2nd International Symposium on Databases in Parallel & Distributed Systems

Proceedings

Second International Symposium on

Databases in Parallel and Distributed Systems

July 2-4, 1990 *Trinity College, Dublin, Ireland*

Edited by Rakesh Agrawal and David Bell

IEEE Computer Society Press
Los Alamitos, California

Washington • Brussels • Tokyo

The papers in this book comprise the proceedings of the meeting mentioned on the cover and title page. They reflect the authors' opinions and are published as presented and without change, in the interests of timely dissemination. Their inclusion in this publication does not necessarily constitute endorsement by the editors, the IEEE Computer Society Press, or The Institute of Electrical and Electronics Engineers, Inc.

Published by

IEEE Computer Society Press
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1264

Cover designed by Wally Hutchins

Printed in the United States of America

IEEE Computer Society Order Number 2052
ACM Order Number 415901
Library of Congress Number 90-81054
IEEE Catalog Number 90CH2895-1
ISBN 0-8186-2052-8 (paper)
ISBN 0-8186-6052-X (microfiche)
ISBN 0-8186-9052-6 (case)
SAN 264-620X

Additional copies can be ordered from:

IEEE Computer Society Press
Customer Service Center
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1264

IEEE Computer Society
13, Avenue de l'Aquilon
B-1200 Brussels
BELGIUM

IEEE Computer Society
Ooshima Building
2-19-1 Minami-Aoyama,
Minato-Ku
Tokyo 107, JAPAN

IEEE Service Center
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331

ACM
Order Department
P.O. Box 64145
Baltimore, MD 212

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

Preface

This volume presents the Proceedings of the Second International Symposium on Databases in Parallel and Distributed Systems held from July 2-4, 1990, in Dublin, Ireland. It contains the eighteen papers selected for presentation and the abstract of the keynote address by Gene Lowenthal at the Symposium. The Symposium was sponsored by the IEEE Computer Society, ACM SIGARCH, the British Computer Society, the Irish Computer Society, and the Office of Naval Research.

The existence of this Symposium, and the success of the First Symposium acknowledge the growing recognition of the importance of two related challenges for database systems as we enter the 1990s. One is to exploit the potential for flexibility of access to dispersed data which may preexist at nodes of a computer network, or are distributed for some other reason. The other is to meet the often stringent performance requirements for some database queries by invoking the support of parallel architectures based on recent hardware and software developments.

The vitality and importance of these two areas of research is reflected in the general variety and quality of the papers that were contributed in response to the Call for Papers issued for the Symposium. Over eighty contributed papers from thirteen countries were received and they were carefully and thoroughly evaluated on the basis of technical quality, significance of contribution, readability, and relevance to the Symposium. Credit for the high quality of the program must be given to all authors who submitted papers, and to program committee members who devoted time and expertise to reviewing the papers and specifying the program.

Many of our professional associates devoted a considerable amount of time toward making the symposium a success. Besides the program committee members, our grateful thanks are due to the keynote speaker Gene Lowenthal for agreeing to travel to Ireland and share his vision and experience, to Jane Grimson and Sushil Jajodia for coordinating the activities of the symposium and interfacing with our sponsors, to Bruce Hillyer for putting together the tutorials, to John Hughes and Marek Rusinkiewicz for organizing the publicity material, to Sean Baker for taking care of local arrangements, to Dudley Dolan for acting as the treasurer, Joe Brandenburg, Theodore Johnson, and Sham Navathe for providing expert opinion on specific papers, and to Wally Hutchins for preparing this proceedings. Finally, our special thanks to Shalini Agrawal and Barbara Gouck. Their help at different stages of the symposium preparation has been essential.

All who contributed to this Symposium are entitled to feel satisfied with the fact that they have helped meet the challenges of understanding and applying parallel and distributed techniques and systems to database problems. We believe that this proceedings represents a valuable collective contribution to the scientific literature on research and development in this area.

Rakesh Agrawal
IBM Almaden Research Center
San Jose, California, USA

David Bell
Institute of Informatics
Jordanstown, Co Antrim, Northern Ireland

Symposium Committee

General Chairpersons

Jane Grimson
Trinity College, Ireland

Sushil Jajodia
George Mason University, USA

Program Chairpersons

Rakesh Agrawal
IBM Almaden Research Center, USA

David Bell
University of Ulster, N. Ireland

Program Committee

Peter Apers *(Twente U., The Netherlands)*
Stefano Ceri *(Politecnico di Milano, Italy)*
Misbah Deen *(Keele U., UK)*
Dennise Eckland *(Intel, USA)*
Hector Garcia-Molina *(Princeton U., USA)*
Theo Haerder *(Kaiserslautern U., FRG)*
Masura Kitsuregawa *(Tokyo U., Japan)*
Paul Larson *(Waterloo U., Canada)*
Witold Litwin *(INRIA, France)*
Michele Missikoff *(IASI-CNR, Italy)*
Eric Neuhold *(GMD, FRG)*
Gunther Schlageter *(Hagen U., FRG)*
Dennis Shasha *(New York U., USA)*
Witold Staniszkis *(ZETO-RODAN, Poland)*
Kevin Wilkinson *(HP Research, USA)*
Philip Yu *(IBM, USA)*
Dina Bitton *(DB Software, USA)*
Umesh Dayal *(DEC, USA)*
Doug DeGroot *(Texas Instruments, USA)*
Ahmed Elmagarmid *(Purdue U., USA)*
Georges Gardarin *(INRIA, France)*
H.V. Jagadish *(AT&T Bell Labs, USA)*
Ravi Krishnamurthy *(HP Research, USA)*
Miron Livny *(Wisconsin U., USA)*
Douglas McGregor *(Strathclyde U., UK)*
C. Mohan *(IBM, USA)*
Andreas Reuter *(Stuttgart U., FRG)*
Joachim Schmidt *(Frankfurt U., FRG)*
Amit Sheth *(Bellcore, USA)*
Patrick Valduriez *(INRIA, France)*
Ouri Wolfson *(Columbia U., USA)*

Tutorials Chairperson

Bruce Hillyer *(AT&T Bell Labs, USA)*

Local Arrangements Chairperson

Sean Baker *(Trinity College, Ireland)*

Publicity Chairpersons

John Hughes *(Ulster U., N. Ireland)*
Marke Rusinkiewicz *(Houstan U., USA)*

Finance Chairperson

Dudley Dolan *(Trinity College, Ireland)*

Table of Contents

Session I
Keynote Address

Chair:
Robert D. Ryan
Office of Naval Research, London U.K.

KEYNOTE ADDRESS

The Market Environment for Database Machines and Servers

Gene Lowenthal

Cooperative Computing, Inc.
Austin, Texas, U.S.A.

Since the early 70s, the industrial research community has pursued the elusive dream of a commercially successful database machine: a dedicated-function computer based on an architecture specialized and optimized for database functions, with price and performance characteristics substantially beyond what can be achieved with general purpose software and hardware.

During this period, several forces have conspired to frustrate achievement of this goal – forces which are for the most part independent of the DBM research itself. Chief among these is the accelerating pace of advances in microelectronics, which simultaneously creates a moving target for database machine vendors while focusing the beleaguered computer manufacturer's R&D resources on trying to keep up with protracted product life cycles. Meanwhile, successive generations of relational database software products are incorporating sophisticated performance techniques that further challenge the database hardware vendor.

The database server, in contrast, finds itself in a far more hospitable environment. At one time, the database machine (say, in the role of a "backend") and the database server in a network were viewed as minor variations on a common theme. Now the differences are understood to be essential, bringing the server concept in tune with prevailing trends as surely as the backend is in conflict with them. The opportunity for database servers is fueled by the growth of distributed computing and the strength of the "open systems" movement, leading to standards at multiple levels of the relational database architecture.

As database server interface standards (de facto or otherwise) are established, a market for these subsystems will emerge which is both very large and broadly based. But, if the market for database machines is to expand beyond narrowly defined niches, product suppliers must overcome far greater obstacles.

Session II
Parallel Processing of Database Operations

Chair:
Peter Apers
Twente University, The Netherlands

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,[1] 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 systems' 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

CH2895-1/90/0000/0004$01.00 © 1990 IEEE

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 [OlRo89] 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[2] 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[2] and NonStop SQL[2] [BoPu88, Borr81, Borr84, EGKS89, Tand87, Tand88], Teradata's DBC/1012[2] [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.

---

2 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[2] 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.[3] 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[2] 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[2] [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

3 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 relationship 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.[4] 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

---

4 *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 [MHWC90]). 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.[5] 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

---

5 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 tables. 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.