Young, "Exploiting Database Parallelism in a Message-Passing Multiprocessor," *IBM Journal of Research and Development* **35**, No. 5/6, 681–696 (September/November 1991).
27. D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen, "The Gamma Database Machine Project," *IEEE Transactions on Knowledge and Data Engineering* **2**, No. 1 (March 1990).
28. M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout, "The Design of XPRS," *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, Los Angeles, CA; Morgan Kaufmann Publishers, Incorporated, Palo Alto, CA (August-September 1988).
29. M. Stonebraker, P. Aoki, and M. Seltzer, *Parallelism in XPRS*, Memorandum No. UCB/ERL M89/16, University of California, Berkeley, CA (February 1989).
30. D. J. Haderle and R. D. Jackson, "IBM Database 2 Overview," *IBM Systems Journal* **23**, No. 2, 112–125 (1984).
31. T. Haerder, H. Schoning, and A. Sikeler, "Evaluation of Hardware Architectures for Parallel Execution of Complex Database Operations," *Proceedings of the Third Annual Parallel Processing Symposium*, Fullerton, CA (1989), pp. 564–578.
32. W. Duquaine, "LU6.2 as a Network Standard for Transaction Processing," *Lecture Notes in Computer Science* **359**, D. Gawlick, M. Haynie, A. Reuter, Editors, Springer-Verlag (1989).
33. R. J. Sundstrom, J. B. Staton III, G. D. Schultz, M. L.

Hess, G. A. Deaton, Jr., L. J. Cole, and R. M. Amy, "SNA: Current Requirements and Direction," *IBM Systems Journal* **26**, No. 1, 13–36 (1987).
34. W. C. McGee, "The Information Management System IMS/VS, Part IV: Data Communication Facilities," *IBM Systems Journal* **16**, No. 2, 136–147 (1977).
35. T. Storey and J. Knutson, "CICS/6000 Online Transaction Processing with AIX," *AIXpert* (November 1992).
36. P. Bernstein, "Transaction Processing Monitors," *Communications of the ACM* **33**, No. 11 (November 1990).
37. F. Ross, "An Overview of FDDI: The Fiber Distributed Data Interface," *IEEE Transactions on Selected Areas in Communications* **7**, No. 7 (September 1989).
38. It should be mentioned that, for large databases, even if only one copy of the data is stored, the total cost of the disks used for storing the database is a major portion of the cost of the complete system configuration.
39. C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," *ACM Transactions on Database Systems* **17**, No. 1 (March 1992).
40. C. Mohan, D. Haderle, Y. Wang, and J. Cheng, "Single Table Access Using Multiple Indexes: Optimization, Execution, and Concurrency Control Techniques," *Proceedings of the Second International Conference on Extending Data Base Technology*, Venice; Springer-Verlag (March 1990). A longer version of this paper is available as Research Report RJ-7341, IBM Almaden Research Center, San Jose, CA 95120 (March 1990, revised May 1990).
41. C. Mohan, "ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes," *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, Brisbane, Australia (August 1990). A different version of this paper is available as Research Report RJ-7008, IBM Almaden Research Center, San Jose, CA 95120 (September 1989).
42. C. Mohan and F. Levine, "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, CA; Association for Computing Machinery, New York (June 1992). A longer version of this paper is available as Research Report RJ-6846, IBM Almaden Research Center, San Jose, CA 95120 (August 1989).
43. C. Mohan, H. Pirahesh, and R. Lorie, "Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, CA; Association for Computing Machinery, New York (June 1992).
44. C. Mohan, *Transaction Processing System and Method with Reduced Locking*, U.S. Patent No. 5,247,672, (September 1993).
45. C. Mohan, "Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems," *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, Brisbane, Australia (August 1990).
46. *Maximize* must be interpreted more carefully in the context of a multiuser environment. As we see (in the text following this point of reference), sometimes too much parallelism may lead to too much waste of resource, and may not decrease the response time significantly. We must avoid these cases for the benefit of other users of the system.
47. J. Z. Teng and R. A. Gumaer, "Managing IBM Database 2 Buffers to Maximize Performance," *IBM Systems Journal* **23**, No. 2, 211–218 (1984).
48. J. Cheng, D. Haderle, R. Hedges, B. Iyer, T. Messinger, C. Mohan, and Y. Wang, "An Efficient Hybrid Join Algorithm: a DB2 Prototype," *Proceedings of the Seventh International Conference on Data Engineering*, Kobe, Japan; IEEE Computer Society Press, Los Alamitos, CA (April 1991). A longer version of this paper is available as Research Report RJ-7884, IBM Almaden Research Center, San Jose, CA 95120 (December 1990).
49. P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price, "Access Path Selection in a Relational Database Management System," *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (June 1979).
50. G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Atlantic City, NJ; Association for Computing Machinery, New York (May 1990).
51. H. Pirahesh, C. Mohan, J. Cheng, T. S. Liu, and P. Selinger, "Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches," *Proceedings of the Second International Symposium on Databases in Parallel and Distributed Systems*, Dublin; IEEE Computer Society Press, Los Alamitos, CA (July 1990). An expanded version of this paper is available as Research Report RJ-7724, IBM Almaden Research Center, San Jose, CA 95120 (October 1990).
52. G. Lohman, C. Mohan, L. Haas, D. Daniels, B. Lindsay, P. Selinger, and P. Wilms, "Query Processing in R*," *Query Processing in Database Systems*, W. Kim, D. Reiner, and D. Batory, Editors, Springer-Verlag (1985). Also available as Research Report RJ-4272, IBM Almaden Research Center, San Jose, CA 95120 (April 1984).
53. L. Haas, J. C. Freytag, G. Lohman, and H. Pirahesh, "Extensible Query Processing in Starburst," *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, Portland, OR; Association for Computing Machinery, New York (May 1989).
54. M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, "Application of Hash to Data Base Machine and Its Architecture," *New Generation Computing* **1**, 63–74 (1983).
55. S. Su, K. Mikkilineni, R. Liuzzi, and Y. C. Chow, "A Distributed Query Processing Strategy Using Decomposition, Pipelining and Intermediate Result Sharing Techniques," *Proceedings of the Second International Conference on Data Engineering*, Los Angeles, CA; IEEE Computer Society, Washington, DC (February 1986).
56. E. Rahm, *A Framework for Workload Allocation in Distributed Transaction Systems*, Technical Report 13/89, University of Kaiserslautern (September 1989).
57. B. Iyer and D. Dias, "System Issues in Parallel Sorting for Database Systems," *Proceedings of the Sixth International Conference on Data Engineering*, Los Angeles, CA; IEEE Computer Society, Washington, DC (February 1990).
58. R. Lorie and H. Young, "A Low Communication Sort Algorithm for a Parallel Database Machine," *Proceedings of the Fifteenth International Conference on Very Large*

Data Bases, Amsterdam; Morgan Kaufmann Publishers, Incorporated, Palo Alto, CA (August 1989).
59. M. Astrahan, M. Blasgen, D. Chamberlin, J. Gray, F. King, B. Lindsay, R. Lorie, J. Mehl, T. Price, F. Putzolu, M. Schkolnick, P. Selinger, D. Slutz, R. Strong, P. Tiberio, I. Traiger, B. Wade, and R. Yost, "System R: A Relational Data Base Management System," *IEEE Computer* (May 1979).
60. Parallelism is limited to the updating of indices (as is done in NonStop SQL Release 2), and subquery execution for symmetric views with subqueries, where subquery tables are affected by the insert.
61. To some extent, this is expected to be addressed in the context of *scrollable cursor* support by the ANSI/ISO SQL standards committee.
62. C. Mohan, "Interactions Between Query Optimization and Concurrency Control," *Proceedings of the Second International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, Tempe, AZ (February 1992). Also available as IBM Research Report RJ-8681, IBM Almaden Research Center, San Jose, CA 95120 (March 1992).
63. T. S. Liu, G. Tang, A. Tsang, and Y. Wang, "Effective Approach to Query I/O Parallelism Using Sequential Prefetch and Horizontal Data Partitions," *IBM Technical Disclosure Bulletin* **36**, No. 9A (September 1993).
64. A. Shibamiya and M. Zimowski, *Data Base Optimizer Using Most Frequency Values Statistics*, U.S. Patent No. 4,956,774 (September 1990).

Accepted for publication February 7, 1994.

C. Mohan *IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120 (electronic mail:* mohan@almaden.ibm.com). Dr. Mohan has been a research staff member at the IBM Almaden Research Center since 1981. He is a designer and an implementer of the R* distributed DBMS and the Starburst extensible DBMS. He has implemented performance enhancements in DB2/2 and DB2/6000™. He has authored numerous papers on concurrency control, recovery, commit protocols, index management, query optimization, and distributed systems. He is a consultant for all the IBM database and transaction processing product groups. Mohan is the primary inventor of the ARIES recovery method and the Presumed Abort commit protocol, which is an X/Open, OSI, and DRDA standard. His research ideas have been incorporated in the IBM products DB2, DB2/2, DB2/6000, SQL/DS™, MQM/ESA, Shared File System/VM, WDSF/VM, and ADSM, in the IBM prototypes R*, Starburst, and QuickSilver, and in IBM's SNA LU6.2 and DRDA architectures. He has received four IBM Outstanding Innovation Awards, two IBM Research Division Awards, and the eighth plateau IBM Invention Achievement Award for his patent activities (10 issued and 17 pending). In 1992 Mohan was elected a member of the IBM Academy of Technology. He was the Program Chair of the Second International Workshop on High Performance Transaction Systems. He is a Vice-Program Chair for the 1994 International Conference on Data Engineering and an editor of the *VLDB Journal*. Mohan received a Ph.D. in computer science from the University of Texas at Austin in 1981 and a B.Tech. from the Indian Institute of Technology, Madras, in 1977.

Hamid Pirahesh *IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120 (electronic mail:* pirahesh@almaden.ibm.com). Dr. Pirahesh has been a research staff member at the IBM Almaden Research Center since 1985. He has been involved in research, design, and implementation of the Starburst extensible database system and DB2/6000 DBMS product. Dr. Pirahesh has close cooperation with the IBM Database Technology Institute and IBM product divisions. He has been active in several areas of database management systems, computer networks, and object-oriented systems, and has served on many program committees of major computer conferences. His recent research activities cover various aspects of database management systems, including extensions for engineering applications and object-oriented systems, complex query optimization, recursion, concurrency control, and recovery. He is the recipient of several IBM Outstanding Innovation Awards, IBM Research Division Awards, and IBM Invention Achievement Awards. Dr. Pirahesh is an associate editor of *ACM Computing Surveys*. He received a B.S. degree in electrical engineering from the Institute of Technology, Tehran, and M.S. and Ph.D. degrees in computer science from the University of California at Los Angeles.

W. Grace Tang *IBM Santa Teresa Laboratory, 555 Bailey Avenue, San Jose, California 95161.* Ms. Tang joined IBM in 1983 and became the technical leader of the Software Engineering Design Language Precompiler. Later she joined DB2 Optimizer Development and was responsible for the design and implementation of Query I/O Parallelism. She is currently the project leader of Query CPU Parallelism. Ms. Tang received a B.S. degree from National Taiwan University and an M.S. degree in computer engineering from the University of Southern California.

Yun Wang *IBM Santa Teresa Laboratory, 555 Bailey Avenue, San Jose, California 95161 (electronic mail:* wang@stlvm14.vnet.ibm.com). Mr. Wang has been with the DB2 development team at the IBM Santa Teresa Laboratory since 1985. He has been involved in the design and implementation of most of the DB2 enhancements in the query optimizer and query processing area. His current major interests include query optimization, query parallelism and heterogeneous distributed database management. Mr. Wang received a B.S. degree in electrical engineering from the National University of Taiwan and an M.S. degree in computer science from the University of California at Santa Barbara.

Reprint Order No. G321-5546.