

# Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches

Hamid Pirahesh, C. Mohan, Josephine Cheng\*, T.S. Liu\*, Pat Selinger

Data Base Technology Institute, IBM Almaden Research Center, San Jose, CA 95120, USA  
{pirahesh, mohan, pat}@ibm.com

\*Data Base Technology Institute, IBM Santa Teresa Laboratory, San Jose, CA 95150, USA

**Abstract** With current systems, some important complex queries may take days to complete because of: (1) the volume of data to be processed, (2) limited aggregate resources. Introducing parallelism addresses the first problem. Cheaper, but powerful computing resources solve the second problem. According to a survey by Brodie,<sup>1</sup> only 10% of computerized data is in data bases. This is an argument for both more variety and volume of data to be moved into data base systems. We conjecture that the primary reasons for this low percentage are that data base management systems (DBMSs) still need to provide far greater functionality and improved performance compared to a combination of application programs and file systems. This paper addresses the issues and solutions relating to *intra-query* parallelism in a relational DBMS supporting SQL. Instead of focussing only on a few algorithms for a subset of the problems, we provide a broad framework for the study of the numerous issues that need to be addressed in supporting parallelism efficiently and flexibly. We also discuss the impact that parallelization of complex queries has on short transactions which have stringent response time constraints. The pros and cons of the *shared nothing*, *shared disks* and *shared everything* architectures for parallelism are enumerated. The impact of parallelism on a number of components of an industrial-strength DBMS are pointed out. The different stages of query processing during which parallelism may be gainfully employed are identified. The interactions between parallelism and the traditional sys-

tems' pipelining technique are analyzed. Finally, the performance implications of parallelizing a specific complex query are studied. This gives us a range of sample points for different parameters of a parallel system architecture, namely, I/O and communication bandwidth as a function of aggregate MIPS.

## 1. Introduction

The widespread adoption of the easy-to-use products of the relational technology has led to greater expectations on the part of the data base user community. Support for high level ad hoc query languages like SQL has replaced the weeks or months of coding required in the case of the prerelational systems with a few days of coding in order to produce programs which access the data base management system (DBMS) to generate complex reports. Along with this has come the expectation that the responses to the queries should also 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. Coupled to this is the fact that the volumes of data to be dealt with also grow by leaps and bounds as the years go by and computerization goes into full swing. Already there are customers who would like to store more than 100 gigabytes of data in a *single table* and keep it all online all the time! The amount of data kept in a single large relational data base is expected to be in the terabyte range in the coming decade. These trends cause the queries to become *data-intensive*. Furthermore, there is growing emphasis on supporting newer, nonstandard data base applications like VLSI CAD, Computer Aided Software Engineering (CASE), etc., where the volumes of data are enormous compared to the traditional business data processing arena [HaSS88].

With competition intensifying in various sectors of the economy (due to, e.g., deregulation in the airline

industry), and direct-mail marketing becoming more and more common, the complexity of the queries that are being posed is also growing. 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 Prolog, and DBMSs (i.e., data base support for logic programming also has this effect) [Wolf88]. These programming environments allow programmers to write applications which initiate many complex DBMS queries. These trends cause the queries to become *logic-intensive*.

The processing power of affordable parallel computers is expected to be over 1000 MIPS shortly. The combination of massive amounts of data plus enormous processing power creates the opportunity for much more complex queries. Hence, we expect that future DBMSs will have to deal with applications which are increasingly data-intensive and logic-intensive.

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 data bases, respectively. We expect the functionality provided by such query languages to grow considerably. 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). Note that bringing function to data would be a good thing even in a nonparallel system just because it would avoid dragging up to the application level numerous records which subsequently get disqualified by the application when it applies some fancy predicates. Given that applications tend to be sequential, in a parallel DBMS, applying the fancy predicates within the DBMS would allow parallelism to be exploited in evaluating those predicates also, thereby potentially reducing the response time tremendously.

DBMSs will 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, possibly, massive amounts of irrelevant data does not 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 is retrieved. Operations such as outer join, recursion, and sampling [OIRO89] should be handled by DBMSs for the same reason.

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. In most cases, one can hope to get realtime responses to data and logic intensive queries only by exploiting parallelism. This may come as a surprise to some people who might be led to think that the way to attack the response time requirement is to stay with the simpler strategy of no intra-query parallelism, but use faster processors, and larger and larger amounts of memory. The limitations to the improvement of response time via faster processors and larger memories alone relate to the following observations:

- Based on the trends of the recent past, it is clear that the growth in the processing capacity of a uniprocessor or a closely-coupled multiprocessor is not going to be sufficient to provide realtime responses to certain types of complex queries using such systems. At least today, it appears that the \$/MIPS (Million Instructions Per Second) 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., 2GB of real memory due to the 31-bit *real* memory addressing used on the IBM/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. (We will return to this

---

1 Presented at the ACM-SIGMOD International Conference on Management of Data, Chicago, May 1988.

point in the section "2.5. I/O Versus CPU Versus Communication Parallelism".) This is at present necessitating the use of techniques like disk striping [CABK88, SaGa86] and disk arrays [PaGK88] to improve the I/O bandwidth. For a long time, systems like IBM's TPF [Hobs87, Scru87, Siwi77] used disk striping in software to improve *inter*-transaction parallelism. But now, striping is needed for supporting *intra*-transaction 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 disks in parallel.

Therefore, in order to gain price-performance advantages and response time improvements, the trend is towards building a data base machine consisting of a large number of smaller machines and exploiting intra-query parallelism.

## 2. Overall System Architecture Options

### 2.1. Shared Everything Versus Shared Disks Versus Shared Nothing

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 [Bhid88]: (1) *shared disks (SD)* or also called *data sharing* [DIRY89, Haer88, MoNa90, MoNP90, Rahm87, Rahm88, Rahm89b, Shoe86], (2) *shared nothing (SN)* or also called *partitioned* [Bora88a, Ston86], and (3) *shared everything*.

With SD, all the disks containing the data bases are shared amongst the different systems and each system has its own buffer pool. Every system that has an instance of the DBMS executing on it may access and modify any portion of the data base 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 amongst 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 IBM's IMS/VS Data Sharing product [CaHS85, ObSW83, PeSt83, StUW82], TPF product [Hobs87, Scru87, Siwi77] and the Amoeba research project

[MoNa90, MoNP90, SNOP85], in DEC's VAX DBMS<sup>2</sup> and VAX Rdb/VMS products [JoRo89, KrLS86, ReSW89], and in NEC's DCS [SMMTG84]. These systems are using the SD architecture for *inter*-transaction parallelism rather than *intra*-transaction parallelism.

With SN, each system owns a portion of the data base and only that portion may be directly read or modified by that system. That is, the data base is partitioned amongst 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 of [MoLO86]) to coordinate its activities. This is the approach taken in Tandem's Encompass<sup>2</sup> and NonStop SQL<sup>2</sup> [BoPu88, Borr81, Borr84, EGKS89, Tand87, Tand88], Teradata's DBC/1012<sup>2</sup> [DeSB87, Nech86, Tera88], MCC's Bubba [AlCo88, Bora88b, CABK88], and the University of Wisconsin's Gamma [DeGS88, GeDe87, ScDG89].

In the shared everything approach, memory, in addition to disks, is also shared across the processors. University of California - Berkeley's XPRS system has adopted this approach [SKPO88, StAS89]. It has been pointed out in [StAS89] that shared everything 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 [CLSW84, HaJa84], for example, is able to very nicely exploit a shared everything machine like an IBM 3090-600J which has 6 processors.

Arguments in favor of SD are given in [HaSS89] in the context of complex objects and parallelism. For complex objects, it is said that partitioning the data, as is required with SN, is a big problem.

### 2.2. Transaction Monitors

In discussing an overall architecture, the role of data communications [Duqu87, Sche87, SSSHD87] and the transaction monitor (like IMS/DC [McGe77], TUXEDO [AnCK89] or CICS [Serl89]) 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. They are in essence an extension of the base operating system.

---

<sup>2</sup> IBM, AS/400, and OS/2 are trademarks of the International Business Machines Corp. Encompass, NonStop SQL, and Tandem are trademarks of Tandem Computers, Inc. DEC, VAX DBMS, VAX, VAXcluster, and Rdb/VMS are trademarks of Digital Equipment Corp. DBC/1012 is a trademark of Teradata Corp. SYBASE is a registered trademark of Sybase, Inc.

Supporting the transaction monitor and the environment that it needs is essential even in a parallel architecture system. Any existing large application base which relies on such an environment must be accounted for. Resources (CPU, I/O, communication) used in the nonDBMS 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. Tandem's Encompass and NonStop SQL provide 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 frontend machines and the actual data management would be done in a backend (the so called *frontend-backend* configuration) where parallelism would be exploited using machines of a different nature from the frontend machines, then two issues must be addressed. First, the cost of the interactions between the frontend and the backend must be taken into account in evaluating the performance implications of this approach on the transaction workload. This division of labor between the frontend and the backend is bound to increase the overall pathlength of a transaction. This increase will be felt especially in the case of the short transactions of the transaction workload. One way to attack this problem is to support the notion of stored procedures (as in the Sybase<sup>2</sup> DBMS [Corn88, Epst88]) and make the frontend issue a single call to the backend to execute a sequence of SQL statements.

The second issue 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 data base (as in the object-oriented DBMSs), or rules (as in rule-based systems). This trend essentially pushes for 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 backend.

### 2.3. 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 [Ross89] switches can sustain high bandwidths and cover more distances compared to copper interconnects, 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 6-way IBM 3090/600J. Most of the processor to processor communication is likely to be messages relating to global locking and buffer coherency protocols [CaHS85, MoNa90, MoNP90, ObSW83, Rahm88, ReSW89].

With SN, the devices may be locally attached to the *owning* processors, perhaps using cheaper technologies. In this case, the processor to processor communications can be significant if a given complex query is accessing data owned by multiple systems.

### 2.4. Short Transactions and Complex Queries

It is very important that the system architecture that is chosen be such that it 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 are also performing online, short transactions which 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, mostly the transaction workload transfers new data from the real world into data bases. Hence, they are the *producers* of the data from the data base viewpoint. Examples are: transactions originating from Automated Teller Machines, point of sale transactions, 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 data bases on two different systems. One of the data bases is the most up-to-date one and it is against that one that the transaction workload is run. The

other data base is an extracted version of the first one and it is on this extracted data base 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.<sup>3</sup> Additionally, there is the expensive extraction process which needs to be performed periodically and which only gives out-of-date data to the ad hoc query users. Some of the advantages of this two data base 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 data base 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 data bases may have an IMS or TPF [Hobs87, Scru87, Siwi77] system which is running the older transaction workload and from which they are unable to migrate away quickly due to performance and/or application rewrite cost reasons. 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 resources (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 data base requests differently. Some server-based systems do not have such a concept, which leads to very unpredictable response times and wide variances. A resource governor would also be essential to control "runaway" queries. DB2 V2R1, 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 application would be highly tempted to choose fine-granularity (e.g., record) locking [MHLPS89, MoPi90]. But this will make the query workload incur significant locking overhead since queries in general access large number 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* (CS - degree 2 consistency of System R [Gray78]). That is, the read locks are given up as the cursor moves from one record to the next. Even though many DBMSs (like DB2, the OS/2 Extended Edition<sup>2</sup> Database Manager [ChMy88], SQL/DS [ChGY81], and NonStop SQL) support CS, the research literature has concentrated only on repeatable read (RR - degree 3 consistency of System R). More implications of CS on data accesses have been discussed in [MHWC90, Moha89, MoLe89].

The locking pathlength 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** Run the queries without locking and use latches [MHLPS89, MHWC90] to assure physical consistency of the pages being read. IMS supports this type of access via what is called *GO processing*. Relational systems like Tandem's NonStop SQL and IBM's AS/400<sup>2</sup> [AnCo88] 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., 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 children objects. In fact, this problem, to a lesser degree, also occurs with cursor stability. Retrieval of the children at two different times during the course of a query may result in two different sets since the read data is locked only briefly and the data might have been updated in between the two retrievals.
- **Transient Versioning** In this approach, for data that is being modified, one or more older versions of it may be maintained [AgSe89, ChGr85, Reed78, Weih87]. With this support, the query workload would be able to read without locking. Just for

---

<sup>3</sup> It should be mentioned that, for large data bases, even if only one copy of the data is stored, the total cost of the disks used for storing the data base is a major portion of the cost of the complete system configuration.

data that is being modified, a slightly older, but a committed version of that data will be exposed to such transactions. The advantage is that the data base that is being exposed will be internally consistent. The concerns may be that not all the exposed data is up to date and the 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 predeclare themselves to be read-only.

[Moha90] presents a technique, called **Commit\_LSN**, for eliminating, most of the time, the need for locking when CS accesses are made. This technique takes advantage of some information (e.g., the log sequence number [MHLPS89, MoPi90]) that is tracked, for recovery purposes, on every page to conclude, without locking, that all the data in a page is in the committed state. It turns out to be of help in reducing the locking overhead even for update transactions, when record locking is in effect. Concurrency is also improved in conjunction with index concurrency control methods like ARIES/IM [MoLe89]. Many applications of the Commit\_LSN technique are described in detail in [Moha90].

## 2.5. I/O Versus CPU Versus Communication 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 [PaGK88] provide both 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 (e.g., a CPU). As a result, different degrees of parallelism are needed for different resource types. Here, we study the relation-

ship between parallelism of two major resource types in DBMSs: CPU and I/O.

Our objective function 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.<sup>4</sup> 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 *minimal* degree of parallelism, while satisfying our objective function. 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 tuples 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 [TeGu84] 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 **TIDs** (tuple identifiers) of qualified tuples from the index, sort the TIDs by page IDs, and then do the I/Os [MHWC90]. 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 tuples 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.

We would like to study the relationship between CPU and I/O parallelism. One concern that we have

---

<sup>4</sup> *Maximize* must be interpreted more carefully in the context of a multiuser environment. As we see shortly, sometimes too much parallelism may lead to too much wastage of resources, and may not decrease the response time significantly. We must avoid these cases for the benefit of other users of the system.

is 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 as fast as that of CPUs. To study the relationship between I/O and CPU parallelism, consider the problem of accessing, directly or through indices, the base tables. If all the data fits in main memory, then each task is CPU bound, and we need only one task per CPU. Hence, degree of parallelism is the number of available CPUs. If data is 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, especially the mainframe ones. Therefore, we need to have numerous disk arms, as in disk arrays, to keep up with each CPU. Let's go through an example. We assume that the processing capacity of each CPU is 30 MIPS. We consider two types of disks:

1. **Slower disks:** 3MBPS (megabytes per second) bandwidth; 20 ms average seek plus search (i.e., rotational latency) time.
2. **Faster disks:** higher bandwidth, moderately lower seek plus search time. Let's assume that these disks are an order of magnitude better in bandwidth (30MBPS) and half order of magnitude better in average seek plus search time (7 ms).

Let's consider two types of queries:

- **Type 1:** complex queries with numerous sequential table scans;
- **Type 2:** complex queries with numerous TID list data accesses, as explained above (mostly doing random I/Os).

The second type of access 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 and/or ORing [MHW90]). The queries of the first type mainly do sequential I/Os. Hence, for each I/O, the seek/search cost is incurred once for a set of

pages (e.g., 64 pages). In this case, the limiting factor is mostly the data transfer bandwidth of the disk. The second type of queries mainly do random I/Os. Hence, the seek/search time delay is usually incurred for every page. In this case, the seek/search time is the limiting factor.

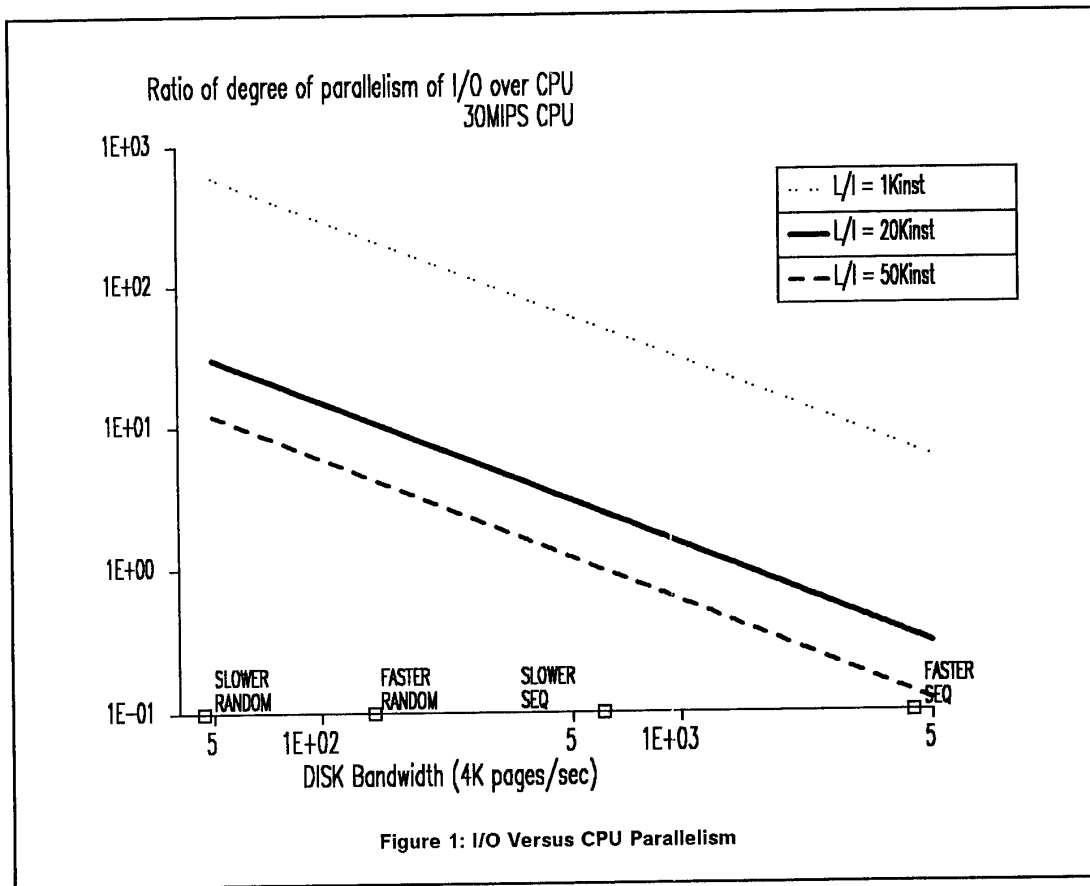
Let's study this more quantitatively. Suppose, for a given query, let  $L$  be the total pathlength and  $I$  the total number of pages retrieved from disk.  $L$  includes CPU instructions for performing I/O, locking, predicate evaluation, sort, join, etc. The ratio  $L/I$  is the average number of instructions of CPU processing incurred for each page retrieved from the disk. This ratio is a good means of characterizing a workload, and we use it to study the balance between CPU and I/O. The value of this ratio depends on the type of a query. Later, in the section "5. A More Quantitative Analysis", we will discuss the details of the performance of a complex query, called *5fanj*. For this query, which is of type 1, the ratio  $L/I$  is about 20K. For queries that have mostly very selective predicates supported by indices, this ratio is typically less than 1K. These queries perform a significant amount of random I/Os. They essentially retrieve one tuple out of every accessed page. For queries that perform significant amount of random I/Os and also significant amount of processing (e.g., sorts, joins, aggregation, etc.) on the retrieved data, the ratio increases significantly.

In a mixed query and transaction workload, we must also consider the effect of transaction workloads on the balance of CPU and I/O. Transaction workloads also perform a significant amount of random I/Os. The  $L/I$  ratio for TP1 transactions [Anon85] is roughly 50K.<sup>5</sup> As we will comment later, these ratios also depend on the buffer sizes. [FTSN89] reports 15 to 45 I/Os per second per MIPS for users of DEC computers. Assuming a page size of 4K bytes, we get  $L/I$  equal to 13 to 34 (this is after converting VAX MIPS to IBM/370 MIPS using the formula given in the reference).

Figure 1 gives the degree of I/O parallelism for each CPU (more precisely, for a task that fully utilizes a CPU) for different  $L/I$  ratios.

For  $L/I = 20$  (*5fanj* type 1 query), the ratio of degree of parallelism of I/O and CPU is about 2.5 for slower disks. This number is about 0.3 for faster disks. In this case, there is not much mismatch between the

<sup>5</sup> Typically, accessing the account table incurs one I/O for the index leaf page, and one I/O for the data page. Updating of the account table incurs one I/O. Each transaction benefits from the caching of teller and branch table. Also, I/Os for the journal table and log (assuming *group commit* [GaKi85] is used) are compensated across multiple transactions. The actual number of I/Os for this heavily depends on the buffer sizes. If we assume one more I/O for all of these, and a pathlength of about 200K instructions, we get 50K per I/O.



degree of parallelism needed for CPU and I/O. If this query is of type 2 and has the same  $L/I$  ratio, the degree of parallelism of I/O and CPU is about 31 for slower disks. This number is about 10 for faster disks. In this case, the degree of I/O parallelism needed is more than an order of magnitude greater than that of CPU, hence there is a significant mismatch between degree of parallelism of CPU and I/O resources.

As the I/O speed increases, we need less parallelism in the I/O subsystem. There are two interesting cases:

1. Degrees of parallelism for CPU and I/O are close to each other.
2. Degree of parallelism for I/O is much more than that for CPU (more than an order of magnitude in the above example).

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 above, 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 CPU. The problem with this approach is that it artificially increases the CPU parallelism significantly (an order of magnitude in our example). This may not be acceptable because as we argued above, 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 [MHWC90]), a CPU task accesses the index and forms a list of pages to be retrieved. This list is given to the I/O subsystem (via the STARTIO instruction). Suppose these pages are stored in a DASD array. The control unit of the DASD array is responsible to initiate I/O (tasks) on different disk devices in parallel to retrieve the pages.

As we increase the efficiency of the DBMS software (more efficient predicate evaluation, sort, join, etc.), the ratio  $L/I$  is reduced. As a result, the curve in Figure 1 moves toward the right, thereby increasing the amount of I/O parallelism that is needed. The same happens when the speed of CPUs increases. The curve moves toward the left with the availability of more main memory which results in less I/Os, or with faster I/O systems. This decreases the amount of needed I/O parallelism. Also, the same happens as the queries become more logic intensive, i.e., as more processing is done on the retrieved pages. Examples are queries with large number of predicates, predicates involving expensive methods on complex data types (e.g., geographic data), and large number of joins (with pipelining).

### 3. Parallel Algorithms

In this section we will discuss ways of parallelizing a query, load balancing issues, and what parallelization impact is on different components of a relational DBMS.

#### 3.1. Targets of Parallelization

There are two (orthogonal) ways of parallelizing complex queries:

- Program Parallelism (PP)
- Data Parallelism (DP)

PP and DP are possible in both the SD and SN architectures.

##### 3.1.1. Program Parallelism (PP)

Let us consider an example which 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 *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 call this the *asynchronous pipeline*. The reasoning behind the name has to do with the fact that the tuples are piped between tasks. But, unlike the *synchronous* pipelining used in sequential plans (e.g., as in System R [SACL79]), here different stages of the pipeline are not executed in a lock-step fashion.

The queue between the producer and consumer tasks is called a *table queue* since its contents are tuples 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 [Grae90], and is also useful in distributed data base management systems and was used in the R\* prototype [LMHDL85, SeAd80]. 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 [HFLP89]. 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 below. Furthermore, the cost of inter-task communication between operators in two different

tasks is considerably higher than that between operators within the same task. In fact, in systems like DB2 [CLSW84], (in most of the cases) the tuples are not copied when they go through synchronous pipelines between operators. This cost is particularly high if tasks are in different processors which are not sharing memory. Analysis of the 5fanj query in the context of a model based on projected pathlengths of MVS and DB2 shows that the pathlength more than doubles if all *synchronous* pipelines are replaced by *asynchronous* pipelines. The extra pathlengths are mostly due to the costs of forming tuples, inserting them into and retrieving them from table queues.

### 3.1.2. Data Parallelism (DP)

DP is the key to supporting a high degree of parallelism. Currently, even in a system which 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 [TeGu84]), a table may be divided up into a number of partitions (one such system, DB2, allows up to 64 partitions, for example). Each partition may be stored on a different device (maybe of even different types) and reorganized independently. DB2's partitioning is based on nonoverlapping key ranges, as specified by the creator of the table. In contrast, systems like TPF, Bubba, DBC/1012, and Grace [KITM83] use hashing to assign tuples 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. We call it the *parallel asynchronous pipeline*. The parallel asynchronous pipeline approach is undesirable from the viewpoint of the tremendous increase in pathlength 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 called the *parallel synchronous pipeline*.

Pipelining helps reduce peaks in data communications and disk I/O. If pipelining is not used, in a SN architecture, the data from the producer is transferred across the network and put on disk at the consumer's system. This may cause a peak in communication [GeDe87], if the producer does not have much work to do (e.g., it is reading the local workfiles and distributing them across the network). But, if

the data is 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. In the section "5. A More Quantitative Analysis", we will revisit this phenomenon. An asynchronous approach and a synchronous approach are presented in [SMLC86] for controlling and managing query pipelines.

## 3.2. Load Balancing Issues

As discussed earlier, 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 discussed before. Computation partitioning must be done such that the load is spread as evenly as possible amongst 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), (2) task level (e.g., load balancing of different tasks accessing a table in parallel). Other discussions on load balancing can be found in [DGRS89, Rahm89a].

### 3.2.1. Physical Resource Level

Here, we consider load balancing among the CPU nodes. Load balancing of I/O and communication resources are not considered. In the partitioned architecture, the data from a disk must be retrieved through the CPU node that the disk is attached to. This node does I/O and locking, applies the local predicates, and extracts the qualified tuples, 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.<sup>6</sup> To reduce the load on this node, we might consider the following alternatives:

1. **Off-loading Predicate Evaluation and Tuple 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 tuples. Another alternative is for the destination node to do the locking. But this requires a

---

<sup>6</sup> Note that the source node also has to deal with updates, causing further load imbalance.

global locking mechanism, as in the SD architecture.<sup>7</sup> The effective communication bandwidth required may also go up considerably because the tuples are not filtered at the source. With this, the SN architecture comes closer to the SD architecture, where raw pages are shared amongst the nodes. But unlike in SD, no buffer coherency protocols are needed.

2. **Data Redistribution.** We can redistribute the data to avoid this situation. This is possible if different pages, or different tuples 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). Note that the SD architecture can handle this 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 particularly a drawback for handling of complex objects. Also, the overhead may be too much for "small" queries - if request has 15 tuple result, it probably will involve at least 15 tasks on 15 nodes.

It is possible that the same set of tuples are demanded from different nodes. If the data is 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 of [Moha90] may be used to reduce contention and to avoid a significant amount of locking overhead.

### 3.2.2. Task Level

With data parallelism, each operator is assigned a portion of the work. Load balancing among tasks still requires extensive research. Here, we briefly mention some of the main issues. Access operators usually have some local predicates, which must be considered in assignment of work to different tasks.

For example, if a *sales* table is partitioned on the value of the month column and a query is issued in which the predicate on month is such that only the December tuples are required, then we need data parallelism at a finer level for processing this query. Each task may handle the data for a particular day. Unfortunately, there may be more data for some days than for others, causing an imbalance in the work to be performed. This situation is worse if we do not know at compile time what month of the year the predicate will specify. This usually happens when the binding for the month variable comes from the application or other parts of the query (e.g., it is a correlation variable passed to a subquery, or it is part of a join column used during access of the inner table).

Similar cases exist for other operators. [lyDi90, lyRV89, LoYo89] address the load balancing issues for the sort operator.

### 3.3. System Components-Level Impact

Providing support for intra-query parallelism in a system designed initially without parallelism in mind requires making changes to a number of components in the system. These are the changes that we would like to address in this section.

Using the System R terminology, 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 management system (*DMS*). We will discuss the enhancements that need to be done at the RDS and DMS levels (corresponding to System R's RD<sub>S</sub> and RSS levels).

Enhancements to the SQL language are necessary to make more parallelism possible within the DBMS. Some implementations of SQL allow the application to provide only one tuple at a time to the DBMS for insertion. As a result, there is little chance for parallelism for such insert operations.<sup>8</sup> We must allow the application to specify a set of tuples and insert them using one SQL command, as is done in SQL/DS. The same is true for the SQL statements *update* or *delete where current of cursor*. Only one tuple at a time can be updated or deleted in this case. We must allow the application to specify update/delete

---

<sup>7</sup> In partitioned architectures, usually all the locking is done locally.

<sup>8</sup> Parallelism is limited to the updating of indices, and subquery execution for symmetric views with subqueries, where subquery tables are affected by the insert.

for a set of tuples in one SQL command.<sup>9</sup> Enhancements to the application program interface (API) are also required to interface DBMS applications which 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 the *compile-time or static optimization*. Another possible approach is the one in which, in addition to doing the compile-time determination of the number of tasks, etc., enough intelligence is built into the run-time support and the plans themselves to dynamically adapt the execution plan based on information about the loads on the different processors, characteristics (size, data distribution, etc.) of the intermediate results, the availability of memory, etc. As can be imagined, this *dynamic optimization* is more difficult 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 without increasing the complexity too much. One idea is a two-phase static optimizer. First, optimize the query ignoring parallelism (i.e., act as if the query will be run in a sequential DBMS). Then, take the query plan chosen above and optimize it further for parallelism. This is the approach chosen in XPRS [StAS89]. 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.

Obviously, there will be cases for which this approach does not produce an optimal plan due to the fragmentation of optimization. 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 resources 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 approach loses. Suppose P1 uses a nonpartitioned index scan and P2 uses a partitioned table scan then P2 may be preferable because it is usually easier to parallelize.

One may consider giving hints to the sequential optimizer to choose alternatives that are easier to parallelize.

As argued above, the parallel optimization of a query depends on how much resources are available. This is usually known at runtime. However, one source of complexity of dynamic optimization is the need for the modification of the plan at runtime, 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 runtime, the degree of parallelism of each part of the plan can be changed based on the actual amount of available resources. This usually does not require complex plan changes at runtime.

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 noncomposite inners. Otherwise, we would be restricted to pipelining as the only means of getting program parallelism amongst 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.

As we will see in the section "5. A More Quantitative Analysis", runtime part of a parallel DBMS must support starting/stopping of tasks, monitoring their progress, and communicating runtime errors. The papers on Gamma [ScDG89] and PROSPECT [DGRSZ88] discuss examples of such support. In a multiuser environment, only a portion of the resources are 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 runtime 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 DBMS level for the following:

- Handling of large buffer sizes, and possibly multiple buffer pools [TeGu84]
- Parallelism
  - I/O parallelism
  - Task structure
- Locking support for parallelized update queries.

<sup>9</sup> To some extent, this is expected to be addressed in the context of *scrollable cursor* support by ANSI SQL standard committee [SQL23].

## 4. Details of Relational Operators

Parallelism is enhanced if we (1) reduce dependencies between operations (e.g., by deferring application of join predicates), and (2) make even the lower level DBMS functions more set oriented (e.g., by performing aggregation during sort).

*Access* refers to accessing permanent tables or workfiles. This operation also includes acquiring any locks, applying the eligible predicates and performing projections. An access may be via indexes 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 [MHW90].

The *join* operation involves taking the results of accessing two or more tables and applying the join predicates. If parallelism amongst 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.

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 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 indexes typically store duplicate keys in a compressed form which 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 [Moha89, Moha90, MoLe89] 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 im-

portant in the context of a data base machine since adopting them could lead to a drastic reduction in communication traffic also.

In the case of the set-oriented insert, delete, and update operations, the selection of the records to be dealt with can go on in parallel with the insert, delete, or update operation. Depending on the consistency level used (RR or CS) during the retrievals, by the time the delete or update operation is executed, the records which 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 sort of problem arises even when there is no intra-transaction parallelism, but the data access is postponed until the index accesses are finished and cursor stability is used during the index access (see [MHW90, Moha90] for more discussions).

The *union* operation will not have much work to do, unless (1) duplicates are to be eliminated and/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 done to remove duplicates would not be possible in most cases.

## 5. A More Quantitative Analysis

A comprehensive quantitative analysis is required to give us the resource requirements of various workloads, and provide us the effect of resource limitations. A comprehensive performance evaluation requires a definition of a benchmark data base and a set of queries for a particular workload. As mentioned before, we are more interested in a complex query workload as opposed to a transaction workload. However, a transaction workload is needed to study the performance of a mix of query and transaction workloads. There has been a substantial effort in the past to define a transaction workload [Anon85, TPC89]. However, work in defining a complex query workload is still in a preliminary stage. Such a workload must include queries with large number of predicates, joins, and aggregations etc. The data base of such a benchmark must be scalable to several terabytes. The recent efforts [Onei89, TuOB89] are steps in this direction. We have introduced a set of such queries using a scaled up version of the DB2 benchmark data base described in [Loos86]. These queries cover the major DBMS operations as far as parallelism is con-

cerned, including predicate evaluation, sort, join, aggregation, and set (and multiset) operations.

An in-depth study of such a benchmark is beyond the scope of this paper. However, we feel it is important to report some results from our effort in this area to give a more quantitative feel about the amounts of different resources needed for complex queries, and what the balance is between these resources. We will explain some of the key observations that arise from studying a complex query, called *5fanj*, and will comment on variations of it. We will give the results of modeling and simulation of 5fanj query, including the CPU, I/O, and communications requirements. We assume a shared nothing system with aggregate CPU power of 10000MIPS, consisting of 334 CPUs each with a processing power of 30 MIPS. We increase the degree of parallelism such that all the CPU power is used. Based on these, we derive the aggregate effective communication and I/O bandwidths.

The SQL formulation of 5fanj query is (also see Figure 3):

```
SELECT ITMTABL, ITPTBAL, PURTABL,
       ITLTABL, ACTTABL
FROM ITMNUMBR, ITPORDER, PURCHASE,
     ITLOCATN, ACTIVLOG
WHERE ITMITEMN=ITPITEMN AND ITPPONUM=PPURPONUM
     AND ITMITEMN=ITLITEMN
     AND ITMITEMN=ACTITEMN
     AND ITMITEMN<'IT010200MNUMBR'
     AND ITPQREM<30000 AND ITPPONUM<'P05600'
     AND ITLWKCN<'WK200' AND ITLTENR<'2000'
     AND ACTLTIME<' 500' AND ACTVDATE<'10'
```

In Figure 3, each node is a table in the FROM clause, and each edge is a conjunct in the WHERE clause predicate. The local (i.e., nonjoin) predicates are represented by loop edges. This query accesses and joins five tables, some with local predicates on them. The query joins the five tables and outputs a column from each table. We assume that the user is able to absorb the output as fast as it is produced. Hence, the response time is determined solely by the DBMS execution time. For each table, the tuple and page cardinality, and the selectivity of the local predicates are given in Figure 2. Initially, we assume that table scans are used to access the data. We will discuss how the results change if we use indices, and if we change the selectivities of the predicates. The tables are physically partitioned across the I/O devices. Each partition is handled by one disk arm. We assume that there is one partition per table per CPU.

Table	Number of Tuples (M=Million)	Number of 4KB Pages (K=1000)	Local Predicate Selectivity %
ITMNUMBR	7.0M	75K	60%
ITPORDER	1.4M	174K	85%
PURCHASE	5.2M	477K	-
ITLOCATN	104.5M	2377K	18%
ACTIVLOG	209.0M	2349K	8%

Figure 2: Characteristics of the Benchmark Data Base and the 5fanj Query

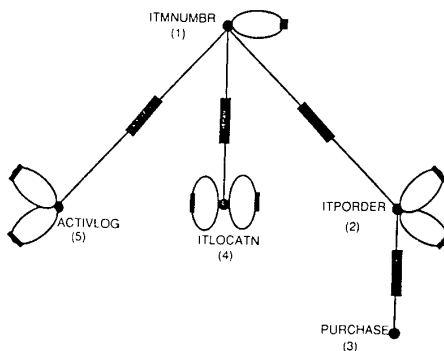


Figure 3: 5fanj Query

The execution plan for this query uses the sort-merge join method, which is usually better than the nested loop method when there is no usable index on the join column of the inner table. The join order we assumed is shown by node numbers in Figure 3 (roughly from the smallest size table to the largest size table). Now, we explain the parallel plan by describing the join between the first two tables. We create one task per partition of ITMNUMBR. Each task accesses its partition, does locking, applies the local predicates, and projects the needed columns of the qualified tuples. This query uses page level locking with cursor stability (level 2 consistency of System R). Then, the task sorts the qualified tuples on the join (with ITPORDER) column and creates the sort runs as temporary workfiles. The same is done for ITPORDER. (During the discussion on scheduling in the section “5.1. Task Scheduling”, we will explain the choices for program parallelism.) For example, we have the choice of accessing and sorting the tables ITLOCATN and ACTIVLOG in parallel or one after another.

For performing the merge of the ITMNUMBR and ITPORDER workfiles and doing a merge join, we create 334 parallel tasks. Each task is assigned a portion of the outer table (ITMNUMBR) join column values and performs that part of the join. Assignment of the range of the join column values for these tasks must be done such that their work is balanced. This may be done by collecting statistics during sort (see [LoYo89]). The result of the join done in each task is sorted and put in workfiles as part of the same task. The join of this intermediate result with the rest of the tables follows the same algorithm. Note that the result of the join of (ITMNUMBR, ITPORDER, PURCHASE) and ITLOCATN

is sorted on the join column ITMNUMBR, which is the same join column for the following join with ACTIVLOG table. Hence, no further sort of the intermediate result is needed for the last join with ACTIVLOG. Hence, the result of join of the first four tables is directly piped (via a synchronous pipeline) to the join with ACTIVLOG.

Figure 4 shows the pathlength for each operator type used in this query. The aggregate pathlength is about 150 billion instructions.<sup>10</sup> Most of our cost estimates are based on IBM’s DB2 [HaJa84]. The cost for Access is separated into two parts. (1) Cost of I/O, page level locking, (2) cost of application of local predicates and extracting the qualified tuples from the pages and projecting out the relevant columns. The cost of I/O and locking is about 10% of the total cost. One reason for such a low cost is that efficient sequential prefetch I/O [TeGu84] was used to access the data. As expected, the cost of sort is dominated by the non-I/O cost. The merge phase, which merges several workfiles, is more I/O intensive because there is little CPU work that needs to be done.

The Access cost is about 38% of the total query cost. 62% of the cost is mainly for sort, merge, and join. In a sense, the Access operators produce the data relevant to the query from the base tables, and the rest of the operators work on that data. Hence, the cost of access may change if the same relevant data must be extracted from a larger data base. We will discuss this variation later in the section “5.4. Further Remarks on Performance”. If shared disks (SD) architecture instead of shared nothing architecture (SN) is used for this query, the cost of access goes up slightly because of global

<sup>10</sup> It should be mentioned that queries with such a high number of instructions are becoming more and more common, particularly for decision support systems. The pathlength of this query is roughly 5 orders of magnitude higher than TPI transactions.

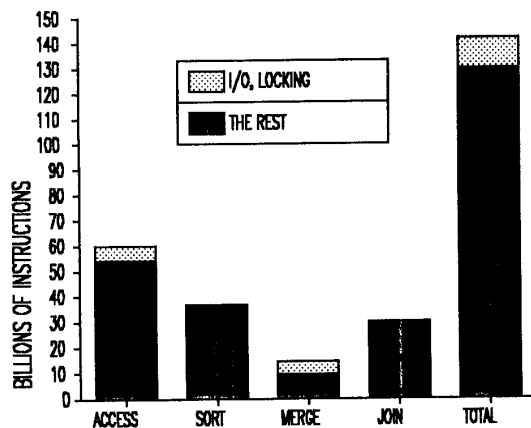


Figure 4: Pathlength for 5fanj Query: per Operator and Aggregate

locking. On the other hand, during the merge phase, no communication is needed to read the data from workfiles, because they are available globally. The difference between the total pathlength on each architecture is insignificant.

### 5.1. Task Scheduling

Determining the needed I/O and communication bandwidth is more difficult because we have to know at any given time what tasks are running; i.e., we have to know the scheduling of the tasks. Each of the tasks to access and sort ITLOCATN and ACTIVLOG is I/O bound if I/O parallelism is not used. Hence, we would like to run them in parallel for better CPU utilization. This results in higher I/O rate. The scheduler's goal is to minimize the response time by maximizing the efficient usage of resources. The scheduler must consider the dependency between tasks (e.g., join of ITMNUMBR and ITPORDER must be done after completion of the access and the sort of each table when using the sort-merge join method). The scheduler must also consider what resources (e.g., CPU, disk arms, memory, and communication ports) are used by each task in order to decide on the start times of the tasks, control of their speeds (e.g., using priorities), and the proper interleavings of different tasks to avoid contention on the above resources.

The complexity of an algorithm for producing an optimal schedule is NP hard. Hence, a workable

scheduler usually needs some heuristics. A complete treatment of the scheduling problem requires further research, and beyond the scope of this paper. For the modeling of our sample problem, we have designed two simplified scheduling algorithms, called *double-stretch*, and *no-stretch*. Neither of these algorithms is the ideal one. We argue that an optimal scheduling algorithm performs somewhere between these two algorithms. Both of these algorithms use, as parameters, the rate that I/O, CPU, and communication resources are consumed by each task to determine the execution schedule. We assume the aggregate CPU MIPS is given. We maximize the usage of CPU resource, and we compute the rate of usage of other resources, such as I/O and communication.

Let's define the following task types for different parts of execution of the 5fanj query. Task type ( $T_1$ ), ( $T_2$ ), ( $T_3$ ), ( $T_4$ ), ( $T_5$ ) for access and sort of tables ITMNUMBR, ITPORDER, PURCHASE, ITLOCATN, ACTIVLOG respectively. As explained above, each task type consists of 334 task instances. Task type ( $T_{12}$ ) for merge and join of output of ( $T_1$ ) and ( $T_2$ ) and sort of the result. Task type ( $T_{123}$ ) for merge and join of ( $T_{12}$ ) output with the output of ( $T_3$ ). Task type ( $T_{12345}$ ) for merge and join of ( $T_{123}$ ) output with the outputs of ( $T_4$ ) and ( $T_5$ ) (as mentioned above, the result of join of the first four tables is directly piped to be merged and joined with the fifth table, hence all of these are part of the task type ( $T_{12345}$ )).

Each task instance runs on one CPU. We can calculate the total amount of CPU, I/O, and communication resource consumption of each task. Based on these, we can calculate the duration of a task. We can calculate the total CPU time essentially based on the pathlength. Likewise, given the I/O parameters of a task (i.e., how many sequential prefetches, how many random I/Os, how many pages, etc.), we can determine the I/O time. Similarly, the communication time can be calculated. Based on these, we determine the time duration of a task, and the rate that CPU, I/O, and communication resources are used. Figure 5 shows one rectangle per task type, where the width is the duration and the height is the CPU utilization of that task. If CPU bound, the height is 100%; otherwise, the height is less than 100%. This diagram also shows the dependency amongst the different tasks. A similar diagram can be obtained for other resources, such as I/O. The critical path is  $((T_4), (T_{12345}))$ . If we remove the tasks associated with this critical path, we get the second critical path:  $((T_5))$ . These two critical paths are numbered 1 and 2 respectively. Likewise, we get  $((T_{1,2,3}), (T_3)), ((T_{1,2}, T_2)), ((T_1))$  for critical paths 3, 4, and 5 respectively.

The *no-stretch* algorithm tries to place these rectangles such that the total height is less than 100% of the aggregate MIPS. This is essentially a bin-packing problem except that the dependency constraints between rectangles must not be violated. Figure 6 shows the result of this scheduling. First, it places the tasks associated with critical path 1,  $((T_4), (T_{12345}))$ . Then, it places the task associated with critical path 2,  $((T_5))$ . Since we have enough CPU MIPS left, task  $(T_5)$  is placed on top of task  $(T_4)$ . That is, they are run in parallel. Then, it places tasks associated with critical path 3,  $((T_{1,2,3}))$ . Note that this task does not fit on top of tasks  $(T_4)$  and  $(T_5)$ . The rest is obvious. The total response time is 23 seconds.

This scheduling algorithm wastes some CPU resource. One way around this is to stretch some of the rectangles, and even break them into pieces and place them in the gaps. We could stretch task  $(T_{123})$  and place it on top of tasks  $(T_4)$  and  $(T_5)$ . This is possible by assigning the left-over CPU resource to task  $(T_{123})$  during this period. This may be done by assigning to this task a lower priority compared with the other two tasks. In a sense, we are slowing down task  $(T_{123})$ . This is acceptable because its

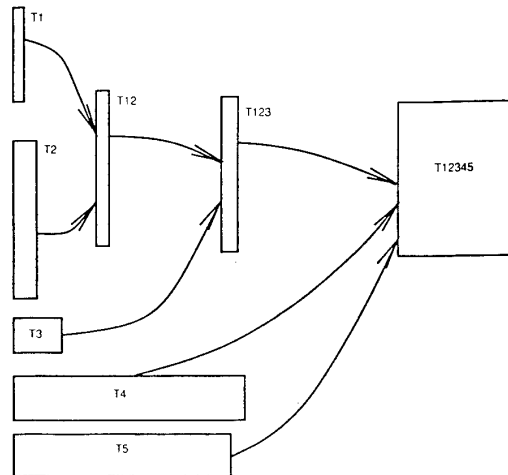


Figure 5: Time-Resource Diagram for Different Task Types

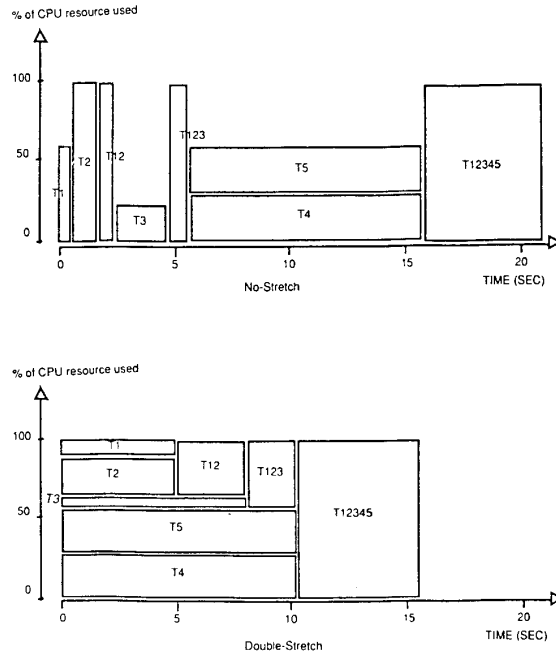


Figure 6: Schedule of Different Tasks for the No-Stretch and Double-Stretch Algorithms.

results are not needed until the start of task ( $T_{12345}$ ). Note that the area of the rectangle, which is the total amount of CPU resource consumed by this task, does not change. As a result of stretching, the height is reduced. Another way is to break a rectangle into pieces and put pieces in the gaps. For this, we start a task, then we stop it for a while, and then we start it again. Combination of stretching and breaking allows us to place the rectangles such that no CPU resource will be wasted. This leads us to the *double-stretch* algorithm. Note that in reality, stretching and breaking may increase the total resource consumption of a task due to some overhead associated with task interference. We expect the design of the scheduler will be such that this cost is not significant.

The result of the double-stretch algorithm is shown in Figure 6. Essentially, this algorithm stretches all the critical paths such that they start at time 0 and finish when their results are needed (note that by definition, any critical path, as defined above, can start at time 0). As a result, it tries to keep as many tasks active as possible at any time, thereby reducing the chances of a CPU being idle. In this algorithm, we place the tasks of critical path 1, ( $T_4$ ),

( $T_{12345}$ ), in sequence. Then, we place the task of critical path 2, ( $T_5$ ), right above ( $T_4$ ). We stretch this task (hence reducing its height) so that it starts at the same time as task ( $T_4$ ). Task ( $T_5$ ) is started at time 0 and finishes when its result is needed. Then, we place tasks associated with critical path 3, ( $T_{1,2,3}$ ), ( $T_3$ ) on top of task ( $T_5$ ). Then, we stretch all the tasks of critical path 3 so that it starts from time 0 and finishes when its results are needed. We do the same for the rest of the critical paths.

At this point, the algorithm calculates the aggregate height (which is a step function). If the aggregate height for any period of time is more than 100% of the aggregate MIPS, then we stretch the rectangles for that period of time, reducing their aggregate height to 100% of the aggregate MIPS. If the aggregate height for any period is less than 100% of the aggregate MIPS, then we increase the degree of parallelism for the tasks active in that period to increase the height. As a result of this, the resource utilization for other resources, such as I/O, may increase.<sup>11</sup> This algorithm does not waste any CPU resource, hence giving the optimum response time under the CPU resource constraint assumption.

One of the major problems with stretching is the increase in memory consumption. A task needs its working set in memory. By stretching, a task uses the resource given to its working memory (set of real memory pages needed for this task) for a longer period of time. This usually results in an increase in the aggregate memory consumption. Another way of looking at this is that as a result of stretching (and breaking) we are increasing the level of multiprogramming, which usually results in greater memory consumption. Hence, we believe that a scheduler must reduce stretching as much as possible.

### 5.2. Effective I/O Bandwidth

Figure 7 shows the I/O rate as a function of time for the no-stretch and double-stretch scheduling algorithms, respectively. For the no-stretch algorithm, the ratio of MIPS over peak I/O rate is about 7 MIPS/MBPS (million instructions per second over million bytes per second). The ratio for the double-stretch algorithm is about 5. The latter ratio was used to study the I/O versus CPU parallelism in the section "2.5. I/O Versus CPU Versus Communication Parallelism". The peak aggregate effective I/O bandwidth is about 2000MBPS. Each 30 MIPS CPU needs about 6MBPS of effective I/O bandwidth. We discussed this in greater detail in the section "2.5. I/O

Versus CPU Versus Communication Parallelism". Observe that the double-stretch algorithm results in higher average effective bandwidth because it has a shorter response time (the same amount of I/O must be done in a shorter period of time). The I/O rate for the no-stretch algorithm has more fluctuation. This is mainly because the double-stretch algorithm keeps more tasks running at the same time, though more slowly, and it introduces a more averaging effect.

### 5.3. Effective Communication Bandwidth

Figure 8 shows the effective aggregate communication bandwidth between nodes. Again, the double-stretch algorithm has less fluctuation than the no-stretch algorithm. For the no-stretch algorithm, the ratio of MIPS over peak effective communication bandwidth is about 9 MIPS/MBPS. The ratio for the double-stretch algorithm is about 16. For a ratio of 16, the aggregate effective communication bandwidth is about 600MBPS.<sup>12</sup> Each CPU needs about 2MBPS effective communication bandwidth. As expected, the average effective communication bandwidth for the no-stretch algorithm is less than that for the double-stretch algorithm since the response time in the former is higher. However, the peak of the no-stretch algorithm is higher. The reason is that the merge tasks that retrieve data from remote

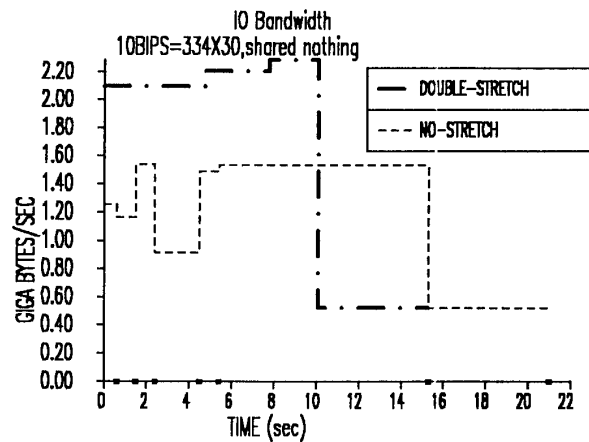


Figure 7: I/O Bandwidth

<sup>11</sup> Note that in this example, the algorithm is always feasible because we only put a constraint on the CPU resource type.

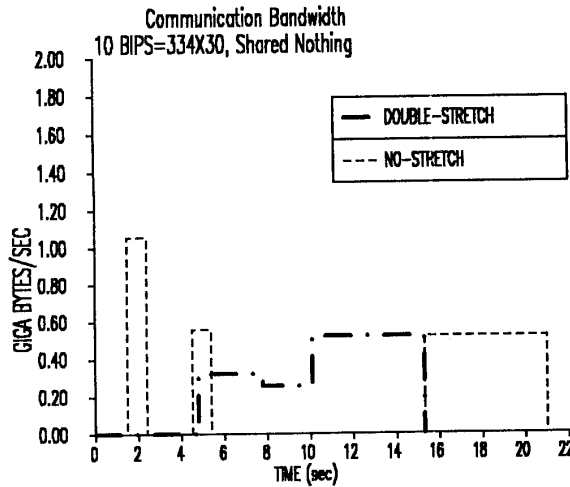


Figure 8: Communication Bandwidth

sort workfiles are not stretched, hence they go at a faster speed, resulting in a higher effective communication bandwidth peak. However, as explained before, the merge operators, which mostly get their inputs from other nodes, pipe their results to the join operators. The join operators (except for one) pipe their results into sort operators. As a result, the speed of the merge operators are controlled (reduced) also by the join and sort operators. Thus, pipelining helps reduce the communication peaks considerably. This is one of the key benefits of pipelining.

#### 5.4. Further Remarks on Performance

In the above query, most of the data accessed from the base tables is used for further processing (sort, join, etc.). This query has low physical selectivity; ratio of relevant data (defined to be the set of tuples that satisfy the local predicates) over the data retrieved from the base tables is low. Now suppose 5fanj query studied above is over a much larger data base (e.g., terabyte), but the relevant data is kept the same. Let's call this the high physical selectivity case, as compared with the low selectivity case studied above.<sup>12</sup> An example of such a situation is physical integration of two existing data bases.

The tuples used by queries on each data base may be scattered across much larger set of pages after integration. In the high physical selectivity case, the access part of the plan must filter out much greater amount of irrelevant data compared with the low physical selectivity case. If no indices are available, then a table scan must be used. As a result, the Access cost will be much higher than what is shown in Figure 4, dominating the rest of the costs, because most of the pages accessed contain little or no relevant data. The effective I/O bandwidth is much higher because the ratio  $L/I$  is much lower, i.e., the CPU does not need to do much work on each retrieved page. Note that this case argues for architectures where the I/O devices are locally attached to the processors. The reason is that in such an architecture it is much cheaper to provide high bandwidth I/O compared with the shared disks case, where all I/Os must go through a processor to disk switch.

In the high physical selectivity case, if indices are available, then usually index ANDing and/or ORing [MHW90] is used to restrict access to pages containing relevant data. In this case, the I/O bandwidth is usually higher than the high physical selectivity case. But the change in the I/O bandwidth is not nearly as drastic as in the case with low physical

<sup>12</sup> Note that this is at least an order of magnitude higher than what the Ethernet LANs provide currently.

<sup>13</sup> Obviously we assume queries are optimized, pushing predicates down for better physical selectivity.

selectivity and no applicable index. The same is true for cost of Access.

If the select list of the query has a large number of fields or large-sized fields, the effective communication bandwidth grows almost proportionately. Techniques similar to semi-join [BeCh81] can be used to reduce the effective communication bandwidth.

## 6. Summary and Conclusions

In this paper, we discussed some architectural alternatives and design approaches for introducing intra-query parallelism in a relational DBMS. We discussed 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 data base 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 which 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 discussed the increasingly important role that disk arrays will play in improving the I/O subsystem's performance to match the latter with that of the CPUs. We also discussed asynchronous pipelining using table queues and the overheads that they impose compared to synchronous pipelining. Parallel synchronous pipelining was pointed out as the preferred method of accomplishing parallelism, whenever possible.

Finally, we emphasized the need for a comprehensive benchmark for complex queries. We presented a quantitative analysis of the usage of different resource types (CPU, I/O, etc.) for a particular complex query. Based on this, we reemphasized the fact that the I/O bandwidth might be very high, if table scans are used and the predicates have high selectivity. In SN, it is usually easier and cheaper to provide high bandwidth I/O since disks are locally attached. With respect to load balancing, we discussed some of the major issues. We may not foresee the 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 (data base reload, unload, reorgan-

ization, etc.), performance of transaction workload on large number of small CPUs, and mix of transaction and query workloads. We are continuing work on the research topics that we have identified in this paper.

## 7. Acknowledgements

We would like to acknowledge a number of our colleagues for their help and insights: Ted Messinger for his efforts to create and maintain a large benchmark data base; Don Haderle for providing extensive technical advice; Akira Shibamiya for helping us understand DB2 execution costs; Jim Brady, John Lindley and Irv Traiger for emphasizing the importance of parallelism; Raymond Lorie, Peter Haas, Bala Iyer, Bruce Lindsay, John McPherson, Inderpal Narang, Kurt Shoens, Irving Traiger for many useful discussions, and Danny Dolev and Barbara Simons for helping us understand the task scheduling techniques.

## 8. References

- ABCGK79** Astrahan, M., Blasgen, M., Chamberlin, D., Gray, J., King, F., Lindsay, B., Lorie, R., Mehl, J., Price, T., Putzolu, F., Schkolnick, M., Selinger, P., Slutz, D., Strong, R., Tiberio, P., Traiger, I., Wade, B., and Yost, R. *System R: A Relational Data Base Management System* **IEEE Computer**, May 1979.
- AgSe89** Agrawal, D., Sengupta, S. *Modular Synchronization in Multiversion Databases: Version Control and Concurrency Control*, **Proc. ACM-SIGMOD International Conference on Management of Data**, Portland, May 1989.
- AICo88** Alexander, W., Copeland, G. *Process and Dataflow Control in Distributed Data-Intensive Systems*, **Proc. ACM-SIGMOD International Conference on Management of Data**, Chicago, May 1988.
- AnCK89** Andrade, J., Carges, M., Kovach, K. *Building a Transaction Processing System on UNIX Systems*, **Proc. Unix Transaction Processing Workshop**, Pittsburgh, May 1989, USENIX Association.
- AnCo88** Anderson, M., Cole, R. *An Integrated Data Base*, In **IBM Application System/400 Technology**, Document Number SA21-9540, IBM, June 1988.

- Anon85** Anon, et al. *A Measure of Transaction Processing Power*, *Datamation*, April 1, 1985.
- BeCh81** Bernstein, P.A., Chiu, D.W. *Using Semi-Joins to Solve Relational Queries*, *Journal of the ACM*, Vol. 28, No. 1, January 1981.
- Bhid88** Bhide, A. *An Analysis of Three Transaction Processing Architectures*, **Proc. 14th International Conference on Very Large Data Bases**, Los Angeles, August 1988.
- BoPu88** Borr, A., Putzolu, F. *High Performance SQL Through Low-Level System Integration*, **ACM SIGMOD International Conference on Management of Data**, Chicago, June 1988.
- Bora88a** Boral, H. *Parallelism and Data Management*, **Technical Report Number ACA-ST-156-88**, MCC, May 1988.
- Bora88b** Boral, H. *Parallelism in Bubba*, **Proc. International Symposium on Databases for Parallel and Distributed Systems**, Austin, December 1988.
- Borr81** Borr, A. *Transaction Monitoring in Encompass: Reliable Distributed Transaction Processing*, **Proc. 7th International Conference on Very Large Data Bases**, Cannes, September 1981.
- Borr84** Borr, A. *Robustness to Crash in a Distributed Database: A Non Shared-Memory Multi-Processor Approach*, **Proc. 10th International Conference on Very Large Data Bases**, Singapore, August 1984.
- CABK88** Copeland, G., Alexander, W., Boughter, E., Keller, T. *Data Placement in Bubba*, **Proc. ACM-SIGMOD International Conference on Management of Data**, Chicago, May 1988.
- CaHS85** Carr, C., Huddleston, R.L., Strickland, J. *Method and Means for the Retention of Locks Across System, Subsystem, and Communication Failures in a Multiprocessing, Multiprogramming, Shared Data Environment*, **U.S. Patent 4,480,304**, IBM, 1985.
- ChGY81** Chamberlin, D., Gilbert, A., Yost, R. *A History of System R and SQL/Date System*, **Proc. 7th International Conference on Very Large Data Bases**, Cannes, September 1981.
- ChGr85** Chan, A., Gray, R. *Implementing Distributed Read-Only Transactions*, **IEEE Transactions on Software Engineering**, Vol. SE-11, No. 2, 1985.
- CHHIM90** Cheng, J., Haderle, D., Hedges, R., Iyer, B., Messinger, T., Mohan, C., Wang, Y. *An Efficient Hybrid Join Algorithm: Design, Prototype, Modelling and Measurement*, **IBM Research Report**, IBM Almaden Research Center, March 1990.
- ChMy88** Chang, P.Y., Myre, W.W. *OS/2 EE Database Manager Overview and Technical Highlights*, **IBM Systems Journal**, Vol. 27, No. 2, 1988.
- CLSW84** Cheng, J., Loosely, C., Shibamiya, A., Worthington, P. *IBM Database 2 Performance: Design, Implementation, and Tuning*, **IBM Systems Journal**, Vol. 23, No. 2, 1984.
- Corn88** Cornelis, R. *Site Autonomy in a Distributed Database Environment*, **Proc. IEEE Compcon Spring '88**, San Francisco, March 1988.
- DeGS88** DeWitt, D., Ghandeharizadeh, S., Schneider, D. *A Performance Analysis of the Gamma Database Machine*, **Proc. ACM-SIGMOD International Conference on Management of Data**, Chicago, May 1988.
- DeSB87** DeWitt, D., Smith, M., Boral, H. *A Single-User Performance Evaluation of the Teradata Database Machine*, **Proc. 2nd International Workshop on High Performance Transaction Systems**, Asilomar, September 1987. Also in *Lecture Notes in Computer Science Vol. 359*, D. Gawlick, M. Haynie, A. Reuter (Eds.), Springer-Verlag, 1989.
- DGRS89** Duppel, N., Gugel, D., Reuter, A., Schiele, G. *Progress Report #6 of PROSPECT*, **University of Stuttgart**, November 1989.
- DGRSZ88** Duppel, N., Gugel, D., Reuter, A., Schiele, G., Zeller, H. *Progress Report #4 of PROSPECT*, **University of Stuttgart**, 1988.
- DIRY89** Dias, D., Iyer, B., Robinson, J., Yu, P. *Integrated Concurrency-Coherency Controls for Multisystem Data Sharing*, **Proc. IEEE Transactions on Software Engineering**, Vol. 15, No. 4, April 1989.
- Duqu87** Duquaine, W. *LU6.2 as a Network Standard for Transaction Processing*, **Proc. 2nd International Workshop on**

- High Performance Transaction Systems**, Asilomar, September 1987. Also in **Lecture Notes in Computer Science Vol. 359**, D. Gawlick, M. Haynie, A. Reuter (Eds.), Springer-Verlag, 1989.
- EGKS89** Englert, S., Gray, J., Kocher, T., Shah, P. *A Benchmark of NonStop SQL Release 2 Demonstrating Near-Linear Speedup and Scaleup on Large Databases*, **Technical Report 89.4**, Tandem Computers, May 1989.
- Epst88** Epstein, R. *RDBMS Architecture Boosts Performance, Reliability, Mini-Micro Systems*, August 1988.
- FTSN89** Serlin O., editor *DEC 9000 to Challenge IBM 3090*, **FT Systems News Letter**, Issue No. 86, Oct. 24 1989.
- GaKi85** Gawlick, D., Kinkade, D. *Varieties of Concurrency Control in IMS/VS Fast Path*, **IEEE Database Engineering**, Vol. 8, No. 2, June 1985.
- GeDe87** Gerber, R., DeWitt, D. *The Impact of Hardware and Software Alternatives on the Performance of the Gamma Database Machine*, **Computer Sciences Department Technical Report #708**, University of Wisconsin at Madison, July 1987.
- Grae90** Graefe, G. *Encapsulation of Parallelism in the Volcano Query Processing System*, **Proc. ACM-SIGMOD International Conference on Management of Data**, Atlantic City, May 1990.
- Gray78** Gray, J. *Notes on Data Base Operating Systems*, In **Operating Systems - An Advanced Course**, R. Bayer, R. Graham, and G. Seegmuller (Eds.), **Lecture Notes in Computer Science**, Volume 60, Springer-Verlag, 1978. Also Available as IBM Research Report RJ2188, IBM San Jose Research Laboratory, February 1978.
- Haer88** Haerder, T. *Handling Hot Spot Data in DB-Sharing Systems*, **Information Systems**, Vol. 13, No. 2, p155-166, 1988.
- HaJa84** Haderle, D., Jackson, R. *IBM Database 2 Overview*, **IBM Systems Journal**, Vol. 23, No. 2, 1984.
- HaSS88** Haerder, T., Schoning, H., Sikeler, A. *Parallelism in Processing Queries on Complex Objects*, **Proc. International Symposium on Databases in Parallel and Distributed Systems**, Austin, p131-143, December 1988. Also Available as Report No. 37/88, University of Kaiserslautern, October 1988.
- HaSS89** Haerder, T., Schoning, H., Sikeler, A. *Evaluation of Hardware Architectures for Parallel Execution of Complex Database Operations*, **Proc. 3rd Annual Parallel Processing Symposium**, Fullerton, p564-578, 1989. Also Available as Report No. 23/89, University of Kaiserslautern, April 1989.
- HFLP89** Haas L., Freytag J.C., Lohman G., Pirahesh H. *Extensible Query Processing in Starburst*, **Proc. ACM-SIGMOD International Conference on Management of Data**, Portland, May 1989.
- Hobs87** Hobson, S. *ALCS - A High-Performance, High-Availability DB/DC Monitor*, **Proc. 2nd International Workshop on High Performance Transaction Systems**, Asilomar, September 1987. Also in **Lecture Notes in Computer Science Vol. 359**, D. Gawlick, M. Haynie, A. Reuter (Eds.), Springer-Verlag, 1989.
- HsDe90** Hsiao, H., DeWitt, D. *Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines*, **Proc. 6th IEEE International Conference on Data Engineering**, Los Angeles, February 1990.
- IyDi90** Iyer, B., Dias, D. *System Issues in Parallel Sorting for Database Systems*, **Proc. 6th IEEE International Conference on Data Engineering**, Los Angeles, February 1990. Also Available as **IBM Research Report RJ6585**, IBM Almaden Research Center, November 1988.
- IyRV89** Iyer, B., Ricard, G., Varman, P. *Percentile Finding Algorithm for Multiple Sorted Runs*, **Proc. 15th International Conference on Very Large Data Bases**, Amsterdam, August 1989.
- JoRo89** Joshi, A., Rodwell, K. *A Relational Database Management System for Production Applications*, **Digital Technical Journal**, No. 8, February 1989.
- KITM83** Kitsuregawa, M., Tanaka, H., Moto-Oka, T. *Application of Hash to Data Base Machine and Its Architecture*, **New Generation Computing**, Vol. 1, pp 63-74, 1983.

- KrLS86** Kronenberg, N., Levy, H., Strecker, W. *VAXclusters: A Closely-Coupled Distributed System*, **ACM Transactions on Computer Systems**, Vol. 4, No. 2, May 1986.
- LDHSY89** Lorie, R., Daudenarde, J., Hallmark, G., Stamos, J., Young, H. *Adding Intra-Transaction Parallelism to an Existing DBMS: Early Experience*, **Data Engineering**, Vol. 12, No. 1, March 1989. Also Available as IBM Research Report RJ6165, IBM Almaden Research Center, 1988.
- LMHDL85** Lohman, G., Mohan, C., Haas, L., Daniels, D., Lindsay, B., Selinger, P., Wilms, P. *Query Processing in R\**, In **Query Processing in Database Systems**, W. Kim, D. Reiner, and D. Batory (Eds.), Springer-Verlag, 1985. Also Available as IBM Research Report RJ4272, April 1984.
- Loos86** Loosley, C. *Measuring IBM Database 2 Release 2 - The Benchmark Game*, **InfoDB**, Vol. 1, No. 2, 1986.
- LoYo89** Lorie, R., Young, H. *A Low Communication Sort Algorithm for a Parallel Database Machine*, **Proc. 15th International Conference on Very Large Data Bases**, Amsterdam, August 1989. Also Available as **IBM Research Report RJ6669**, IBM Almaden Research Center, February 1989.
- McGe77** McGee, W.C. *The Information Management System IMS/VS - Part IV: Data Communication Facilities*, **IBM Systems Journal**, Vol. 16, No. 2, 1977.
- MHLPS89** Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P. *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*, To Appear in **ACM Transactions on Database Systems**. Also Available as **IBM Research Report RJ6649**, IBM Almaden Research Center, January 1989.
- MHWC90** Mohan, C., Haderle, D., Wang, Y., Cheng, J. *Single Table Access Using Multiple Indexes: Optimization, Execution, and Concurrency Control Techniques*, **Proc. International Conference on Extending Data Base Technology**, Venice, March 1990, Springer-Verlag. Also Available as **IBM Research Report RJ7341**, IBM Almaden Research Center, March 1990.
- MISu88** Mikkilineni, K., Su, S. *An Evaluation of Relational Join Algorithms in a Pipelined Query Processing Environment*, **IEEE Transactions on Software Engineering**, Vol. 14, No. 6, June 1988.
- Moha89** Mohan, C. *ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes*, **IBM Research Report RJ7008**, IBM Almaden Research Center, September 1989.
- Moha90** Mohan, C. *Commit\_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems*, **IBM Research Report RJ7344**, IBM Almaden Research Center, February 1990.
- MoLe89** Mohan, C., Levine, F. *ARIES/IIM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging*, **IBM Research Report RJ6846**, IBM Almaden Research Center, August 1989.
- MoLO86** Mohan, C., Lindsay, B., Obermarck, R. *Transaction Management in the R\* Distributed Data Base Management System*, **ACM Transactions on Database Systems**, Vol. 11, No. 4, December 1986. Also Available as IBM Research Report RJ5037, IBM Almaden Research Center, February 1986.
- MoNa90** Mohan, C., Narang, I. *ARIES/SD: A Transaction Recovery and Concurrency Control Method for the Shared Disks Environment*, **IBM Research Report**, IBM Almaden Research Center, Forthcoming, 1990.
- MoNP90** Mohan, C., Narang, I., Palmer, J. *A Case Study of Problems in Migrating to Distributed Computing: Page Recovery Using Multiple Logs in the Shared Disks Environment*, **IBM Research Report RJ7343**, IBM Almaden Research Center, March 1990.
- MoPi90** Mohan, C., Pirahesh, H. *ARIES-RRH: Restricted Repeating of History in the ARIES Transaction Recovery Method*, **IBM Research Report RJ7342**, IBM Almaden Research Center, February 1990.
- Nech86** Neches, P. *The Anatomy of a Data Base Computer - Revisited*, **Proc. IEEE**

- ObSW83** **Compcon Spring '86**, San Francisco, March 1986.
- OIRo89** Obermarck, R., Strickland, J., Watts, V. *Method and Means for the Sharing of Data Resources in a Multiprocessing, Multiprogramming Environment*, U.S. Patent 4,399,504, IBM, August 1983.
- OIRo89** Olken, F., Rotem, D. *Random Sampling from  $B^+$  Trees*, Proc. 15th International Conference on Very Large Data Bases, Amsterdam, August 1989.
- Onei89** O'Neil, P. *Revisiting DBMS Benchmarks*, *Datamation*, p47, 48, 52, 54, September 15, 1989.
- PaGK88** Patterson, D., Gibson, G., Katz, R. *A Case for Redundant Arrays of Inexpensive Disks (RAID)*, Proc. ACM-SIGMOD International Conference on Management of Data, Chicago, May 1988.
- PeSt83** Peterson, R.J., Strickland, J.P. *Log Write-Ahead Protocols and IMS/VS Logging*, Proc. 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Atlanta, March 1983.
- Rahm87** Rahm, E. *Design of Optimistic Methods for Concurrency Control in Database Sharing Systems*, Proc. 7th International Conference on Distributed Computing Systems, Berlin, September 1987.
- Rahm88** Rahm, E. *Design and Evaluation of Concurrency and Coherency Control Techniques for Database Sharing Systems*, Internal Report 182/88, University of Kaiserslautern, 1988.
- Rahm89a** Rahm, E. *A Framework for Workload Allocation in Distributed Transaction Systems*, Technical Report, University of Kaiserslautern, September 1989.
- Rahm89b** Rahm, E. *Recovery Considerations for Data Sharing Systems*, Technical Report, University of Kaiserslautern, October 1989.
- Reed78** Reed, D. *Naming and Synchronization in a Decentralized Computer System*, PhD Thesis, Technical Report MIT/LCS/TR-205, MIT, September 1978.
- ReSW89** Rengarajan, T.K., Spiro, P., Wright, W. *High Availability Mechanisms of VAX DBMS Software*, Digital Technical Journal, No. 8, February 1989.
- Ross89** Ross, F. *An Overview of FDDI: The Fiber Distributed Data Interface*, IEEE Transactions on Selected Areas in Communications, Vol. 7, No. 7, September 1989.
- SACL79** Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., Price, T. *Access Path Selection in a Relational Database Management System*, Proc. ACM-SIGMOD International Conference on Management of Data, June 1979.
- SaGa86** Salem, K., Garcia-Molina, H. *Disk Striping*, Proc. 2nd IEEE International Conference on Data Engineering, Los Angeles, February 1986.
- ScDG89** Schneider, D., DeWitt, D., Ghandeharizadeh, S. *An Overview of the Gamma Database Machine*, Proc. IEEE Compcon Spring '89, San Francisco, March 1989.
- Sche87** Scherr, A.L. *Structures for Networks of Systems*, IBM Systems Journal, Vol. 26, No. 1, p4-12, 1987.
- Scru87** Scrutchin, T. *TPF: Performance, Capacity, Availability*, Proc. IEEE Compcon Spring '87, San Francisco, February 1987.
- SeAd80** Selinger, P.G., Adiba, M. *Access Path Selection in Distributed Database Management Systems*, Proc. International Conference on Databases, Aberdeen, Scotland, July 1980.
- Serl89** Serlin, O. *CICS is Key to IBM's OLTP Success*, UNIX World, November 1989.
- Shoe86** Shoens, K. *Data Sharing vs. Partitioning for Capacity and Availability*, Database Engineering, Vol. 9, No. 1, March 1986.
- Siwi77** Siwec, J.E. *A High-Performance DB/DC System*, IBM Systems Journal, Vol. 16, No. 2, 1977.
- SKPO88** Stonebraker, M., Katz, R., Patterson, D., Ousterhout, J. *The Design of XPRS*, Proc. 14th International Conference on Very Large Data Bases, Los Angeles, August-September 1988.
- SMLC86** Su, S., Mikkilineni, K., Liuzzi, R., Chow, Y.C. *A Distributed Query Processing Strategy Using Decomposition, Pipelining and Intermediate Result Sharing Techniques*, Proc. 2nd

- IEEE International Conference on Data Engineering**, Los Angeles, February 1986.
- SMMTG84** Sekino, A., Moritani, K., Masai, T., Tasaki, T., Goto, K. *The DCS - A New Approach to Multisystem Data-Sharing*, **Proc. National Computer Conference**, Las Vegas, July 1984.
- SNOP85** Shoens, K., Narang, I., Obermarck, R., Palmer, J., Silen, S., Traiger, I., Treiber, K. *Amoeba Project*, **Proc. IEEE Compcon Spring '85**, San Francisco, February 1985.
- SQL23** Melton, J. (Ed.) (*ISO-ANSI Working Draft*) *Database Language SQL2 and SQL3, X3H2-90-001 and DBL SEL-3, ISO/IEC JTC1/SC21/WG3 N986*, December 1989.
- SSSHD87** Sundstrom, R.J., Staton III, J.B., Schultz, G.D., Hess, M.L., Deaton, Jr., G.A., Cole, L.J., Amy, R.M. *SNA: Current Requirements and Direction*, **IBM Systems Journal**, Vol. 26, No. 1, p13-36, 1987.
- StAS89** Stonebraker, M., Aoki, P., Seltzer, M. *Parallelism in XPRS*, **Memorandum No. UCB/ERL M89/16**, University of California, Berkeley, February 1989.
- Ston86** Stonebraker, M. *The Case for Shared Nothing*, **IEEE Database Engineering**, Vol. 9, No. 1, 1986.
- StUW82** Strickland, J., Uhrowczik, P., Watts, V. *IMS/VS: An Evolving System*, **IBM Systems Journal**, Vol. 21, No. 4, 1982.
- Tand87** The Tandem Database Group *NonStop SQL: A Distributed, High-Performance, High-Availability Implementation of SQL*, **Proc. 2nd International Workshop on High Performance Transaction Systems**, Asilomar, September 1987. Also in **Lecture Notes in Computer Science Vol. 359**, D. Gawlick, M. Haynie, A. Reuter (Eds.), Springer-Verlag, 1989.
- Tand88** Tandem Performance Group *A Benchmark of Non-Stop SQL on the Debit Credit Transaction*, **ACM-SIGMOD International Conference on Management of Data**, Chicago, June 1988.
- TeGu84** Teng, J., Gumaer, R. *Managing IBM Database 2 Buffers to Maximize Performance*, **IBM Systems Journal**, Vol. 23, No. 2, 1984.
- Tera88** Teradata *DBC/1012 Data Base Computer Concepts and Facilities - Release 3.1*, **Documber Number C02-0001-05**, Teradata Corp., May 1988.
- TPC89** Transaction Processing Performance Council *TPC Benchmark A, Draft 6-PR Proposed Standard*, 1989. Available from ITOM International Co., POB 1450, Los Altos, CA 94023.
- TuOB89** Turbyfill, C., Orji, C., Bitton, D. *AS<sup>3</sup>AP-A Comparative Relational Database Benchmark*, **Proc. IEEE-Compcon Spring '89**, San Francisco, February 1989.
- Weih87** Weihl, W. *Distributed Version Management for Read-Only Actions*, **IEEE Transactions on Software Engineering**, Vol. SE-13, No. 1, January 1987.
- Wolf88** Wolfson, O. *Sharing the Load of Logic-Program Evaluation*, **Proc. International Symposium on Databases in Parallel and Distributed Systems**, Austin, p46-55, December 1988.