National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services Branch

Direction des acquisitions et
des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

Parallel Hierarchical Database System

by

Yifeng Xu

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario
August 1993
© Copyright
1993, Yifeng Xu

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-89813-5

Canada

Yifeng XU. Parallel Hierarchical Database System

Name

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Computer Science

SUBJECT TERM

0984

SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0488
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 027
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language 0679
Ancient 0289
Linguistics 0290
Modern 0291
Literature 0401
Classical 0294
Comparative 0295
Medieval 0297
Modern 0296
African 0316
American 0591
Asian 0305
Canadian (English) 0352
Canadian (French) 0355
English 0593
Germanic 0311
Latin American 0312
Middle Eastern 0315
Romance 0313
Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion 0318
General 0321
Biblical Studies 0319
Clergy 0320
History of 0322
Philosophy of 0469
Theology 0323

SOCIAL SCIENCES

American Studies 0323
Anthropology 0324
Archaeology 0326
Cultural 0327
Physical 0310
Business Administration 0272
Accounting 0770
Banking 0454
Management 0338
Marketing 0385
Canadian Studies 0501
Economics 0503
General 0505
Agricultural 0508
Commerce-Business 0509
Finance 0510
History 0511
Labor 0358
Theory 0366
Folklore 0351
Geography 0578
Gerontology 0578
History 0578
General 0578

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science 0615
General 0616
International Law and Relations 0617
Public Administration 0814
Recreation 0452
Social Work 0626
Sociology 0627
Criminology and Penology 0938
Demography 0631
Ethnic and Racial Studies 0628
Individual and Family Studies 0629
Industrial and Labor Relations 0630
Public and Social Welfare 0700
Social Structure and Development 0344
Theory and Methods 0709
Transportation 0999
Urban and Regional Planning 0453
Women's Studies

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture 0473
General 0285
Agronomy 0475
Animal Culture and Nutrition 0476
Animal Pathology 0359
Food Science and Technology 0478
Forestry and Wildlife 0479
Plant Culture 0480
Plant Pathology 0817
Plant Physiology 0777
Range Management 0746
Wood Technology

Biology 0306
General 0287
Anatomy 0308
Biostatistics 0309
Botany 0379
Cell 0329
Ecology 0353
Entomology 0369
Genetics 0793
Limnology 0410
Microbiology 0307
Molecular 0317
Neuroscience 0416
Oceanography 0433
Physiology 0821
Radiation 0778
Veterinary Science 0472
Zoology 0786
Biophysics 0760
General 0760
Medical

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Polynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences 0566
General 0300
Audiology 0992
Chemotherapy 0567
Dentistry 0350
Education 0769
Hospital Management 0758
Human Development 0982
Immunology 0564
Medicine and Surgery 0347
Mental Health 0569
Nursing 0570
Nutrition 0380
Obstetrics and Gynecology 0354
Occupational Health and Therapy 0381
Ophthalmology 0571
Pathology 0419
Pharmacology 0572
Pharmacy 0382
Physical Therapy 0573
Public Health 0574
Radiology 0575
Recreation

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences 0485
Chemistry 0749
General 0486
Agricultural 0487
Analytical 0488
Biochemistry 0738
Inorganic 0490
Nuclear 0491
Organic 0494
Pharmaceutical 0495
Physical 0754
Polymer 0405
Radiation 0605
Mathematics 0986
Physics 0606
General 0608
Acoustics 0748
Astronomy and Astrophysics 0607
Elementary Particles and High Energy 0798
Fluid and Plasma 0759
Molecular 0609
Nuclear 0610
Optics 0752
Radiation 0611
Solid State 0463
Statistics 0346
Applied Sciences 0984
Applied Mechanics 0346
Computer Science

Engineering 0537
General 0538
Aerospace 0539
Agricultural 0540
Automotive 0541
Biomedical 0542
Chemical 0543
Civil 0544
Electronics and Electrical 0348
Heat and Thermodynamics 0545
Hydraulic 0546
Industrial 0547
Marine 0547
Materials Science 0794
Mechanical 0548
Metallurgy 0743
Mining 0551
Nuclear 0552
Packaging 0549
Petroleum 0765
Sanitary and Municipal 0554
System Science 0790
Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis,
Parallel Hierarchical Database System
submitted by
Yifeng Xu
in partial fulfilment of the requirements
for the degree of Master of Computer Science

Director, School of Computer Science

Thesis Supervisor

Carleton University
August, 1993

Abstract

There has been a lot of debate in the database community about the suitability of various architectures for transaction processing. Three typical architectures are shared-everything (SE), shared-nothing (SN) and shared-disk (SD), each of which has advantages and disadvantages. A compromising hierarchical architecture (HA) looks promising. There have only been some preliminary favorable analytical results for two-level hierarchical architectures.

In this thesis we specify and design a system and a queueing model for the two-level hierarchical architectures including shared-nothing and shared-everything. Based on the results of the previous studies, many reasonably good choices are adopted for each component of the two-level hierarchical architectures. This thesis provides a performance comparison of the different architectures under a set of reasonable assumptions.

Acknowledgements

I would like to take this opportunity to express my profound gratitude to **Dr. S. Dandamudi**, my thesis supervisor, whose patience and support have been essential to the completion of the thesis, and for devoting his time to the stimulating discussions with such enthusiasm and expertise.

I would also like to thank the following people for their input and support:

- **Jian Xia**, my wife, for proving just how much better she really is.
- **MengYing Xu**, my daughter, for bringing about so much joy to us.
- **Prof. Ekow Otoo** and **Prof. John Neilson** from our school, and **Prof. Tuncer Oren** from Department of Computer Science at University of Ottawa, for their useful comment and advice.
- **Bo Xiao**, for being there when I needed someone to talk to.

Finally, I would like to thank NSERC and the School of Computer Science for financial support.

List of Abbreviations

2HA	Two-level Hierarchical Architecture
2PC	Two-phase Commit Protocol
2PL	Two-phase Locking
ABJ	Adaptive Load Balancing Parallel Hash Join
<i>ABJ</i> ⁺	Extended ABJ
All-Joins	All Join-based Transactions
All-Updates	All Update-based Transactions
CCM	Cluster Cache Manager
CDM	Cluster Data Manager
CPU	Central Processing Unit
CS	Cluster Scheduler
CTM	Cluster Transaction Manager
DBMS	Database Management System
DC	Data Contention
DP	Data Parallelism
GNM	Global Network Manager
GTM	Global Transaction Manager
HA	Hierarchical Architecture
I/O	Input and Output
Mpl	Multiprogramming Level
PP	Program Parallelism

RC	Resource Contention
R/W	Read and Write operation
SD	Shared-Disk
SE	Shared-Everything
SN	Shared-Nothing
TIJ	Tuple Interleaving Parallel Hash Join
WFG	Wait-for-Graph

Contents

Abstract	iii
Acknowledgements	iv
List of Abbreviations	v
List of Tables	x
List of Figures	xi
1 Introduction	1
1.1 Motivation	1
1.2 Three Typical Parallel Database System Architectures	3
1.3 Hierarchical Architecture	5
1.4 Outline of the Thesis	6
2 Background	8
2.1 Multiprogramming and Two-phase Locking	8
2.1.1 Multiprogramming vs. Thrashing	8
2.1.2 Two-phase Locking vs. Thrashing	9
2.1.3 Two-phase Locking vs. Deadlock	10
2.1.4 Summary	13

2.2	Inter- and Intra-Transaction Parallelism	14
2.3	Data Load Balancing with Heat and Temperature	16
2.4	Transaction Reliability and Two-phase Commit Protocol	17
2.5	Performance Evaluation	19
3	Performance Model	21
3.1	HA System Modeling	21
3.1.1	Transaction Workload Model	21
3.1.2	Database Model	25
3.1.3	System Model Assumptions	26
3.2	Host Queueing Model	28
3.2.1	Transaction-Generator	28
3.2.2	Global Transaction Manager (GTM)	28
3.3	Cluster Queueing Model	30
3.3.1	Cluster Transaction Manager (CTM)	30
3.3.2	Cluster Scheduler (CS)	34
3.3.3	Cluster Data Manager (CDM)	35
3.4	Network Queueing Model	38
4	Performance Experiment and Analysis	39
4.1	Simulation Parameter Settings	39
4.1.1	Database and Transaction Parameters	41
4.1.2	System Parameters	43
4.2	Performance Validation	44
4.3	Performance with All Update-based Transactions	48
4.4	Performance with All Join-based Transactions	52

4.5	Performance Sensitivity	52
4.5.1	Sensitivity to Mixed Transactions	55
4.5.2	Sensitivity to Data Skew	56
4.5.3	Sensitivity to System Size	57
4.5.4	Sensitivity to the Database Size	59
4.6	Summary	60
5	Conclusions	63
5.1	Summary	63
5.2	Contributions of the Thesis	64
5.3	Directions for Further Research	65
A	Performance Model Design	67
A.1	Host Simulation Design	67
A.1.1	Initialization Module	69
A.1.2	GTM-PROCESSING Module	69
A.1.3	Events for Host	70
A.1.4	GNM-PROCESSING Module	72
A.2	Cluster Simulation Design	72
A.2.1	CTM-PROCESSING Module	72
A.2.2	Events for Cluster Transaction Manager	77
A.2.3	CS-PROCESSING Module	81
A.2.4	Events for Cluster Scheduler	83
A.2.5	CDM-PROCESSING Module	85
A.2.6	DISK-PROCESSING Module	87
A.2.7	CPU-PROCESSING Module	87
A.2.8	Events for Cluster Data Manager	88
	References	89

List of Tables

- 1.1 A Summary of Research Done 3
- 4.1 The different architectures for 8 processors 39
- 4.2 Default Simulation Parameter Settings 40

List of Figures

1.1	Two-Level Hierarchical Architecture	7
3.1	Two-Level Hierarchical Architecture System Model	22
3.2	Two-Level Hierarchical Architecture Queueing Model	29
4.1	Response Time of 1-1-1, 8-8-1 for All-Updates	45
4.2	Throughput of 1-1-1, 8-8-1 for All-Updates	45
4.3	Abort Ratio of 1-1-1, 8-8-1 for All-Updates	46
4.4	Disk Utilization of 1-1-1, 8-8-1 for All-Updates	46
4.5	CPU Utilization of 1-1-1, 8-8-1 for All-Updates	47
4.6	Network Utilization of 1-1-1, 8-8-1 for All-Updates	47
4.7	Response Time of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All-Updates	49
4.8	Throughput of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All-Updates	49
4.9	Abort Ratio of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All-Updates	50
4.10	Disk Utilization of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All-Updates	50
4.11	CPU Utilization of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All-Updates	51
4.12	Network Utilization of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All-Updates	51
4.13	Response Time of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All Joins	53
4.14	Throughput of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All Joins	53
4.15	Disk Utilization of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All Joins	54
4.16	CPU Utilization of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All Joins	54
4.17	Network Utilization of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All Joins	55

4.18	Response Time of S-8-1, S-4-2, S-2-4 and S-1-8 for Mixed Transactions	56
4.19	Throughput of S-8-1, S-4-2, S-2-4 and S-1-8 for Mixed Transactions	57
4.20	Data Skew Effect on Response Time of S-8-1, S-4-2, S-2-4 and S-1-8 for Mixed Transactions	58
4.21	Data Skew Effect on Throughput of S-8-1, S-4-2, S-2-4 and S-1-8 for Mixed Transactions	58
4.22	Impact of Number of Processing Modules on Response Time for Mixed Transactions	59
4.23	Impact of Number of Processing Modules on Throughput for Mixed Transactions	60
4.24	Impact of database Size on Response Time of S-8-1, S-4-2, S-2-4 and S-1-8 for Mixed Transactions	61
4.25	Impact of database Size on Throughput of S-8-1, S-4-2, S-2-4 and S-1-8 for Mixed Transactions	61

Chapter 1

Introduction

1.1 Motivation

The design of a parallel database system for high speed transaction processing is the main objective of this research. Basically, there are three different architectures, i.e. shared-everything (SE), shared-nothing (SN) and shared-disk (SD) for multiprocessor database systems. There has been a lot of debate in the database community about the suitability of various architectures for transaction processing. Bhide [Bhide 88b] attempted to provide answers about the performance of these architectures under a synthetic data management workload with uniform assumptions. His results show that the shared-everything outperforms the shared-nothing and the shared-disk by a fairly wide margin although it is hard to interpret the results since the model assumes many key low-level phenomena such as memory system performance. However, performance is only one part of the equation. Other considerations, primarily scalability, hardware cost and availability will tend to favor the shared nothing approach. When scalability is thought as a very important factor shared-nothing has dominated the research. Although shared-nothing systems have the best scalability and reliability [Stonebraker 86] they are very sensitive to the data skew problem [Lakshmi 88]. Shared-everything and shared-disk allow the collabo-

rating processors to share the workload more efficiently; however, shared-everything suffers from the limitation of the memory and disk I/O bandwidth and shared-disk suffers from the disk I/O bandwidth problem. As a compromise, Hua et al. [Hua 90] presented a hybrid structure in which shared-everything clusters are interconnected through a communication network to form a shared-nothing structure at the inter-cluster level. The key idea is to use shared-everything clusters to minimize the data skew effect and to use shared-nothing inter-cluster structure for good scalability. Their study showed that the hybrid approach provides the best performance for the selected workload. Besides, Kitsuregawa et al. [Kitsuregawa 90] presented the Super Database Computer (SDC), which is a high-performance relational database server for a join-intensive environment under development at the University of Tokyo. SDC is also a hybrid architecture of shared-nothing and shared-everything. It consists of several processing modules connected to each other through a message-passing inter-connection network. Each processing module itself is designed as a tightly coupled multiprocessor system, which has good computational power. Some encouraging simulation results for single join have been obtained for the SDC, which implies that the hybrid architecture is promising. Actually, from CEDAR [Gajski 83] we have arrived at this kind of hybrid architecture independently. Some other papers [Pirahesh 90, Bhide 88a, Bhide 88b] also suggest the hybrid architecture as a compromise. In fact, hybrid architecture is just a two-level hierarchical architecture (2HA for short).

As mentioned above, Hua et al. [Hua 90] and Kitsuregawa et al. [Kitsuregawa 90] presented the performance of a single join for the two-level hierarchical architecture and Bhide [Bhide 88b] provided the performance of shared-nothing and shared-everything under the assumption of all update-based transactions. This is summarized in Table 1.1 with crosses. In the table, the Mixed refers to the combination of update-based and join-based transactions. Our work involves designing and implementing a multiple-architecture simulator, which provides a uniform test bed for shared-nothing, shared-everything and two-level hierarchical architectures. Using the simulator, we could explore the performance of shared-nothing, shared-

Workload	SN	SE	2HAs
Single Update	X	X	
Multiple Updates	X	X	
Single Join			X
Multiple Joins			
Mixed			

Table 1.1: A Summary of Research Done

everything and two-level hierarchical architectures under a variety of workloads such as multiple updates, multiple joins or mixed transactions, most of which have not yet been studied by the database community so far (see the blanks in Table 1.1).

In the thesis we will study the performance of two-level hierarchical architectures and compare their performance to that of shared-nothing and shared-everything under a set of reasonable assumptions.

1.2 Three Typical Parallel Database System Architectures

Parallel database systems have been an active area of research in database design. Several research prototypes as well as a few commercial products have been built. Most of them belong to one of the three architectures discussed above, namely, shared-everything (SE), shared-nothing (SN) and shared-disk (SD). In a shared-everything system, all disks are directly accessible by all the processors with the same access times and all memory is shared. There are several parallel processing systems that are based on this shared-everything architecture. These include the Sequent Symmetry system [Sequent 86], the DEC Firefly [Thacker 88], the IBM 3090 series of machines and SPUR [Hill 86]. But these are not the database management systems. Examples of shared-everything data servers include XPRS [Stonebraker 88] and to some extent, the SABRE data server [Gardarin 83]. In the

shared-nothing architecture, each disk is connected to a single processor and each processor has its own private memory. Shared-nothing is the approach taken in Tandem's NonStop SQL [Tandem 87], Teradata's DBC/1012 [Neches 88], MCC's Bubba database machine [Boral 90], and the University of Wisconsin's Gamma database machine [DeWitt 90]. In the shared-disk architecture, any processor can access any disk but each processor has its own private memory. Shared-disk is the approach used in IBM's IMS/VS Data Sharing product [Peterson 83, Strickland 82], TPF product [Hobson 87, Scrrutchin 87] and the Amoeba research project [Mohan 91a, Mohan 91b, Mohan 90, Shoens 85], in DEC's VAX database management system and VAX Rdb/VMS [Joshi 91, Kronenberg 86] and in NEC's DCS [Sekino 84]. More recently, for the VAX cluster environment, third-party database management systems like ORACLE and INGRES have been modified to support data sharing [Serlin 91]. Hitachi and Fujitsu also have products which support the shared-disk environment.

The coherency control problem limits the number of processors that can efficiently cooperate in a shared-everything or a shared-disk system. In a shared-everything system, processors share a single global memory address space. The shared memory is typically a physically distributed memory to accommodate the aggregate demand on the shared memory from a large number of processors. An interconnection network is usually used to allow any processor to access any memory module. This communication network, however, increases the memory-access latency. The performance of conventional processors is quite sensitive to memory-access latency. If the memory-access latency exceeds one instruction time the processor must idle until the storage cycle completes. A popular solution to this problem is to have cache memory with each processor. However, the use of caches requires a mechanism to ensure cache coherency. As we increase the number of processors, the number of messages due to cache coherency control (i.e. cross interrogation) increases. Unless this problem can be solved, scaling a shared-everything system into the range of hundreds of processors will be impractical. Similarly in a shared-disk system, it is obvious that inter-processor coherency control is necessitated due to

the caching of the shared database pages in main memory database buffers. The buffer invalidation problem tends to be the limiting factor for the size of shared-disk system.

In contrast, data coherency control is not a problem in a shared-nothing system. Besides, the processor and memory are physically localized in a node, and memory-access latency is not a problem. Nevertheless, shared-nothing systems are very sensitive to the data skew problem [Lakshmi 88]. When the data are seriously skewed, rebalancing the data load among the processing nodes is necessary to obtain good system performance.

From the above discussion, it seems that there is no absolute winner. When the individual processors are very powerful, a small number of processors can be interconnected in a shared-everything structure to avoid the communication overhead and the data skew problems (e.g., IBM 3090 Model 600). On the other hand, if the applications demand a performance level far exceeding the capability of typical shared-everything systems, a microprocessor-based shared-nothing structure offers a solution for handling very large database (e.g., iPSC/2 version of GAMMA [DeWitt 90], NonStop SQL [Englert 89], DBC/1012 [Teradata 88], SDC [Kitsuregawa 90] etc.). But, at the same time, shared-nothing architecture brings about the communication overhead and data skew problems.

Therefore, the main purpose of this thesis is to combine shared-everything with shared-nothing to offset their disadvantages and still retain their advantages. A two-level hierarchical architecture is proposed to achieve this. This two-level hierarchical architecture is the focus of the research.

1.3 Hierarchical Architecture

Hierarchical Architecture (HA for short) is a tree-like structure in which all leaf nodes are shared-everything clusters and all non-leaf nodes comprise of a layered shared-nothing system. For example, the two-level hierarchical architecture consists of a loosely coupled multiple cluster system, in which each cluster is a shared-

everything multiprocessor system. This architecture is shown in Figure 1.1. Let N be the number of clusters in the system, and n be the number of processors in each cluster. We note that the two-level hierarchical architecture has a shared-everything structure when $N=1$, and it becomes a shared-nothing organization when $n=1$.

The motivation is to cluster the processing modules to minimize the communication overhead associated with the shared-nothing structure and to balance workloads within the cluster, and yet each cluster size should be kept small enough to satisfy the limitations of the memory and I/O bandwidth.

1.4 Outline of the Thesis

The remainder of the thesis is structured as follows.

Chapter 2 discusses several important concepts of parallel database systems. These concepts include the multiprogramming level, thrashing, two-phase locking, deadlock, inter- and intra-parallelism, data heat and temperature, transaction failure and two-phase commit protocol etc.

Chapter 3 presents a detailed two-level hierarchical architecture system model and queueing model, which includes a global transaction manager, cluster transaction managers, cluster schedulers, cluster data managers and a global network manager. This chapter also introduces the transaction workload model, database model and the relevant algorithms and protocols for the queueing model.

Chapter 4 provides a detailed performance comparison among the shared-nothing, the shared-everything and the two-level hierarchical architectures under different transaction workloads such as all-updates, all-joins and mixed transactions. In addition, the performance sensitivity of different architectures to mixed workloads, data skew, system size and database size is also provided.

Chapter 5 provides a summary of the thesis. The contributions of the thesis and directions for further research are also given.

Finally, the performance model design for the two-level hierarchical architecture is given in Appendix A.

Figure 1.1: Two-Level Hierarchical Architecture

Chapter 2

Background

2.1 Multiprogramming and Two-phase Locking

2.1.1 Multiprogramming vs. Thrashing

In computer systems with multiple CPU and I/O resources, multiprogramming is usually employed in order to permit a number of tasks to run simultaneously. It enables multiple users to concurrently access the resources of a single system. It also provides for increased utilization of computer system resources, enabling the systems to accomplish more work per unit of time. Like operating systems, most database management systems (DBMS for short) are multiprogrammed in order to permit resource sharing and to maximize transaction throughput. However, it is well known that transaction processing systems can be subject to thrashing. The term thrashing, coined by Denning [Denning 68] for overload effects in virtual storage systems, generally describes a phenomenon where an increase in the load results in a decrease in throughput. Generally speaking, there are at least two distinct potential sources for thrashing behavior. The first source is related to buffer management, and is very similar to the virtual memory thrashing problem faced by operating systems. Transactions use and compete for buffer pages in much the same way that processes compete for virtual memory, so buffer thrashing can occur if the buffer pool becomes over-utilized. This kind of contention for physical resources (memory, disks and processors etc.) is called the resource contention (RC). The

second potential source of thrashing is very different, and seems to be quite unique to database systems. The source of this type of contention is the concurrency control manager, which coordinates transactions' accesses to shared data in order to prevent data consistency problems that could otherwise result. This kind of contention for logical resources (data granules etc.) is called the data contention (DC).

2.1.2 Two-phase Locking vs. Thrashing

Thrashing phenomena in database systems with locking were studied extensively by Tay et al. [Tay 85] using an analytical closed mean value transaction flow model. A similar model was used by Dan et al. [Dan 88] to analyze optimistic protocols. Franaszek and Robinson [Franaszek 85] applied a random graph model that also reveals the thrashing behaviour. A comprehensive analytical framework for data and resource contention in database systems was recently presented by Ryu et al. [Ryu 90] who also report thrashing. Most of the results for these analytic models, including thrashing, were confirmed through simulation studies conducted by Agrawal et al. [Agrawal 87]. All the papers above suggest that a mechanism for controlling the concurrency level is necessary. Further, Tay et al. [Tay 85] and Iyer [Iyer 88] more concretely suggested the criteria of such a control mechanism for blocking concurrency control algorithms such as two-phase locking (2PL) etc. Carey et al. [Carey 90] presented an approach called "Half-and-Half" algorithm to eliminate thrashing in the case of two-phase locking. In this algorithm, there is a local controller added to the standard software architecture of a database management system. The local controller adapts to dynamically control the multiprogramming level of the system so that the performance of the system (e.g. transaction throughput) can be maximized. In addition, Heiss et al. [Heiss 91] also considered the problem of controlling the concurrency level in transaction processing systems as a dynamic optimum search problem. Regardless of any internal details of the system, they are solely interested in the functional relationship of the concurrency level n as the input and the resulting performance P as the output of the system so that the concurrency level n can be dynamically adjusted to maximize the output

performance.

The two-phase locking is the most widely used concurrency control algorithm in the existing systems, especially in shared-nothing database systems. It was shown in a performance study [Agrawal 87] that, for a fairly wide range of database workloads and system configurations, a concurrency control algorithm based on two-phase locking will outperform concurrency control algorithms that use transaction restarts rather than blocking to resolve conflicts between transactions. The simulation results [Carey 89] also reveal that in shared-nothing database systems two-phase locking provides the best performance, followed by the basic timestamp ordering, wound-wait, the optimistic algorithm in that order. Therefore, this thesis will consider the two-phase locking as the concurrency control algorithm. A number of concurrency control performance studies [Balter 82, Carey 84, Franaszek 85, Tay 85, Agrawal 87] have indicated that the two-phase locking can cause thrashing under high levels of data contention. This can occur due to one of two causes. If the system resources are lightly utilized, transaction waiting can lead to thrashing [Balter 82, Franaszek 85, Tay 85, Agrawal 87]. Such wait-induced thrashing occurs because data contention is so high that, due to lock waiting, allowing a new transaction into the system actually leads to a decrease rather than an increase in the number of runnable transactions. In a system with resources that are more heavily utilized, thrashing can occur due to transaction aborts [Carey 84, Agrawal 87]. Such abort-induced thrashing occurs at the point where admitting a new transaction becomes likely to lead to a deadlock, and results from the fact that a transaction that executes for some time and is later aborted leads to wasted resource capacity. Here, allowing a new transaction into the system ends up increasing the system overhead for permitting shared use of the resources and data managed by the database management system.

2.1.3 Two-phase Locking vs. Deadlock

Two-phase locking brings the possibility of deadlock. Since locking realizes consistency by blocking transactions whose lock requests cannot be granted, two or more transactions may hold locks needed by each other, consequently blocking each

other's execution forever.

Deadlocks are usually characterized in terms of a wait-for-graph (WFG). In a database context, a wait-for-graph G is a directed graph where each vertex represents a transaction; each edge of the form $T_i \rightarrow T_j$ ($T \in G$) means that transaction T_i is waiting for transaction T_j , i.e., T_j owns a lock that T_i wants. It can be shown that there exists a deadlock if and only if there is a cycle in the wait-for-graph [Coffman 71, Holt 72]. Strategies based on deadlock detection require that the wait-for-graph be explicitly built and maintained. In continuous detection, the wait-for-graph is always kept cycle-free by checking for and breaking cycles every time a transaction blocks. The connected component of the wait-for-graph involving the newly blocked transaction could be searched using a cycle detection algorithm [Aho 75].

In periodic detection, the graph is searched only periodically for cycles, and all connected components of the graph must be searched in this case. An associated problem is that of selecting a transaction or a set of transactions to restart in order to break any deadlock cycles that form. There are usually five victim selection criteria. These are:

1. Current Blocker: Pick the transaction that blocked most recently (i.e., the one that just blocked, the current blocker, in the case of continuous detection).
2. Random Blocker: Pick a transaction at random from among the participants in the deadlock cycle.
3. Min Locks: Pick the transaction that is holding the fewest locks.
4. Youngest: Pick the transaction with the most recent initial startup time (i.e., the one that began running most recently).
5. Min Work: Pick the transaction that has consumed the least amount of physical resources (CPU+I/O time) since it first began running.

In deadlock prevention, the wait-for-graph is not explicitly maintained. Deadlocks are prevented by never allowing blocked states that can lead to circular waiting.

The Wound-Wait, Wait-Die, Immediate-Restart and Running-Priority etc. belong to the category of deadlock prevention algorithms.

In the case of the timeout strategy for dealing with deadlocks, a transaction whose lock request cannot be granted is simply placed in the blocked queue. The transaction is later restarted if its waiting time exceeds some threshold value. Thus, like the detection strategies, timeout will restart transactions that are actually involved in deadlocks; like the prevention strategies, timeout may also restart some transactions that are not involved in any cycle.

Agrawal et al. [Agrawal 87] studied and compared the relative performance of different classes of deadlock resolution strategies in centralized database systems by means of simulation. A major conclusion of their study is that the choice of the best deadlock resolution strategy is dependent upon the system's operating region. In a low conflict situation, the performance of all deadlock resolution strategies is basically identical. There are just not enough deadlocks in such situations to distinguish among the algorithms. In a situation where resources are fairly heavily utilized, continuous deadlock detection is the deadlock resolution strategy of choice. In this situation, it is best to choose an algorithm that minimizes transaction restarts and hence wastes little in the way of resources. Such situations arise in database systems with physical resources and workloads such that a number of transactions tend to compete for these resources at all times (i.e., primarily non-interactive workloads). If the transactions are primarily interactive, however, with long think times during which locks are held, then a deadlock detection strategy can cause excessive blocking and perform poorly. In such situations, an algorithm like wound-wait, which balances the blocking and restart ratios, provides the best performance. And the continuous deadlock detection outperformed the periodic deadlock detection due to higher blocking and restart ratios associated with the periodic detection, but the performance difference was generally quite small. Among the victim selection criteria, the minimum number of locks was found to be better than the other criteria because if a transaction that is holding a large number of locks is restarted, it has to contend for all of its locks again, increasing the probability of blocking and restarts.

Finally, their study highlighted the difficulty in choosing an appropriate timeout interval for the timeout strategy. In no situation did timeout become the strategy of choice. Therefore, to settle the deadlock problem in our system, we shall use periodic deadlock detection for global checking similar to the method suggested in [Carey 89]. There are two reasons for choosing periodic detection for global deadlocks. First, the message cost tradeoffs in the distributed case will make periodic detection cheaper. For example, IBM system R uses continuous detection but the distributed System R*, uses periodic detection. Second, the probability that the system would enter a deadlock state is relatively low according to [Gray 81]. Also because of the rare deadlocks in a large database system we can use deadlock prevention for local clusters, which means that deadlock prevention applies whenever a transaction with write operation blocks.

We will abort the transaction with min-locks instead of the youngest [Carey 89] victim selection criteria to resolve deadlocks. The reason for this selection is as follows. Usually, larger transaction requests more locks than a small transaction. Aborting min-locks implies aborting the transaction with minimum work done. Besides, we will use an adaptive periodic method for global deadlock detection, as used in GAMMA [DeWitt 90].

2.1.4 Summary

In summary, the choice of the multiprogramming level is very important for the performance of transaction processing systems. Unfortunately, the optimal multiprogramming level is workload-dependent, and the system workload may vary over time. So the study of multiprogramming level is significant. Under certain workload conditions we could find an optimal multiprogramming level so that system's physical resources are not over-utilized and data contention is not very high. In the case of time-varying workload an adaptive algorithm that dynamically changes the multiprogramming level should be used to maximize the performance of the system.

We will also explore the overall performance under data contention and resource contention. Data contention is determined by the multiprogramming level, database

size, granule size, transaction size and transaction service time etc. Resource contention is related to the machine size, i.e. the number of CPUs, the number of disks and the number of interconnection networks, and their functionality.

2.2 Inter- and Intra-Transaction Parallelism

In a transaction processing system, there are two kinds of parallelism: inter- and intra-transaction parallelism. Multiprogramming is related to inter-transaction parallelism, which allows many transactions to execute simultaneously. Intra-transaction parallelism exploits parallelism within a transaction. There are two orthogonal ways of implementing the intra-transaction parallelism in transaction processing.

- Program Parallelism (PP): by this we mean the execution of multiple operations of a given transaction in parallel.
- Data Parallelism (DP): by this we mean the execution of a single operation by operating on its input data (most possibly, different pieces of data) in parallel.

An execution plan is a partially ordered set of operators [Haas 89]. Examples of operators are index or data access and predicate evaluators, sort, join, aggregation etc. The number of operators depend on the complexity of queries. Obviously, the degree of parallelism obtainable by program parallelism 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 dependencies among operators.

Data parallelism is the key to support a high degree of parallelism. Currently, even in a system which does not employ parallelism within a query, a table or relation may be divided up into a number of partitions and each partition may be stored on a different device and reorganized and processed independently.

Owing to the shared-nothing systems' data partitioning, shared-nothing systems are good for both inter- and intra-parallelism. For example, the Non-Stop SQL employs inter-transaction parallelism; DBC/1012, GAMMA and Bubba machines all employ intra-transaction parallelism in addition to inter-transaction parallelism.

A performance evaluation of the Non-Stop SQL for a standard multi-user Debit Credit workload showed linear-throughput gains as the system size was increased from 2 to 32 processors [Tandem 88], demonstrating that inter-transaction parallelism can be very effective in a distributed database machine architecture. Recent measurements of the GAMMA showed that intra-transaction parallelism can be highly effective as well, as impressive single-user speedups were obtained for a variety of queries [DeWitt 88]. Because hierarchical architecture is a generalization of shared-nothing both inter-transaction and intra-transaction parallelism (the combination of program parallelism and data parallelism) will be included in the hierarchical architecture systems.

The increased parallelism, however, is not always better. If the response time is considered as the important metric, the system's objective would be to maximize the use of the given limited resources to minimize the response time up to a threshold. Different degrees of parallelism may satisfy this objective. One question is what the degree of parallelism should be. Sometimes too much parallelism may lead to too much wastage of resources, and may not decrease the response time significantly. Pirahesh et al. [Pirahesh 90] argued that it is important to find the minimal degree of parallelism while still satisfying the objective. They claim 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 decrease 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 [Teng 84] 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.

In addition, parallelism is not always helpful. Carey et al. [Carey 89] examined the response time of a shared-nothing system as a function of data partitioning and intra-parallelism while holding the size of the system constant. They found that the parallelism simply doesn't help when the workload is high. The reason is that

splitting a transaction into pieces indeed makes each piece a fraction as long, but under high loads each node becomes the reciprocal times of the fraction as loaded (thus nullifying any gain due to parallelism). They also observed that the effects of speedup due to parallelism are subtle in the case of the larger database, but they are clear in the smaller case, where data contention is more significant. This is due to the fact that moving to parallelism relieves data contention.

In summary, parallelism is a very difficult subject which is related to workload, resources, database size and granule size etc. Without doubt it becomes one of the main topics in parallel database systems.

2.3 Data Load Balancing with Heat and Temperature

In the data management context, common to all the existing shared-nothing systems is the notion of horizontal partitioning of data [Ries 78]. A relation is chopped up into a number of fragments each of which resides on a different node. Programs execute where the data “live”. The idea received more rigorous treatment in a variety of contexts by several authors. [Kim 85, Salem 86] focus on response time improvements due to horizontal partitioning. In [Livny 87] both throughput and response time advantages were characterized. Finally, [Patterson 88] also considered reliability issues. Probably the most thorough discussion of data placement is given in [Copeland 88]. This paper describes data placement in Bubba. It is argued that a relation is the most appropriate unit for placement (robust statistics, potential use by the optimizer). Associated with each relation are its heat (frequency of access) and its temperature (heat/size). Heat is used in determining how many nodes to spread a given relation over; hotter relations require more I/O “ports”, or a higher degree of spread. Temperature is used in determining whether a relation should be memory resident; higher temperature relations require higher bandwidth ports. The data placement algorithm first determines the degree of spread and memory residency. It then assigns relations to nodes so that each node’s temperature and

heat are roughly equivalent to the others.

Bhide et al. [Bhide 88a, Bhide 88b] have made use of the heat idea in their workload simulation. They statically assume that transactions access pages according to Zipf's Law with 80% of the accesses going to 20% of the database. The 20% frequently accessed pages are distributed uniformly over all the disks and also over all the processors for shared-nothing. Data placement balancing is important because the transaction processing workload distribution is dictated by data placement in most of the shared-nothing systems. Nevertheless, dynamic reorganization of data placement is still an unsolved issue.

In our architectures we will apply the horizontal partitioning, the heat and temperature concepts to our simulations.

2.4 Transaction Reliability and Two-phase Commit Protocol

Transaction reliability is achieved through fault-tolerance techniques that recover from many kinds of failures. Those techniques are implemented in a database management system module traditionally called the recovery manager.

In a computer system, failures can be caused by human errors, transaction programming errors, or hardware component crashes. Human errors (or malicious intentions) and transaction programming errors can be detected and, it is hoped, corrected by better data protection and semantic integrity control. The most important types of failure that the recovery manager is concerned with are transaction, system, and media failures. A transaction failure occurs when a transaction cannot proceed because of an action failure, a deadlock situation, an incorrect ordering of concurrent accesses, or some other problem. The transaction must be aborted and may be reinitiated. A system failure occurs because of a serious error in a system table or a hardware crash (such as a power failure). The system must be stopped and restarted. The content of main memory is assumed to be entirely lost because of system restart. Secondary memory is not affected by a system failure. A media

failure occurs with a nonrecoverable error on disks. Media failures can be due to a hardware problem (such as head crash) or a software error that makes the disks unreadable. A subset of data on disk is lost. This is the most serious type of failure.

A transaction failure can occur several times per minute. A system failure can occur several times per month. A media failure arises only several times per year but may not be recoverable.

The objective of reliability is to recover a consistent database state after failure while minimizing user interaction. Ideally, recovery should be transparent to the user. The single principle on which all reliability algorithms are based is data duplication: corrupted or lost data should always be recoverable from the redundant copies. Managing this duplication is expensive, for it can well increase the number of disk accesses to process updates.

Atomic operations have been recognized as effective tools for implementing reliable distributed systems and commit protocols are the means of implementing atomicity. The simplest and most popular protocol is the two-phase commit protocol (2PC for short), which has been devised for distributed systems. It is the approach taken in Tandem's NonStop SQL, Teradata's DBC/1012, MCC's Bubba and the University of Wisconsin's GAMMA.

In our architectures, we will adopt an optimized version of the distributed two-phase commit protocol [Gray 78] which has a centralized implementation. The analytical results from [Chen 89] reveal that, for this protocol, the coordinator is the bottleneck when the coordinator and the participants have the same service rate that is not very high. So in our architecture, we will adopt a strategy that will uniformly assign a subhost to coordinate among all the clusters and thus load balancing among clusters can be well maintained.

In addition, in order to achieve peak performance, we only consider the transaction failure and recovery, and ignore the system failure, media failure as well as their recovery in our architectures. Therefore, in our simulation the distributed two-phase commit protocol works under the normal operation, i.e., no site failures.

2.5 Performance Evaluation

Performance evaluation of database machines can be done with various objectives, which can be grouped into three classes:

1. evaluation of a specific design to quantify its performance in some way and, if possible, to improve it;
2. comparison with a conventional system;
3. comparison of various architectural designs.

Performance evaluation is a hot topic for database machines, as this kind of special purpose system was specifically meant to improve the speed and the efficiency of database applications. There is, therefore, an obvious need to measure the performance of the database machines and to compare them among themselves and with the traditional database management systems.

Efficiency is the performance metric that is most widely studied, and common background and widely accepted indexes already exist in performance modeling and evaluation of computer systems. For this reason, the term “performance evaluation” usually refers to efficiency aspects.

We can divide database machine performance indices into four classes referring to system work, responsiveness, productivity and utilization.

System work indices give a global measure of how much the database machine has to work for carrying out a certain transaction. Static execution cost is taken as the representative index. The static execution cost is the index usually obtained by deterministic analysis. The execution of a transaction is divided into a sequence of steps. The cost of each step, i.e., the time necessary for its execution, is calculated and the execution cost is obtained by adding up these partial costs.

Responsiveness indices give a measure of the time necessary for processing a given transaction. Execution time and response time are typical indices of this class. The execution time is a global index, which measures the time necessary for executing

a transaction in a dedicated machine. It is of limited use because it does not take a multiuser environment into account. However, response time measures database machine performance in a multitransaction environment. We call response time the time which elapses between the beginning of the transaction and its end, when a mix of transactions is running in the system. This index is equal to the execution time only if the mix is composed of only one transaction and the machine is completely dedicated to its execution.

Productivity indices give a measure of the amount of information processed by the system in a unit of time. The throughput, i.e., the number of transactions executed in the unit time, is the most important index in this class. Generally speaking, throughput is the quantity of information processed in a unit time. When it is used as a global index for evaluating database machine performance, it measures the number of transactions executed per unit time.

Utilization indices give a measure of the length of time specific components of the database machine are used for during the whole transaction. Utilization is a local index which refers to a specific component of the system. It is defined as the ratio between the length of time the specified component is used for during a given interval and the duration of that interval.

In our simulation study, we will take response time, throughput as the global indices because there is no fixed transaction for deterministic analysis and there is a multitransaction environment, as well as take utilization as the local index for locating bottlenecks and adjusting the system.

Chapter 3

Performance Model

3.1 HA System Modeling

Figure 3.1 shows a two-level hierarchical architecture system model. When a transaction is received from a user terminal, the Host distributes the transaction to all the relevant clusters via the Global Interconnection Network. Since the data is distributed across the clusters (actually across the disks) a transaction can be executed by all clusters simultaneously. Each cluster maintains a queue of transactions and schedules a transaction for execution independent of the other clusters in order to maximize its access operations and to minimize its idle time. Besides, transactions are also executed within the clusters concurrently. Host is mainly responsible for interfacing and preprocessing. The clusters do all of the primary database operations.

3.1.1 Transaction Workload Model

Workload models are very important in performance modeling. Most existing workload models for database system performance evaluation are too special to be adopted. [Carey 84, Carey 88, Carey 89], [Bhide 87, Bhide 88a, Bhide 88b] and [Agrawal 87] all modeled a transaction as a sequence of read/write operations from the concurrency point of view. They ignored the communication overhead among the processing modules which is very serious in database systems. [Hua 90, Hua 91] selected a join as the workload model to study the performance of a hybrid architecture.

TERM _x	-----	Terminal x
GTM	-----	Global Transaction Manager
GNM	-----	Global Network Manager
CTM	-----	Cluster Transaction Manager
CS	-----	Cluster Scheduler
CDM	-----	Cluster Data Manager
CRM	-----	Cluster Recovery Manager
CCM	-----	Cluster Cache Manager

Figure 3.1: Two-Level Hierarchical Architecture System Model

The emphasis of these studies has been on the communication overhead problem. However, Hua considered an analytical model for only a single join operation and avoided the concurrency behavior of the system. For example, what is the effect of many join transactions operating simultaneously in the system? What is the coexisting effect of join and update operations? etc. Therefore, in our transaction workload model, we would like to introduce both the update-based operations such as read/write sequence and query-based operations such as join, selection, projection etc.

Update-based transaction model

Update-based transaction is just a sequence of read and write operations on pages because page is the most suitable unit for disk and CPU processing. The abstract form of each operation is $\langle \text{Read or Write}, \text{RelationName}, \text{PageName} \rangle$ whose meaning is to read or write a page PageName for RelationName. For the write operation, we assume any modified page has to be read before it is written, i.e., no blind write is allowed.

Query-based transaction model

Query-based operations are just read-only operations because the intermediate results are assumed to be kept in the temporary area of the disks and no permanent modification of the physical database is involved. In a multiprocessor-based database system, database is usually declustered among multidisks due to disk capacity limitation and for efficient parallel processing. Partitioning techniques are particularly suitable for selection and projection operations, which can be processed locally. So they can be modeled by read-only update-based transactions. However, serious problems arise with the join operation, which incurs considerable communication overhead. Therefore, in the hierarchical and shared-nothing architectures, consideration of the join operation transaction is necessary and important. For simplicity and without loss of generality, we only consider join transactions that is a join operation between two relations.

For the join operation, we take hash-based join as the join method in our two-level hierarchical architecture. The rationale for this is as follows. There are several relational join algorithms that are suitable for single processor as well as multiprocessor environments. Nested-loop, Sort/merge and Hash-based algorithms are the most commonly used algorithms. The analysis in [Lakshmi 88] shows that the nested-loop algorithm is useful only when the relations to be joined are relatively small. Furthermore, when the relations are large, hash-based algorithms are superior to sort/merge provided the relations are not already in sorted order or the final result need not be sorted before presenting to the user. Because multiprocessor or multicomputer systems are mainly aimed at very large relations ($\geq 10^6$ tuples and 200 bytes each tuple) hash-based join methods should work better.

Schneider [Schneider 89] claimed that, besides being faster under most conditions, hash-based join algorithms have the property of being easy to parallelize because tuples of a relation in one bucket are never joined with tuples of the other relation in another bucket and thus corresponding buckets can be handled independently. After comparing the performance of four parallel hash-based join algorithms (Sort/merge with Hash, Simple Hash, Grace Hash-join and Hybrid Hash-join) in a shared-nothing environment, he also claimed that in the case where the join attribute values are highly skewed and the available memory relative to the size of the smaller relation is limited, the optimizer should choose a non-hash-based algorithm such as sort-merge. Actually it is the shared-nothing architecture that is very sensitive to the data skew problem. If we could eliminate some negative effects of non-uniformly distributed inner relations with some other useful architectures such as shared-everything (shared-everything has the least-skewed effect) hash-based join method can still work well.

More importantly, there have been several hash-based join algorithms to deal with the data-skew problem such as Tuple Interleaving Parallel Hash Join (*TIJ*), Adaptive Load Balancing Parallel Hash Join (*ABJ*) and Extended Adaptive Load Balancing Parallel Hash Join (*ABJ⁺*) [Hua 91]. The analysis in [Hua 91] has shown that these schemes are able to provide savings over conventional parallel hash-based

join algorithms even at low skew conditions. In addition, Hua et. al suggest that *ABJ* algorithm should be used if the degree of skew is mild ($\sigma < 60\%$), where σ is called the degree of skew and means that, for each of the joining relations, the skewed processing node has $\sigma \times 100\%$ more tuples than each of the remaining processing nodes; *ABJ*⁺ algorithm should be used if the degree of skew is significant ($\sigma \geq 60\%$); and *TIJ* algorithm should be used in lieu of *ABJ*⁺ algorithm if the relations are very seriously skewed initially, and the communication bandwidth is sufficiently large. *TIJ* and *ABJ*⁺ are better only in the high skew situations because they have the I/O and communication overheads associated with the deferred bucket allocation strategies, which are more serious than the mild skew problem. If we could eliminate the partial I/O and communication overhead problems, *TIJ* and *ABJ*⁺ can do better even if the degree of skew is mild.

Based on these observations, the hash-based join has been selected as the join algorithm in the two-level hierarchical architecture because the clusters of two-level hierarchical architecture are shared-everything architecture and *ABJ*⁺ has been chosen for the two-level hierarchical architecture because the I/O and communication overhead among a cluster can be reduced.

3.1.2 Database Model

We model the database as a collection of relations, each of which is declustered uniformly among all the disks. A set of tuples comprises a relation. The partition on each disk is defined by an attribute frequency distribution function and the distinct attribute size. We assume that the attribute frequency distribution functions are either uniform or normal. We also assume that the distribution for every disk is the same and every distinct attribute appears in every disk. So from the attribute distribution and the distinct attribute size we may easily derive the size of each attribute provided we correspond every real attribute to an abstract attribute: the ordinal number. For uniform distribution, every attribute gets the same number of tuples; for normal distribution, we assume that the middle attribute has the highest probability of appearing in the relation and then we need a variance parameter

to derive the probability of other attributes. If the attribute A with the smallest frequency pA has x numbers of tuples then any attribute B with frequency pB will have $\text{ceil}(pB/pA) \times x$ numbers of tuples. The relevant parameters for database modeling are as follows.

NumRelations: Number of relations in the database

AttributeSize_i: Number of distinct attributes in relation i

AttriFreqDist_i: The attribute frequency distribution function on each disk for relation i

RelationSize_i: Number of tuples in relation i (Note: we determine the range for any relation. But the concrete relation size is determined by its *AttributeSize* and *AttriFreqDist*.)

TupleSize: Bytes in a tuple (Bytes/tuple)

PageSize: Number of tuples per page (tuples/page)

3.1.3 System Model Assumptions

The following assumptions are made about the system.

- Each data item is stored at exactly one site (No replicated data).
- Relations are uniformly distributed among all the disks in the system.
- Each transaction consists of multiple processes that can be executed at multiple sites and whose workloads depend on the selected data.
- Sufficient buffer space exists to allow the retention of updates until commit time and a log-based recovery scheme is used so that only the log pages must be forced prior to commit.
- Two-phase locking is assumed to be the concurrency control method. There are several important aspects of the two-phase locking protocols as discussed below.

1. static vs. dynamic. In a static locking protocol locks are requested in advance for all the objects accessed by the transaction at the begin-transaction time; while in dynamic locking, locks are requested right before the objects are accessed. Our study assumes a dynamic strategy with all locks held until the end of the transaction.
 2. locking granularity. As in conventional systems, the granularity of the lockable units, e.g., block, page, or record, is an important performance factor in parallel database machines. A physical locking protocol at the page level is assumed in our experiments.
 3. lock modes. We consider only read and write lock modes.
- Two-phase commit protocol for reliability.

There are a number of different communication paradigms that can be employed in implementing a two-phase commit protocol. These are the centralized two-phase commit protocol, linear two-phase commit protocol and distributed two-phase commit protocol. We use the distributed two-phase commit protocol as the two-phase commit protocol in two-level hierarchical architecture because it eliminates the need for the second phase of the protocol since the participants can reach a decision on their own. Certainly, the distributed two-phase commit protocol incurs more message passing overhead than the centralized two-phase commit protocol. But fortunately, it is an intensive message passing process. Considering that the join operation also employs an intensive data distributing process, we are able to expect that the workload for the communication network could be much more uniform if both the intensive processes could be done intermittently.

- Transaction failure, transaction recovery are considered; no system failure and recovery, no media failure and recovery are considered.

3.2 Host Queueing Model

The queueing model for host is given in Figure 3.2. When a transaction is initiated by a user terminal, the transaction type, the accessed relations and pages are chosen by the transaction-generator. Then it waits in the global Transaction Ready Queue until the number of active transactions in the system fall below the degree of multiprogramming (Mpl). Mpl ranges from 1 to the number of processing modules. GTM is the center of the host queueing model.

3.2.1 Transaction-Generator

The transaction-generator is responsible for generating the workload for the host. There are two types of transactions: update-based and query-based. For each simulation run, the distribution for both types is determined. The selecting proportion decides whether the transaction is update-intensive or query-intensive etc. We assume that any involved relation for both transactions is uniformly selected among the total number of relations.

3.2.2 Global Transaction Manager (GTM)

Once a transaction enters the system, GTM loads a global coordinator for the transaction at the host node. For update transaction, GTM will split the transaction into cluster-related sub-transactions, which are transferred to the appropriate CTMs (Cluster Transaction Manager) to initiate the global cohorts. For the join transaction, GTM broadcasts the transaction to all the cluster CTMs to initiate its global cohorts. Then GTM will assign one global cohort as the global coordinator's subhost for coordination and will keep all the participation information such as the transaction id, subhost id, the sub-transaction workload and the participants' list etc. in the global coordinator and the log. Finally, GTM will send all the subtransactions, each of which includes the necessary information such as transaction id, subhost id and participants' list etc. to the appropriate clusters.

CCCM ---- Cluster Concurrency Control Manager

CCM ---- Cluster Cache Manager

Figure 3.2: Two-Level Hierarchical Architecture Queueing Model

When a global coordinator gets the message “Transaction-Done” from a CTM it writes a commit record in the log and informs the finishing of the transaction to the user terminal that initiated the transaction and will initiate another new transaction.

When a global coordinator receives the message “Transaction-Abort” from a CTM it writes an abort record in the log, delays for a period of time Delay-time and then moves into the Global Transaction Ready Queue with high priority for rerunning. However, the transaction workload pattern is preserved.

3.3 Cluster Queueing Model

The cluster model is given in Figure 3.2. It mainly consists of cluster transaction manager (CTM), cluster scheduler (CS) and cluster data manager (CDM) etc.

3.3.1 Cluster Transaction Manager (CTM)

When CTM receives a subtransaction (called cluster transaction) from the CTM Ready Queue it initiates a global cohort, which is actually a local cluster coordinator for the cluster transaction. The cluster transaction is split into a number of sub-transactions according to the transaction type and the accessed data items. Each sub-transaction corresponds to a cluster cohort. For update transaction, the transaction is split into the disk-related subtransactions. (Note that we only consider one subtransaction per disk.) For join transaction, each disk has a sub-transaction which is responsible for the split phase in ABJ^+ (Note that within a cluster, we assume that the matching attributes or tuples have been distributed to the corresponding disks after the split phase).

After a cluster coordinator receives all the “Work-Done” messages from its cluster cohorts it writes a ready record in its log and sends a “Vote-Commit” message back to all the involved participants.

Let us assume that a cluster coordinator has obtained all the “Vote-Commit” messages from all the participants including itself. If the transaction is update-based

it writes a commit record and informs all its cluster cohorts to commit. At the same time, if it is also a subhost, it will send a "Transaction-Done" message back to the global coordinator in the host. So the local cluster updated database is written onto the cache log and the states of the updated database in the cache are changed, as well as all locks are released. The actual database update is deferred until the buffer replacement policy needs to reclaim an updated buffer. If the transaction is join-based, the cluster coordinator writes a commit record, and informs all its cluster cohorts to report the sizes of its subbuckets and to release all the locks. Then the size information and a "Split-Done" message are sent back to the cluster coordinator. Finally, the cluster coordinator combines them and sends them with a "Global-Split-Done" message back to its subhost.

In the case of join-based transactions, when the subhost has received all the "Global-Split-Done" messages from all its global cohorts, it stores a Split-End record in its log and adds up the sizes of the matching subbuckets distributed across the disks to derive the sizes of the corresponding buckets. It then decides how many tuples each cluster will send and receive, and informs all the cluster coordinators (i.e., global cohorts) including itself to send and receive the required tuples. After receiving all the tuples each cluster coordinator calculates and tunes the number of join buckets according to the memory capacity (i.e., Bucket tuning phase). Each join bucket becomes a join unit of a new join cluster cohort which has no data conflicts with any other cluster cohorts. After all the new join cluster cohorts have finished their work they will release all the temporarily occupied area and send "Done" message back to their cluster coordinator which will send a "Transaction-Done" message back to the host if the cluster coordinator is also a subhost.

If a cluster coordinator receives any "Abort" message from its cluster cohorts it forces other involved cluster cohorts to abort, sends an "Vote-Abort" message to all other sibling participants and sends a "Transaction-Abort" message to its host. To abort an update-based transaction, all updated database belonging to the transaction are simply deleted in the cache and the related database states are changed. To abort a join-based transaction the work done is just ignored.

When a cluster coordinator receives any "Vote-Abort" message from other clusters it informs all its cluster cohorts to abort.

When a cluster cohort is chosen to abort in the case of deadlock by global deadlock detector the cluster coordinator writes an abort record in the log and sends an "Vote-Abort" message to all the involved cluster sites. Subhost is chosen by a Round-Robin strategy to deal with global deadlock with a centralized periodic deadlock detection algorithm originally proposed in [Agrawal 83, Chin 84]. It is responsible for collecting fragments of the cluster wait-for-graph from each cluster, for locating cycles and selecting a victim to abort. The protocol is shown in Algorithm 3.1.

The cycle detection mentioned in Algorithm 3.1 is done by the deadlock detection algorithm shown in Algorithm 3.2 that is based on the deadlock detection algorithm in [Desai 90, Page 598]. Whenever a deadlock is detected, CTM will abort one of the transactions with the minimum number of locks in the involved cycle.

Initialization:

Period is initialized to the lower bound at the system beginning and is put into all the initialized clusters.

Subhost designation is based on round-robin strategy. Cluster 1 is assigned to subhost at the beginning.

Begin

Every cluster site sends its local cluster wait-for-graph to the designated subhost in a Period time from last global detection;

The subhost creates a global wait-for-graph with the collected local wait-for-graph from each site;

Whenever cycles are detected in the global wait-for-graph, halves the period (at least the lower bound) and aborts the transactions holding the minimum number of locks to break the cycles;

If no global deadlock is found the Period is doubled (at most the upper bound).

End

Algorithm 3.1: Centralized Period-Adaptable Deadlock Detection Protocol

Input: An adjacent linked list representing wait-for-graph that contains: transaction IDs, the number of locks held by each transaction and all the transactions-waiting-for.

Data Structure Used: A Boolean variable Deadlock-Situation. A first-in, first-out stack, Transaction-stack, to hold transaction IDs: this stack will contain the transactions in a deadlocked chain if a deadlock is detected.

Output: Whether the system is deadlocked and if so, the transactions in the cycle.

Begin

```

Initialize Deadlock-Situation to false;
Initialize Transaction-stack to empty;
for next transaction in the adjacent linked list
    begin
        Push the next transaction ID into Transaction-stack;
        While Transaction-stack not empty
            if next transaction-waiting-for of the transaction
            on top of Transaction-stack exists
            then
                begin
                    T-next= next transaction-waiting-for
                    if T-next exists in wait-for-graph
                    then
                        if T-next transaction ID is in stack
                        then
                            Deadlock-Situation=True;
                            return(Deadlock-Situation)
                        else
                            Push T-next transaction ID
                            to Transaction-stack
                        end
                    end
                end
            else
                Pop Transaction-stack
            endWhile
        endfor
    return(Deadlock-Situation)
End

```

Algorithm 3.2: Deadlock Detection Algorithm

3.3.2 Cluster Scheduler (CS)

CS handles concurrency control requests from cluster cohorts, requests to force some cluster cohorts to abort and requests for cluster cohort to commit etc. CS also informs CTM with the messages such as “Work-Done”, “Split-Done”, “Done” and so on.

Cluster cohorts require read locks on items that they read, converting their read locks to write locks on items that need to be updated (Note: in an update transaction all reads happen before all writes.). As usual, read locks can be shared but write locks cannot. Locks are set dynamically, as the cluster cohort executes, with the cohort blocking until the next item to be accessed has been successfully locked. Locks are held until the transaction has been successfully committed or aborted. The blocked queue holds the cluster cohorts that are waiting for locks. Algorithm 3.3 gives the request granting algorithm.

Of course, local deadlocks are possible. Local cluster deadlock prevention applies whenever two cluster cohorts want to write on the shared lock page. The cohort with the later write-request will be aborted. If any cluster cohort aborts due to local cluster deadlock it sends an “Abort” message back to its cluster coordinator.

If a cluster cohort finishes its work it sends a “Work-Done”, “Split-Done” or “Done” message to its cluster coordinator.

Begin**Cases request of**

read operation (read request processing)

case requestPage of

. noLock: set read lock and granted;

readLock: granted (shared);

otherwise blocked and ungranted

endcase

write operation (write request processing)

case requestPage of

readLock, unshared and same transaction request:

upgrade it to writeLock and granted;

otherwise blocked and ungranted

endcase**EndCases****End**

Algorithm 3.3: Request Granting Algorithm

3.3.3 Cluster Data Manager (CDM)

CDM receives the operation of granted cluster cohort from CS and processes it by manipulating storage. The model is shown in Figure 3.2.

Any granted cluster cohort turns to cluster cache manager (CCM) which will direct the cluster cohorts to disk server for cache non-hit and to CPU servers for cache hit. We will adopt a no-redo/no-undo recovery strategy for update-based transactions in our model which is based upon [Elhardt 84]. The cache protocol is shown in Algorithm 3.4.

There is a single queue for all CPUs (shared-everything cluster) since the scheduler assigns a job to the next available CPU. There is a single queue for each disk because the data is disk-oriented.

Data Structures:

Possible cache states for each page:

(Original,Unfixed,Changed): Committed updated version

(Original,Unfixed,Unchanged): Clean version (Database version)

(Original,Fixed,Unchanged): Shared by transactions

(Copy,Fixed,Changed): Updating by transaction

Regular Queue: A sorted queue holding the pages with (Original, Fixed,Unchanged) and (Copy,Fixed,Changed)

Unfixed Queue: A LRU queue holding the pages with (Original, Unfixed,Unchanged) and (Original,Unfixed,Changed)

ReadRequest: to read a page p

If p not in Cache

 then turn to disk server to fetch p into Cache

If p in Cache

 then move p to Regular Queue with the state (Original, Fixed,Unchanged)

WriteRequest: to write a page p

If p not in Cache

 then turn to disk server to fetch p into Cache

If p in Cache then

 If p in Regular Queue

 then upgrade it to (Copy,Fixed,Changed)

 If p in Unfixed Queue then

 If p in (Original,Unfixed,Unchanged)

 then

 move p to Regular Queue with (Copy,Fixed,Changed)

 else

 If Cache full

 then

 Replace a page q in the front of Unfixed Queue

 If q in (Original,Unfixed,Unchanged)

 then just delete it

 else move q to disk server for
 updating database

 Put a copy of p into Regular Queue with
 (Copy,Fixed,Changed)

Algorithm 3.4: Cache Protocol

To process a page p from disk server

```
If Cache full
  then
    Replace a page q in the front of Unfixed Queue
    If q in (Original,Unfixed,Unchanged)
      then
        just delete it
      else
        move q to disk server for updating database
  If ReadRequest
    then Put p into Regular Queue with (Original,Fixed,Unchanged)
  If WriteRequest
    then Put p into Regular Queue with (Copy,Fixed,Changed)
```

To commit a transaction T

```
All corresponding copies of T are written to safe;
The all copies become (Original,Unfixed,Changed) and are put back
into the rear of Unfixed Queue;
For every page p with (Original,Fixed,Unchanged) acted by T
  Delete the actor T;
  If no more transactions acting on p
    then move p back to Unfixed Queue with (Original,
      Unfixed,Unchanged)
```

To abort a transaction T

```
All corresponding copies of T are simply deleted;
For every page p with (Original,Fixed,Unchanged) acted by T
  Delete the actor T;
  If no more transactions acting on p
    then move p back to Unfixed Queue with (Original,
      Unfixed,Unchanged)
```

Algorithm 3.4: Cache Protocol (Continued)

3.4 Network Queueing Model

The model of the global interconnection network is encapsulated by the Global Network Manager (GNM). In our network model, we take network communication bandwidth as the only parameter for measuring messages passing among host and clusters. The message-passing time within the network is calculated by the network bandwidth and the message size.

Chapter 4

Performance Experiment and Analysis

4.1 Simulation Parameter Settings

We assume that there are 128 terminals attached to the host and there are 8 processing modules in the system by default. Every processing module has one processor, one memory and one disk. We can cluster processing modules to obtain different architectures. For example, clustering two processing modules gets four clusters, each of which has two processing modules. Table 4.1 lists different possible architectures for 8 processing modules. We use the notations 8-8-1, 8-4-2, 8-2-4 and 8-1-8 to represent these architectures, respectively. 8-8-1 belongs to shared-nothing (SN). 8-4-2 and 8-2-4 belong to two-level hierarchical architectures (2HAs). And 8-1-8 belongs to shared-everything (SE). We will make use of simulation results to compare the performance among SN, 2HAs and SE.

In order to compare the performance of the different architectures, the parameter settings should be consistent. Table 4.2 lists default simulation parameter values used in the experiments.

	SN	2HA1	2HA2	SE
ClusterNumber	8	4	2	1
ClusterSize	1	2	4	8

Table 4.1: The different architectures for 8 processors

Number of terminals (TN)	128
Number of processors (processing module)	8
Number of clusters (CN)	8, 4, 2, 1
Number of processors within a cluster	1, 2, 4, 8
Tuple size	200 bytes
Page size	1000 bytes
Number of pages for each relation on each disk	100 pages
Number of relations	32
Access probability for each relation	1/32
Number of read operations for each update-based transaction (TS)	32
Write probability for update-based transactions	1/4
Rate of GTM, CTM, CS and Cache	infinite
Rate of CPU	1.0 MIPS
Network bandwidth	1.0 Mbytes/sec
Time for processing a R/W page	22.5 ms
Time for reading a join bucket	2.8 ms
Time for reading and writing a join page	5.6 ms
Instructions for processing a R/W page	2K
Instructions for processing a join page	4K
Instructions for processing a newjoin bucket	4K
Cache capacity of a cluster	TS/CN*TN
Multiprogramming level	1 .. processing module
Mean of think time distribution, Uniform	0..100 sec
Mean of delayed time distribution, Uniform	1 sec
Upper bound of deadlock detection	60 sec
Lower bound of deadlock detection	5 sec
Bytes required to send a simple message such as "Vote-Commit" etc.	32 bytes
Hashed database size information for join	400 bytes

Table 4.2: Default Simulation Parameter Settings

4.1.1 Database and Transaction Parameters

We assume that there are 32 relations, each of which has 100 pages on each disk. So the total database size in the system is $32 * 8 * 100 = 25,600$ pages. It is also assumed that every relation has the same number of attributes and every attribute has the same number of tuples except when static data skew effects are considered. Static data skew can be defined with the attribute normal distribution. The degree of static data skew depends on the variance of the normal distribution and the attribute size. In our simulation, we assume that the skewed attributes have more tuples than other, non-skewed, attributes and the number of tuples for the skewed and non-skewed attributes are uniformly distributed among all the disks.

The relation selection is uniform among all the 32 relations. The number of pages read by an update-based transaction is determined by the number of read operations for each update-based transaction. Each page belongs to a selected relation. The write probability decides the number of pages that will be updated in a transaction. For every join-based transaction two relations are selected. We assume the same data heat ¹ and data temperature ² among all the disks. So every update-based transaction can be created by Algorithm 4.5.

For join-based transactions, the dynamic data skew after hashing is important. We will adopt a definition of data skew degree that is similar to that in [Hua 91] for SN to investigate the impact on the performance among SN, SE and 2HAs.

In the case of SN, the following definitions are used.

T: Relation size in pages for the two join relations;

Ps: Size in pages of the skewed partition;

Pu: Size in pages of each of the remaining unskewed partitions;

N: Number of processing modules;

σ : The degree of data skew which is defined as

$$\sigma = \frac{|Ps - Pu|}{Ps}$$

¹As discussed in Section 2.3

²As discussed in Section 2.3

We also assume there is only one single skewed processing module. The remaining nodes are unskewed and have the same number of data pages. This assumption is based on the fact that the most seriously skewed processing module constitutes the bottleneck in SN. Then, we could have the following.

$$Ps + (N - 1) \times Pu = T \Rightarrow \begin{cases} Ps = \frac{T}{1+(N-1) \times (1-\sigma)} \\ Pu = \frac{(1-\sigma) \times T}{1+(N-1) \times (1-\sigma)} \end{cases}$$

Similarly in the case of 2HAs, Ps needs be redefined as follows because of the SE cluster. In SE, only skewed pages are amortized among all the processing modules within a cluster.

Ps : Size in pages of the skewed partition for each processing module within a skewed cluster.

Therefore,

$$Ps = \frac{|T - Pu \times (N - CS)|}{CS}$$

where, CS is the number of processing modules within a cluster.

In our simulator the bucket size is equal to the page size. So Ps and Pu define the number of join buckets for every processing module.

Output: A transaction with a sequence of operations.

Begin

 Initialize Disk-Order to 0;

for every new operation of a transaction

begin .

Do

 Relation-Name=Uniformly select a relation name
 among all the relations;

 Relation-Page=Select a page uniformly among all the
 pages of the selected Relation-Name in Disk-Order;

While (The Relation-Page of the Relation-Name
 exists in the transaction);

 Disk-Order= (Disk-Order+1) modulo the total number of disks;

 Determine the read and write or just read according
 to the write probability;

endfor

End

Algorithm 4.5: Update-Based Transaction Generation Algorithm

4.1.2 System Parameters

The processing rates for GTM, CTM, CS and Cache, which are common to all the architectures, can be infinitive without destroying the consistent behavior among the different architectures. GTM is same for any architecture. Although different architectures have different number of CTMs, CSes and Caches, the overall rates should be consistent. For example, 8-8-1 has eight CTMs and 8-4-2 has four CTMs. But the CTM for 8-4-2 should be twice faster than that for 8-8-1. Therefore, we assume all the rates for GTM, CTM, CS and Cache are infinitive for simplicity.

We also assume that the disk time to read or write a page for the update-based transactions is 22.5 ms, the disk time for reading a join bucket is 2.8 ms and the disk time for reading and writing a join page is $2 * 2.8 = 5.6$ ms. The calculation of disk time is based on disk's seek time, rotational delay and block transfer time. For each page access of update-based transactions, all three factors should be considered

because of the randomness. But, for the join-based transactions, the seek time and rotational delay can be minimized if the pages or buckets are consecutively stored on the permanent or temporary disk area and the bulk of data pages are prefetched into the cache awaiting processing.

4.2 Performance Validation

The following performance indices are used in the simulation experiments. Response time of a transaction is defined as the difference between its completion time and its initiated time by a user terminal. Throughput is defined as the number of completed transactions per unit time. Abort ratio is defined as the average number of aborts induced to complete a transaction. Disk, CPU and network utilization define the fraction of busy time. The disk or CPU utilization shown in the following figures is always referred to the maximum utilization among disks or CPUs.

There are two methods to validate our simulator. One way is to use system work index. For a given transaction, the simulation results are consistent with the static execution costs.

Another way is to use similar parameter values used in [Carey 89] for simulation, where the difference from our default values is $Mpl = 8$ and 300 pages/disk for 8-8-1, and compare the results. We will also set $Mpl = 1$ and 2400 pages/disk for 1-1-1 in order to maintain the consistent parallelism and database size. The simulation results with 1-1-1 and 8-8-1 for all update-based transactions are shown in Figure 4.1 through Figure 4.6. These results are consistent with the results reported in [Carey 89].

In the following sections we will set $Mpl = 4$ by default since the simulation results have shown the performance becomes stable and acceptable at this level for all the architectures with 8 processing modules.

Figure 4.1: Response Time of 1-1-1, 8-8-1 for All-Updates

Figure 4.2: Throughput of 1-1-1, 8-8-1 for All-Updates

Figure 4.3: Abort Ratio of 1-1-1, 8-8-1 for All-Updates

Figure 4.4: Disk Utilization of 1-1-1, 8-8-1 for All-Updates

Figure 4.5: CPU Utilization of 1-1-1, 8-8-1 for All-Updates

Figure 4.6: Network Utilization of 1-1-1, 8-8-1 for All-Updates

4.3 Performance with All Update-based Transactions

Figure 4.7 through Figure 4.12 show the performance of 8-8-1, 8-4-2, 8-2-4 and 8-1-8 architectures for all update-based transactions. Some observations are as follows.

From Figure 4.7 it can be seen that the performance of 8-2-4 is very close to that of 8-1-8 architecture. And even the performance of 8-4-2 is better than that of 8-8-1. This shows that the hierarchical architectures improves performance over the shared-nothing architecture.

From Figure 4.8 we can see that all the throughputs increase initially due to the decrease in abort ratio and then decrease due to the reduced system load. The increase in initial throughput for 8-8-1 is higher than that associated with the other systems. This is because 8-8-1 induces higher transaction aborts as shown in Figure 4.9.

The disk and CPU utilizations follow the throughput as shown in Figure 4.10 and 4.11, respectively.

The network utilization in Figure 4.12 is also similar to the throughput performance except in the cases of 8-2-4 and 8-1-8; this is because the amount of communication messages is negligible for 8-2-4 and 8-1-8 systems.

From the data presented here, it may be noted that 8-1-8 outperforms 8-2-4, which outperforms 8-4-2, which in turn outperforms 8-8-1. These differences are evident only in the lower think time range, where the system load is sufficient to generate a significant amount of data contention. The results imply that clustering of more processing modules makes better use of the processing modules and improves system performance.

Figure 4.7: Response Time of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All-Updates

Figure 4.8: Throughput of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All-Updates

Figure 4.9: Abort Ratio of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All-Updates

Figure 4.10: Disk Utilization of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All-Updates

Figure 4.11: CPU Utilization of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All-Updates

Figure 4.12: Network Utilization of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All-Updates

4.4 Performance with All Join-based Transactions

Results for all join-based transactions are presented in Figure 4.13 through Figure 4.17. These results are for $\sigma = 0.9$ and $Mpl = 4$. Abort ratio is always zero because all-joins involves no lock conflicts. Actually, changing the data skew σ values does not affect the performance of 8-1-8 (already verified by simulation).

The flat throughput indicates that the system is in a stable processing state as long as there are enough transactions waiting for processing regardless of the increase of think time. And the response time in Figure 4.13 decreases linearly by the amount of think time as the think time increases. If we continue to increase the think time of transactions until there are not enough transactions waiting, the response time and throughput will decrease significantly. Also note that the performance of 8-2-4 (2HA) is similar to that of the 8-1-8 (SE) architecture. This implies that 2HAs can be designed to provide performance very close to that provided by SE architecture.

Join transaction, which involves substantial system resources, significantly affect the utilization of disks and CPUs. There are two factors that affect the utilization of disks and CPUs. One is the architecture factor. Because 2HAs have the clustered processing modules the workload is well-balanced within a cluster, which speeds up transaction processing. The other factor is the data skew. Data skew destroys the workload balance among the clusters because skewed cluster gets more workload than the non-skewed clusters. In 2HAs, the data skew can be shared among the processing modules in a cluster so that the effect of data skew is reduced. The simulation results presented here support the above observations.

4.5 Performance Sensitivity

We assume a mixed transaction workload with 20% update-based transactions and 80% join-based transactions in the following simulation runs. This section investigates the impact of mixed transactions, data-skew, system size and database size on

Figure 4.13: Response Time of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All Joins

Figure 4.14: Throughput of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All Joins

Figure 4.15: Disk Utilization of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All Joins

Figure 4.16: CPU Utilization of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All Joins

Figure 4.17: Network Utilization of 8-8-1, 8-4-2, 8-2-4, 8-1-8 for All Joins

the performance of SN, 2HAs and SE.

4.5.1 Sensitivity to Mixed Transactions

In the mixed transactions, there are 80% join transactions which dominate the system execution time. So the performance is similar to that for all join-based transactions. Although there are 20% update-based transactions, which can cause abortion, the abort ratio is less than that for all update-based transactions. Therefore, for all architectures, the response time and throughput for mixed transactions are always better than those for all join-based transactions. Comparing Figures 4.18 and 4.19 with Figures 4.13 and 4.14 confirms the above observation. Another observation from these figures is that the reduction in response time for SN is more than that for 2HAs and SE, which implies that SN is more sensitive to mixed transactions than 2HAs and SE.

If we change the proportion in the mixed workload, we predict that the performance of any architecture is between that for all-updates and that for all-joins of

Figure 4.18: Response Time of 8-8-1, 8-4-2, 8-2-4 and 8-1-8 for Mixed Transactions

the corresponding architecture.

4.5.2 Sensitivity to Data Skew

This section will investigate the data skew impact on the performance of 8-8-1, 8-4-2, 8-2-4 and 8-1-8 for the mixed transactions workload with $Mpl = 4$ and $Thinktime = 0$. Figure 4.20 shows the response time performance and the throughput is shown in Figure 4.21. It can be seen from these figures that 8-8-1 is the most sensitive to the data skew. The response time at $\sigma = 0.5$ is 18% more than that at $\sigma = 0.1$ for 8-8-1. And the response time at $\sigma = 0.9$ is 35% more than that at $\sigma = 0.5$ for 8-8-1; the response time at $\sigma = 0.5$ is 7.79% more than that at $\sigma = 0.1$ for 8-4-2. And the response time at $\sigma = 0.9$ is 19% more than that at $\sigma = 0.5$ for 8-4-2; the response time at $\sigma = 0.5$ is only 1.29% more than that at $\sigma = 0.1$ for 8-2-4. And the response time at $\sigma = 0.9$ is only 13% more than that at $\sigma = 0.5$ for 8-2-4. Obviously, the performance for 8-2-4 and 8-4-2 are closer to the performance for 8-1-8 than that for 8-8-1, which indicates that the data skew has the least influence on 8-2-4, followed

Figure 4.19: Throughput of 8-8-1, 8-4-2, 8-2-4 and 8-1-8 for Mixed Transactions

by 8-4-2, and followed by 8-8-1.

4.5.3 Sensitivity to System Size

The impact of system size on the performance of different architectures is considered in this section. We have run several simulations for 4-4-1, 4-2-2 and 4-1-4 with $Mpl = 2$, as well as 16-16-1, 16-8-2, 16-2-8 and 16-1-16 with $Mpl = 8$ maintaining the same database size. The simulation results for response time and throughput are shown in Figures 4.22 and 4.23, respectively.

Because the database size is kept the same for all system sizes, maintaining the data skew $\sigma = 90\%$ will cause less skewed partitions for larger systems than those for smaller systems. Thus, the data skew effect will be smaller with increasing system size, which explains why the curves for SN, 2HAs and SE approach together with increasing number of processing modules.

Comparing the performance difference at 4 processing modules and at 8 processing modules, it can be seen that the response time reduces by 58% and the

Figure 4.20: Data Skew Effect on Response Time of 8-8-1, 8-4-2, 8-2-4 and 8-1-8 for Mixed Transactions

Figure 4.21: Data Skew Effect on Throughput of 8-8-1, 8-4-2, 8-2-4 and 8-1-8 for Mixed Transactions

Figure 4.22: Impact of Number of Processing Modules on Response Time for Mixed Transactions

throughput increases by 138% for 8-8-1; and the corresponding values for 8-1-8 are 52% and 107% for 8-1-8. Comparing the performance at 8 processing modules and at 16 processing modules we could see that, for 8-8-1, the response time reduces by 55% and the throughput increases by 123%; and the corresponding values for 8-1-8 are 37% and 60%. The performance of 8-4-2 and 8-2-4 is always between that of 8-8-1 and that of 8-1-8. The data shows that the decreasing rate of response time for SN is more than those for 2HAs and SE, which indicates that SN is more sensitive to system size than 2HAs and SE.

4.5.4 Sensitivity to the Database Size

Due to the limitation of our simulation resources, the system size could not be too large. So in our default settings, the number of pages for each relation on each disk is set to 100. This section studies how the database size affects the performance of the different architectures. Figures 4.24 and 4.25 show results of

Figure 4.23: Impact of Number of Processing Modules on Throughput for Mixed Transactions

response time and throughput, respectively. These results indicate that the increase rate of the response time for 8-2-4 and 8-4-2 is always smaller than that for 8-8-1 within the range of the database sizes considered. From the results, we predict that the performance of 2HAs will be always better than that of SN regardless of the database size. In other words, SN is more sensitive to database size than 2HAs.

4.6 Summary

This chapter has studied the performance of shared-nothing (SN), two-level hierarchical architectures (2HAs) and shared-everything (SE) for different transaction workloads such as all-updates, all-joins and 20% : 80% mixed transactions.

Section 4.1 presented the assumptions and the default parameter values used in the simulation experiments. In Section 4.2 the results of the simulation experiments are validated by comparing with those presented in a previous study by Carey et al. [Carey 89]. Sections 4.3 and 4.4 presented the performance of 8-8-1, 8-4-2, 8-

Figure 4.24: Impact of database Size on Response Time of 8-8-1, 8-4-2, 8-2-4 and 8-1-8 for Mixed Transactions

Figure 4.25: Impact of database Size on Throughput of 8-8-1, 8-4-2, 8-2-4 and 8-1-8 for Mixed Transactions

2-4 and 8-1-8 architectures for all update-based transactions and for all join-based transactions, respectively. These results indicate that the performance of 2HAs is closer to that of SE than that of SN, which suggests that 2HAs can improve system performance over SN while providing system scalability.

The performance sensitivity to mixed workload, data skew, system size and database size is given in Section 4.5. The results in Section 4.5.1 indicate that SN is more sensitive to mixed transactions than 2HAs and SE; Section 4.5.2 shows that SN is much more sensitive to data skew degree than all the other architectures. The impact of system size and database size on the performance of SN, 2HAs and SE is considered in Section 4.5.3 and 4.5.4. The results indicate that SN is more sensitive to the system size and the database size than 2HAs and SE.

From the simulation results, it can be concluded that 2HAs always provide better performance than SN. Also, a properly designed 2HA can provide performance very close to that provided by SE. Thus, a 2HA can be designed to provide good performance (similar to that of SE) while improving the system scalability (similar to that of SN).

Chapter 5

Conclusions

5.1 Summary

The goal of this thesis is to specify and design a parallel hierarchical database system and compare the performance of this architecture with that of shared-everything and shared-nothing architectures. Shared-everything provides better performance than shared-nothing, but it is not scalable to large system size. On the other hand, shared-nothing architecture provides good system scalability. The results from the simulation experiments presented here have shown that a properly designed hierarchical system can provide performance very close to that of the shared-everything while providing system scalability similar to that provided by the shared-nothing architecture. The results indicate that, when designing a hierarchical system, the cluster size should be maintained as large as the hardware limitation would allow. Then, the performance of the hierarchical system would be very similar to that of shared-everything under the system and workload models considered here

Chapter 1 has presented a two-level hierarchical architecture and the motivation for considering the hierarchical architecture. A brief discussion of the advantages and disadvantages of the shared-nothing and shared-everything is also included in this chapter.

Chapter 2 has discussed several important concepts of parallel database systems. These concepts include the multiprogramming level, thrashing, two-phase locking,

deadlock, inter- and intra-parallelism, data heat and temperature, transaction failure and two-phase commit protocol etc.

Chapter 3 has put forward a detailed two-level hierarchical architecture system model and a two-level hierarchical architecture queueing model which includes a global transaction manager, cluster transaction managers, cluster schedulers, cluster data managers and a global network manager. This chapter has also introduced the transaction workload model, database model and the relevant algorithms and protocols for the queueing model.

A detailed performance comparison among shared-nothing, two-level hierarchical architectures and shared-everything has been provided in Chapter 4. From the simulation results, we can see that the response time and throughput performance of the two-level hierarchical architectures are always better than that of shared-nothing for the different transaction workloads such as all-updates, all-joins and mixed transactions. The results have also shown that the two-level hierarchical architectures are less sensitive to mixed workloads, data-skew, system size and database size than shared-nothing.

5.2 Contributions of the Thesis

This section summarizes the contributions of the thesis to the area of parallel database architectures.

Most of the previous studies concentrated on the performance of shared-nothing, shared-disk and shared-everything under the update-based transaction workloads. Although a single join workload is considered in two-level hierarchical architecture a more general multiple-join workloads have not yet been considered in the literature. And no comprehensive performance evaluation of the hierarchical architecture has been reported. Even for the shared-nothing and shared-everything architectures, simulation experiments have been done only for update-based transaction workloads. This thesis has specified, designed and simulated a parallel database system which can not only be tailored to model two-level hierarchical architectures but can also be

used to model shared-nothing and shared-everything architectures. The multiple-architecture simulator is convenient for doing research to compare the performance of various architectures such as two-level hierarchical architectures, shared-nothing and shared-everything. A very detailed performance evaluation of a fixed-size two-level hierarchical architectures, shared-nothing and shared-everything has been done under a set of reasonable assumptions. The material presented in Chapters 3, 4 and Appendix A are the contributions of this thesis.

5.3 Directions for Further Research

Because of the time limit, we could not complete all possible simulation experiments. But the existing simulator can provide a good test bed for any later simulation experiments. In the following a few research issues are identified.

The simulation results presented here are obtained under certain assumptions. Although these assumptions are reasonable, more simulation experiments are needed to test the impact of some of these assumptions. As mentioned in Section 2.1.4, data contention is a function of multiprogramming level (multiprogramming level), database size, granule size, transaction size and transaction service time etc. We have studied the performance influence of multiprogramming level and database size. A natural question that arises is how the granule size, transaction size and transaction service time affect the performance of the different architectures?

We have considered the data skew effect for the join-based and mixed transactions. If transactions access pages with hot data and the frequently accessed pages are uniformly distributed among all processing modules, it would be interesting to see the impact of this on the performance of these architectures? If the data skew is also hot, how will different architectures perform?

Even for studying the performance sensitivity of multiprogramming level, more experiments could be done. For a given architecture, with the fixed database size, granule size and transaction size, how will multiprogramming level affect the per-

formance and when will thrashing occur? For a dynamic workload, how will multi-programming level adapt to achieve the maximum performance?

In the future, we could explore the impact of different concurrency control methods on the performance of these architectures. We could study how various hash-based join algorithms will influence the behavior of the architectures. We could also study the performance of higher-level hierarchical architectures.

Appendix A

Performance Model Design

Based on the two-level hierarchical architecture queueing model shown in Figure 3.2 this appendix will list its detailed performance model design. Several definitions and assumptions are given in the following.

- Servers: GTM, CTMs, CSes, CDMs, DISKs, CPUs and GNM.
- Functional modules: GTM-PROCESSING, CTM-PROCESSING, CS-PROCESSING, CDM-PROCESSING, DISK-PROCESSING, CPU-PROCESSING and GNM-PROCESSING.
- Assumptions for server queues:
 - Message Queue is a priority queue; All other queues conform to FCFS strategy (First-Come-First-Serve);
 - No preemptive strategy is considered;
 - Among all the queues for a server, Message Queue has the highest priority and the priority decreases from left to right written order in the definitions of queues for servers.
- Queues for GTM: Message Queue, Delayed Transaction Ready Queue and Global Transaction Ready Queue.
- Queues for CTM: Message Queue, CTM Ready Queue.

- Queues for CS: Message Queue, Cluster Cohort Priority Queue and Cluster Cohort Ready Queue.
- Queues for CDM: Message Queue, CDM Priority Ready Queue and CDM Ready Queue.
- Queues for DISK: Disk Queue.
- Queues for CPU: CPU Queue.
- Queues for GNM: GNM Message Queue.
- Status of servers: Every server is either Busy or Idle.

When a server runs its functional module, it is in Busy state;

GTM will be Busy if (there is at least one message in its Message Queue) OR ((the number of active transactions are less than the assigned multiprogramming level) AND (there is at least one transaction in Delayed Transaction Ready Queue or in Global Transaction Ready Queue));

Each CTM will be Busy if (there is at least one message in its Message Queue) OR (there is at least one cluster transaction in CTM Ready Queue);

Each CS will be Busy if (there is at least one message in its Message Queue) OR (there is at least one cluster cohort in Cluster Cohort Priority Ready Queue or Cluster Cohort Ready Queue);

Each CDM will be Busy if (there is at least one message in its Message Queue) OR (there is at least one cluster cohort in CDM Priority Ready Queue or CDM Ready Queue);

Each DISK will be Busy if there is at least one cohort in its DISK Queue;

Each CPU will be Busy if there is at least one cohort in the shared CPU Queue;

GNM will be Busy if there is at least one message in its GNM Message Queue.

A.1 Host Simulation Design

A.1.1 Initialization Module

1. Initialize every server to the Idle state;
2. Every terminal initiates a transaction;
3. Insert every transaction into Global Transaction Ready Queue;
4. Create a Global-Preprocessing-End event.

A.1.2 GTM-PROCESSING Module

If GTM Message Queue is not empty

- Then
 1. Pick up a message from the GTM Message Queue;
 2. Case message of
 - “Transaction-Done” message: Create a Transaction-Done-End event.
 - “Transaction-Abort” message: Create a Transaction-Abort-End event.EndCase
- Else if (the number of active transactions $< Mpl$) AND ((there is a delayed transaction in Delayed Transaction Ready Queue) OR (there is a transaction in the Global Transaction Ready Queue))
 - Then if (there is a transaction in the Delayed Transaction Ready Queue)
 - * Then
 1. Load a delayed transaction from the Delayed Transaction Ready Queue;
 2. Create the corresponding Cluster-Trans-Arrival events for all the relevant clusters;

3. Increase the number of active transactions by one;
 4. Call GTM-PROCESSING module.
- * Else create a Global-Preprocessing-End event.
- Else GTM just idles.

A.1.3 Events for Host

1. Global-Preprocessing-End event

- Load a transaction from the head of the Global Transaction Ready Queue;
- Load a global coordinator for it into Global-Coordinator-Pool;
- If update-based transaction, split it into cluster-based subtransactions;
- If join-based transaction, initiate a join subtransaction for each cluster;
- Assign a subhost for the transaction;
- Keep all the information such as transaction id, transaction type, subhost id, workload data, participants list etc. into the global coordinator;
- Distribute all the subtransactions to the relevant clusters (Create the Cluster-Trans-Arrival events for all the relevant clusters);
- Increase the number of active transactions by one;
- Call GTM-PROCESSING module.

2. Transaction-Done-from-CTM event

- Insert the “Transaction-Done” message into GTM Message Queue;
- If GTM idle, then call GTM-PROCESSING module.

3. Transaction-Abort-from-CTM event

- Put the “Transaction-Abort” message into GTM Message Queue;
- If GTM idle, then call GTM-PROCESSING module.

4. Transaction-Done-End event

- Find the corresponding global coordinator from the Global-Coordinator-Pool;
- Create a Thinking event for the terminal that initiated the global transaction;
- Destroy the global coordinator;
- Decrease the number of active transactions by one;
- Call GTM-PROCESSING module.

5. Transaction-Abort-End event

- Find the corresponding global coordinator from Global-Coordinator-Pool for the aborted transaction;
- Move the global coordinator to the Delayed-Pool-for-Restarting (Create a Delayed-Trans event);
- Decrease the number of active transactions by one;
- Call GTM-PROCESSING module.

6. Delayed-Trans event

- Move the corresponding transaction from the Delayed-Pool-for-Restarting to the Delayed Transaction Ready Queue;
- If GTM idle, then call GTM-PROCESSING module.

7. Thinking event

- Call Transaction Generator to initiate a new transaction;
- Insert it into the Global Transaction Ready Queue;
- If GTM idle, then call GTM-PROCESSING module.

A.1.4 GNM-PROCESSING Module

If (there exists a message in the head of GNM Message Queue)

- Then pick up the message and create a corresponding event for the message (the event time is determined by the message volume)
- Else GNM just idles.

A.2 Cluster Simulation Design

A.2.1 CTM-PROCESSING Module

If CTM Message Queue is not empty

1. Then

- (a) Pick up a message from CTM Message Queue;
- (b) Case message of
 - “Work-Done” message
 - Find the corresponding cluster coordinator in the Cluster-Coordinator-Pool;
 - Register the “Work-Done” message of the cluster cohort;
 - If all the cluster cohorts have reported work done then create Vote-Commit events for the involved clusters via GNM;
 - Call CTM-PROCESSING module.
 - “Abort” message
 - Find the corresponding cluster coordinator in the Cluster-Coordinator-Pool;
 - Inform all the involved cluster cohorts to abort (Insert “Abort” message into CS Message Queue);
 - Create Vote-Abort events for all the involved clusters via GNM;

- Create Transaction-Abort-from-CTM event for GTM;
 - Remove the cluster coordinator;
 - Call CTM-PROCESSING module;
 - If CS idle, then call CS-PROCESSING module.
- “Vote-Commit” message
 - Find the involved cluster coordinator in the Cluster-Coordinator-Pool;
 - Register the “Vote-Commit” message;
 - If the cluster coordinator has obtained all the “Vote-Commit” messages from all the participants, then
 - i. Write a commit record in the log;
 - ii. Inform all the cluster cohorts in the Join-Pool-for-Cluster-Cohorts of CS to commit (Insert “Commit” message into CS Message Queue);
 - iii. If update-based transaction
 - * Then
 - A. If the cluster coordinator is also a subhost, then create a Transaction-Done-from-CTM event;
 - B. Remove the cluster coordinator;
 - C. Call CTM-PROCESSING module.
 - * Else (join-based transaction)
 - A. Create a ClusterSize-Processing-End event.
 - If CS idle, then call CS-PROCESSING module;
- “Vote-Abort” message
 - Find out the Cluster Coordinator in the Cluster-Coordinator-Pool;
 - Inform all the cluster cohorts to abort (Insert “Abort” message into CS Message Queue);

- If the Cluster Coordinator is also a subhost, then create a Transaction-Abort-from-CTM event;
- Remove the Cluster Coordinator;
- If CS idle, then call CS-PROCESSING module;
- Call CTM-PROCESSING module.
- “Split-Done” message (with size information from a cluster cohort in CS)
 - Find the corresponding cluster coordinator;
 - Register the “Split-Done” message and the size information of the cluster cohort;
 - If the cluster coordinator has received all the “Split-Done” messages then create Vote-Commit events for the involved clusters via GNM;
 - Call CTM-PROCESSING module.
- “ClusterSize-Processing-Done” message
 - Create a Global-Split-Done-End event for the subhost (Send the combined size information back to its subhost via GNM);
 - Call CTM-PROCESSING module.
- “Global-Split-Done” message (from other cluster coordinators)
 - Find the subhost in Cluster-Coordinator-Pool;
 - Register the “Global-Split-Done” messages;
 - The subhost adds up the sizes of the matching subbuckets;
 - If the subhost has received all the “Global-Split-Done” messages from all its participants
 - * Then derive the sizes of the corresponding buckets and decide how many tuples each cluster will send and receive (Create a GlobalSize-Processing-End event for the subhost);
 - * Else call CTM-PROCESSING module.

- “GlobalSize-Processing-Done” message (for the subhost)
 - Find the subhost;
 - Inform all the involved participants with a “Start-Distributing” message and the distribution information (Create all the Start-Distributing events via GNM);
 - Call CTM-PROCESSING module of the subhost.
- “Start-Distributing” message (with the distribution information from subhost)
 - Find the corresponding cluster coordinator;
 - Distribute tuples according to distribution information;
 - Using the distribution information create Distribution-Done-End events via GNM;
 - Call CTM-PROCESSING module.
- “Distribution-Done” message
 - Find the corresponding cluster coordinator in the Cluster-Coordinator-Pool;
 - Register the “Distribution-Done” messages;
 - If the cluster coordinator has received all the “Distribution-Done” messages from the involved clusters
 - * Then create a NewJoin-Cluster-Cohorts-End event;
 - * Else call CTM-PROCESSING module.
- “Done” message (from a cluster cohort in CS, for new-join cluster cohorts only)
 - Find the corresponding cluster coordinator;
 - Register the “Done” message of the cluster cohort;
 - If (the cluster coordinator has received all the “Done” messages from its cluster cohorts) then
 - i. If (it is also a subhost) then send a “Transaction-Done” message back to its host (Create a Transaction-Done-from-CTM

- event);
- ii. Remove the cluster coordinator.
 - Call CTM-PROCESSING module.
- “Internal-Clock-Interrupt” message
 - Send a “Check-CS-Ready” message to CS Message Queue;
 - If CS idle, then call CS-PROCESSING module.
- “Wait-for-Graph” message (for subhost only)
 - Register the “Wait-for-Graph” message;
 - If the subhost has received all the “Wait-for-Graph” messages
 - * Then combine them and detect deadlock (Create a Detection-End event).
- “Detection-Done” message
 - If a deadlock occurs
 - * Then Create Resolution-End event
 - * Else
 - i. Send a “No-Deadlock” message to all the involved clusters (Create No-Deadlock events for all the involved clusters via GNM);
 - ii. Double the Detection-Period (there is an upper bound);
 - iii. Set the next deadlock detection subhost;
 - iv. Call CTM-PROCESSING module.
- “Resolution-Done” message
 - Select victims to abort with min-lock strategy;
 - Send “Force-Abort” messages to all the involved clusters (Create all the Force-Abort-End events via GNM);
 - Halve the Detection-Period (there is a lower bound);
 - Set the next deadlock detection subhost;

- Send “Transaction-Abort” to the global coordinator in the host
(Create a Transaction-Abort-from-CTM event);
 - Call CTM-PROCESSING module;
 - Call CS-PROCESSING module.
 - “Force-Abort” message (from the subhost)
 - Create an Internal-Clock-With-Detection-Period event;
 - Inform all the involved cluster cohorts to abort (Send “Abort” message to CS Message Queue);
 - Remove the cluster coordinator;
 - Call CTM-PROCESSING module;
 - Call CS-PROCESSING module.
 - “No-Deadlock” message (from deadlock detection subhost)
 - Create an Internal-Clock-With-Detection-Period event;
 - Call CTM-PROCESSING module;
 - Call CS-PROCESSING module.
- EndCase

2. Else

- If the CTM Ready Queue is not empty
 - Then create a Cluster-Preprocessing-End event for the first transaction in the queue
 - Else CTM just idles.

A.2.2 Events for Cluster Transaction Manager

1. Cluster-Trans-Arrival event

- If a transaction belongs to Cluster *i*, then insert the transaction into the CTM Ready Queue in Cluster *i*;

- If Cluster *i* is idle, then call CTM-PROCESSING module.

2. Cluster-Preprocessing-End event

- Load a cluster transaction from the CTM Ready Queue;
- Load a global cohort (i.e., cluster coordinator) for the cluster transaction in Cluster-Coordinator-Pool, keep transaction id, subhost id, participants' list etc. in it;
- Split the workload of cluster transaction into the workload of the disk-based subtransactions (One subtransaction per disk);
- For every subtransaction, load a cluster cohort and keep the necessary subtransaction information such as workload, cluster coordinator's id etc. in the cluster cohort;
- Send cluster cohorts to the Cluster Cohort Ready Queue of CS for concurrency control execution;
- Call CTM-PROCESSING module;
- If CS idle, then call CS-PROCESSING module.

3. Vote-Commit event

- Insert the "Vote-Commit" message into the CTM Message Queue;
- If CTM idle, then call CTM-PROCESSING module.

4. Vote-Abort event

- Insert the "Vote-Abort" message into the CTM Message Queue;
- If CTM idle, then call CTM-PROCESSING module.

5. ClusterSize-Processing-End event

- Insert the “ClusterSize-Processing-Done” message into the CTM Message Queue;
- If CTM idle, then call CTM-PROCESSING module.

6. Global-Split-Done-End event

- Insert the “Global-Split-Done” message into the CTM Message Queue;
- If CTM idle, then call CTM-PROCESSING module.

7. GlobalSize-Processing-End event

- Insert the “GlobalSize-Processing-Done” message into the CTM Message Queue;
- If CTM idle, then call CTM-PROCESSING module.

8. Start-Distributing event

- Insert the “Start-Distributing” message into CTM Message Queue;
- If CTM idle, then call CTM-PROCESSING module.

9. Distribution-Done-End event

- Insert the “Distribution-Done” message into CTM Message Queue;
- If CTM idle, then call CTM-PROCESSING module.

10. NewJoin-Cluster-Cohorts-End event

- Calculate the number of join buckets based on the required capacity;
- Load new-join cluster cohorts, the number of which is equal to Cluster Size, and distribute the workload among the new-join cluster cohorts as uniformly as possible;

- Send the new-join cluster cohorts to the Cluster Cohorts Ready Queue of CS for execution;
- If CS idle, then call CS-PROCESSING module;
- Call CTM-PROCESSING module.

11. Internal-Clock-With-Detection-Period event

- Insert the “Internal-Clock-Interrupt” message into the head of the CTM Message Queue;
- If CTM idle, then call CTM-PROCESSING module

12. Send-Wait-For-Graph-End event

- Insert the “Wait-for-Graph” message into the head of CTM Message Queue;
- Call CTM-PROCESSING module.

13. Detection-End event

- Insert the “Detection-Done” message into the head of CTM Message Queue;
- Call CTM-PROCESSING module.

14. Resolution-End event

- Insert the “Resolution-Done” message into the head of CTM Message Queue;
- Call CTM-PROCESSING module.

15. Force-Abort-End event

- Insert the "Force-Abort" message into the head of CTM Message Queue;
- Call CTM-PROCESSING module.

16. No-Deadlock event

- Insert the "No-Deadlock" message into the head of CTM Message Queue;
- Call CTM-PROCESSING module.

A.2.3 CS-PROCESSING Module

If Message Queue for CS is not empty

1. Then

- (a) Pick up a message from CS Message Queue;
- (b) Case message of
 - "Commit" message: Create CS-Commit-Processing-End event.
 - "Abort" message: Create CS-Abort-Processing-End event.
 - "Check-CS-Ready" message: Create a Send-Wait-For-Graph-End event via GNM for the CTM of the assigned subhost.
 - "Force-Abort" message: Create CS-Force-Abort-Processing-End event.

EndCase

2. Else (Note: every cluster cohort has an index that refers to the first operation for update-based and refers to the first page for join-based.)

If (Cluster Cohort Priority Ready Queue not empty) OR (Cluster Cohort Ready Queue not empty)

- Then

- (a) If Cluster Cohort Priority Ready Queue not empty
 - Then load a cluster cohort from the Cluster Cohort Priority Ready Queue
 - Else load a cluster cohort from the Cluster Cohort Ready Queue.
- (b) If update-based or join-based cluster cohort
 - Then
 - If the cluster cohort has finished all the work
 - * Then
 - i. If update-based then send a “Work-Done” message to the cluster coordinator (Insert the “Work-Done” message into CTM Message Queue);
 - ii. If join-based then send a “Split-Done” message to the cluster coordinator (Insert the “Split-Done” message into CTM Message Queue);
 - iii. Make the cluster cohort enter Join-Pool-for-Cluster-Cohorts;
 - iv. Call CS-PROCESSING module;
 - v. If CTM idle, then call CTM-PROCESSING module.
 - * Else
 - i. If update-based then set request to the operation at the index;
 - ii. If join-based then set request to the page at the index;
 - iii. Increase index by one;
 - iv. Create a Request-Processing-End event (Check Granted or Ungranted).
 - Else
 - If the cluster cohort has finished the work (the case of a new-join cluster cohort)
 - * Then

- i. Send a “Done” message to the cluster coordinator (insert the “Done” message into CTM Message Queue);
 - ii. Ask for destroying.
 - iii. Call CS-PROCESSING module.
 - iv. If CTM idle, then call CTM-PROCESSING module.
- * Else
 - i. Send it to CDM (Insert it into CDM Ready Queue);
 - ii. Call CS-PROCESSING module;
 - iii. If CDM idle, then call CDM-PROCESSING module.
- Else CS just idles.

A.2.4 Events for Cluster Scheduler

1. Request-Processing-End event

If Granted (returned from RGA(Request)) Then

- Send the cluster cohort to CDM Ready Queue;
- If CDM idle, then call CDM-PROCESSING module;
- Call CS-PROCESSING module.

If Aborted Then

- Insert the “Abort” message to CTM Message Queue;
- Call CS-PROCESSING module;
- If CTM idle, then call CTM-PROCESSING module.

If Ungranted Then

- Send the cluster cohort to the Blocked-Pool-for-Cluster-Cohorts;

- Call CS-PROCESSING module.

2. CS-Commit-Processing-End event

- From Join-Pool-for-Cluster-Cohorts find out all the cluster cohorts;
- For every cluster cohort
 - (a) If update-based then inform CDM to commit the transaction (i.e., update the physical database and change the state of updated database in the cache etc.); (Send “Commit-From-CS” message to CDM)
 - (b) If join-based then send all the size information and a “Split-Done” back to its cluster coordinator.
- Release all the locks held by the cluster cohorts;
- Wake up all ready cluster cohorts in the Blocked-Pool-for-Cluster-Cohorts and send them to CDM;
- Destroy all the involved cluster cohorts;
- Call CS-PROCESSING module;
- If CDM idle, then call CDM-PROCESSING module.

3. CS-Abort-Processing-End event

- Find all the cluster cohorts in the Join-Pool-for-Cluster-Cohorts and the Blocked-Pool-for-Cluster-Cohorts;
- Release all the locks held by them;
- Wake up all the ready cluster cohorts in the Blocked-Pool-for-Cluster-Cohorts and send them to CDM;
- Insert the “Abort-From-CS” message into CDM Message Queue;

- Destroy all the involved cluster cohorts;
- Call CS-PROCESSING module;
- If CDM idle, then call CDM-PROCESSING module.

4. CS-Force-Abort-Processing-End event

- For every aborted transaction
 - (a) Find all the cluster cohorts in the Join-Pool-for-Cluster-Cohorts and the Blocked-Pool-for-Cluster-Cohorts;
 - (b) Release all the locks held by the cluster cohorts;
 - (c) Wake up all the ready cluster cohorts in the Blocked-Pool-for-Cluster-Cohorts and send them to CDM Ready Queue;
 - (d) Insert the "Abort-From-CS" message into CDM Message Queue;
 - (e) Destroy all the involved cluster cohorts.
- Call CS-PROCESSING module;
- If CDM idle, call CDM-PROCESSING module.

A.2.5 CDM-PROCESSING Module

If CDM Message Queue is not empty

1. Then

- (a) Pick up a message from CDM Message Queue;
- (b) Case message of
 - "Commit-From-CS" message
 - i. Change the cache states with commit processing;
 - ii. Call CDM-PROCESSING module.

“Abort-From-CS” message

- i. Change the cache states with abort processing;
- ii. Call CDM-PROCESSING module.

EndCase

2. Else if (CDM Priority Ready Queue is not empty) OR (CDM Ready Queue is not empty)

- Then

- (a) If CDM Priority Ready Queue not empty

- Then load a granted cluster cohort from CDM Priority Ready Queue;
 - Else load a granted cluster cohort from CDM Ready Queue.

- (b) If the involved page is not in cache

- Then

- * Insert the cluster cohort into the related disk queue for reading (Note: after reading, the cache states are changed using the replacement policy and the cluster cohort is inserted into CDM Priority Ready Queue.
 - * If the disk idle, then call DISK-PROCESSING module;
 - * Call CDM-PROCESSING module.

- Else

- * Case Request mode of

- Reading: Insert the cluster cohort into CPU queue for processing reading;

- Writing:

- i. If the page p is in original, unfixed and changed state then obtain a copy and change it into Copy, Fixed and Changed state;

- ii. If the page p is in Original, Unfixed and Unchanged state then change it into Copy, Fixed and Unchanged state;
- iii. If the page p is in Original, Fixed and Unchanged state then upgrade it to the Copy, Fixed and Changed state;
- iv. Insert the cluster cohort into CPU queue for processing writing.

Join: Insert the cluster cohort into CPU queue for processing join operation;

NewJoin: Insert the cluster cohort into CPU queue for processing new join operation.

EndCase

- * If one of the CPUs in a cluster is available, then call CPU-PROCESSING module;
- * Call CDM-PROCESSING module.

- Else CDM just idles.

A.2.6 DISK-PROCESSING Module

If (there exists a cohort in the head of the Disk Queue of a disk)

- Then pick up the cohort and create a new Disk-Processing-End event for the cohort
- Else the disk just idles.

A.2.7 CPU-PROCESSING Module

If (there exists a cohort in the head of CPU Queue) AND (there is one CPU available)

- Then pick up the cohort and create a new CPU-Processing-End event for the cohort and CPU (depending on different Request mode create a different event time)

- Else the CPU just idles.

A.2.8 Events for Cluster Data Manager

1. Disk-Processing-End event

- If it is a cluster cohort with update operation mode (Updating the physical database)
 - then just ignore it
 - else
 - (a) Change the cache states using the replacement policy;
 - (b) Send it to the CPU Queue for processing.
- Call DISK-PROCESSING module.

2. CPU-Processing-End event

- According to the types of Cluster Cohort adjust the index of the next operation;
- Insert the cluster cohort into Cluster Cohort Priority Ready Queue in CS;
- Call CPU-PROCESSING module;
- If CS idle, then call CS-PROCESSING module.

References

- [Agrawal 83] Agrawal, R., Carey, M.J., and DeWitt, D., "Deadlock Detection is Cheap", *SIGMOD RECORD*, January 1983, pp. 19-34.
- [Agrawal 87] Agrawal, R., Carey, M.J., and Livny, M., "Concurrency Control Performance Modeling: Alternatives and Implications", *ACM TODS* 12, 4 (December 1987), pp. 609-654.
- [Aho 75] Aho, A.V., Hopcraft, E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Reading, MA: Addison-Wesley, 1975, Ch. 5.
- [Balter 82] Balter, R., Berard, P., and Decitre, P., "Why Control of the Concurrency Level in Distributed Systems is More Fundamental than Deadlock Management", *Proc. 1st ACM Symp. on Principles of Dist. Computing*, August 1982, pp. 183-193.
- [Bhide 87] Bhide, A. and Stonebraker, M., "Performance Issues in High Performance Transaction Processing Architectures", *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, pp. 277-299.
- [Bhide 88a] Bhide, A. and Stonebraker, M., "A Performance Comparison of Two Architectures for Fast Transaction Processing", *IEEE fourth Conference on Data Engineering*, 1988, pp. 536-545.

- [Bhide 88b] Bhide, A., "An Analysis of Three Transaction Processing Architectures", in *Proceedings of the 14th VLDB Conference*, Los Angeles, California, 1988, pp. 339-350.
- [Boral 90] Boral, H., Alexander, W., Clay, L., Copeland, G., Danforth, S., Franklin, M., Hart, B., Smith, M., and Valduriez, P., "Prototyping Bubba, a Highly Parallel Database System", *IEEE Trans. on Knowledge and Data Engineering*, Vol. 2, No. 1, March 1990, pp. 4-23.
- [Carey 84] Carey, M., and Stonebraker, M., "The Performance of Concurrency Control Algorithms for Database Management Systems", *Proc. 10th VLDB Conference*, Singapore, August 1984, pp. 107-118.
- [Carey 88] Carey, M.J., and Livny, M., "Distributed Concurrency Control Performance: A study of Algorithms, Distribution, and Replication", *Proc. of the 14th VLDB Conf.*, Los Angeles, California, 1988, pp. 13-25.
- [Carey 89] Carey, M.J., and Livny, M., "Parallelism and Concurrency Control Performance in Distributed Database Machines", *Proc. of the 1989 ACM SIGMOD Inter. Conf. on the Management of Data*, Portland, Oregon, Vol. 18, No. 2, June 1989, pp. 122-133.
- [Carey 90] Carey, M.J., Krishnamurthi, S., Livny, M., "Load Control for Locking: The 'Half-and-Half' Approach", *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Nashville, Tennessee, 1990, pp. 72-84.
- [Chen 89] Chen, T.-C., Ramarao, K.V.S., "Performance Analysis of Distributed Commit Protocols", *1989 IEEE Conf. on Data Engineering*, pp. 505-512.
- [Chin 84] Chin, W.N., "Some comments on 'Deadlock Detection is Cheap' in SIGMOD Record Jan 83", *SIGMOD RECORD*, March 1984, pp. 61-63.
- [Coffman 71] Coffman, E.G., Elphic, M.J., and Shoshani, A., "System deadlocks", *ACM Comput. Surveys*, Vol. 3, No. 2, June 1971, pp. 67-78.

- [Copeland 88] Copeland, G., Alexander, W., Boughter, E., and Keller, T., "Data Placement in Bubba", *Proceedings of the 1988 SIGMOD Conference*, June 1988, pp. 99-108.
- [Dan 88] Dan, A., Towsley, D.F., and Kohler, W.K., "Modelling the Effect of Data and Resource Contention on the Performance of Optimistic Concurrency Control Protocols", *Proc. 4th Conf. on Data Engineering*, Los Angeles (February 1988) pp. 418-425.
- [Denning 68] Denning, P.J., "Thrashing: Its Causes and Prevention", *Proc. AFIPS FJCC 33*, 1968, pp. 915-922.
- [Desai 90, Page 598] Desai, B.C., *An Introduction to Database Systems*, West Publishing Company, 1990.
- [DeWitt 88] DeWitt, D., Ghandeharizadeh, S., and Schneider, D., "A Performance Analysis of the Gamma Database Machine", *Proc. ACM SIGMOD Conf.*, Chicago, IL, June 1988, pp. 350-360.
- [DeWitt 90] DeWitt, D., Ghandeharizadeh, S., Schneider, D., Bricker, A., Hsiao, H.-i, and Rasmussen, R., "The Gamma Database Machine Project", *IEEE Trans. on Knowledge and Data Engineering*, Vol. 2, No. 1, March 1990, pp. 44-61.
- [Elhardt 84] Elhardt, K., and Bayer, R., "A Database Cache for High Performance and Fast Restart in Database", *ACM Transactions on Database systems*, Vol. 9, No. 4, December 1984, pp. 503-525.
- [Englert 89] Englert, S. et. al., "A Benchmark of Nonstop SQL Release 2 Demonstrating Near-Linear Speedup and Scaleup on Large Databases", *Tandem Tech. Report*, 89.4, May 1989.
- [Franaszek 85] Franaszek, P., and Robinson, J.T., "Limitations of Concurrency in Transaction Processing", *ACM TODS-10*, 1 (March 1985) pp. 1-28.

- [Gajski 83] Gajski, D., Kuck, D., and Sameh, A., "Cedar - A Large Scale Multiprocessor", *Proc. Int. Conf. Parallel Processing*, August 1983, pp. 524-529.
- [Gardarin 83] Gardarin, G., Bernadat, P., Temmerman, N., Valduriez, P., and Viemont, Y., "Design of a Multi Processor Relational Database System", In *Information Processing '83*, Paris, September 1983, pp. 363-367.
- [Gray 78] Gray, J.N., "Notes on database operating systems", in *Operating Systems: An advanced course, Lecture Notes in CS*, 1978, pp. 393-481.
- [Gray 81] Gray, J., et al., "A Straw Analysis of Probability of Waiting and Deadlock", *IBM Research*, RJ 3066, San Jose, Ca., Feb. 1981.
- [Heiss 91] Heiss, H.-U., and Wagner, R., "Adaptive Load Control in Transaction Processing Systems", in *Proceedings of the 17th International Conference on VLDB*, Barcelona, September 1991, pp. 47-54.
- [Haas 89] Haas, L., Freytag, J.C., Lohman, G., and Pirahesh, H., "Extensible Query Processing in Starburst", *Proc. ACM-SIGMOD International Conference on Management of Data*, Portland, May 1989, pp. 377-388.
- [Hill 86] Hill, M., et al., "Design Decisions in SPUR", *IEEE Computer*, Nov. 1986, pp. 8-22.
- [Hobson 87] 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, pp. 134-146.
- [Holt 72] Holt, R.C., "Some deadlock properties of computer systems", *ACM Computing Surveys*, Vol. 4, No. 3, September 1972, pp. 179-196.
- [Hua 90] Hua, K.A., Lee, C., and Peir, J.-K., "A High Performance Hybrid Architecture for Concurrent Query Execution (Extended Abstract)", *Proceed-*

- ings of Symposium on Parallel and Distributed Processing*, Dallas, Texas, December 1990, pp. 348-351.
- [Hua 91] Hua, K.A., and Lee, C., "Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning", in *Proceedings of the 17th International Conference on VLDB*, Barcelona, September 1991, pp. 525-535.
- [Iyer 88] Iyer, B.R., "Limits in Transaction Throughput - Why Big is Better", *IBM Research Report No. RJ6584* (Nov. 1988), IBM Res. Div., Yorktown Heights, NY10598.
- [Joshi 91] Joshi, A., "Adaptive Locking Strategies in a Multi-Node Shared Data Model Environment", *Proc. 17th International Conference on Very Large Data Bases*, Barcelona, September 1991.
- [Kim 85] Kim, M.Y., "Parallel Operation of magnetic Disk Storage Devices: Synchronized Disk Interleaving", *Database Machines: Fourth International Workshop*, D.J. DeWitt and H. Boral (eds.), Springer-Verlag, 1985.
- [Kitsuregawa 90] Kitsuregawa, M., and Ogawa, Y., "Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC)", in *Proceedings of the 16th VLDB Conference*, Brisbane, Australia 1990, pp. 210-221.
- [Kronenberg 86] Kronenberg, N., Levy, H., and Strecker, W., "VAX clusters: A Closely-Coupled Distributed System", *ACM Trans. on Computer Systems*, Vol. 4, No. 2, May 1986, pp. 130-146.
- [Lakshmi 88] Lakshmi, M.S. and Yu, P.S., "Effect of Skew on Join Performance in Parallel Architecture", In *Proceedings of International Symposium on Databases in Parallel and Distributed Systems*, Austin, Texas, December 1988, pages 107-117.
- [Livny 87] Livny, M., Khoshafian, S., and Boral, H., "Multi-Disk Management Algorithms", *Proceedings of the 1987 ACM-SIGMETRICS Conf. on Mea-*

- surement and Modeling of Computer Systems*, Banff, Alberta, Canada, May 1987, pp. 69-77.
- [Mohan 90] Mohan, C., Narang, I., and 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.
- [Mohan 91a] Mohan, C., and Narang, I., "Efficient Locking and Caching of Data in the Multisystem Shared Disks Transaction Environment", *IBM Research Report RJ8301*, IBM Almaden Research Center, August 1991.
- [Mohan 91b] Mohan, C., and Narang, I., "Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a shared Disks Transaction Environment", *Proc. 17th International Conference on Very Large Data Bases*, Barcelona, September 1991, pp. 193-207.
- [Neches 88] Neches, P., "The Ynet: An Interconnect Structure for a Highly Concurrent Data Base Computer System", *Proc. 2nd Symposium on the Frontiers of Massively Parallel Computation*, Fairfax, October 1988, pp. 429-435.
- [Patterson 88] Patterson, D., Gibson, G., and Katz, R.H., "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proceedings of the 1988 SIGMOD Conference*, June 1988, pp. 109-116.
- [Peterson 83] Peterson, R.J., and 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, pp. 216-243.
- [Pirahesh 90] Pirahesh, H., Mohan, C., Chang, J., Liu, T.S., and Selinger, P., "Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches", RJ7724(71589), IBM Almaden Research Center, September 1990.

- [Ries 78] Ries, D., and Epstein, R., "Evaluation of Distribution Criteria for Distributed Database Systems", *University of California-Berkeley Technical Report*, UCB/ERL M78/22, May 1978.
- [Ryu 90] Ryu, I.K., and Thomasian, A., "Analysis of Database Performance with Dynamic Locking", *JACM* 37,3 (July 1990) pp. 491-523.
- [Salem 86] Salem, K., and Garcia-Molina, H., "Disk Striping", *Proceedings of the International Conference on Data Engineering*, February 1986, pp. 336-342.
- [Schneider 89] Schneider, D., and DeWitt, D., "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment", *Proceedings of the 1989 ACM SIGMOD Intern. Conf.*, 1989, pp. 110-121.
- [Schrutchin 87] Schrutchin, T. W., "TPF: Performance, Capacity, Availability", *Proc. IEEE Compcon Spring '87*, San Francisco, February 1987, pp. 158-160.
- [Sekino 84] 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, pp. 59-68.
- [Sequent 86] Sequent Computers, "Balance Guide to Parallel Programming", June 1986.
- [Serlin 91] Serlin, O.,(Ed.), "Oracle Version 6.2 for Loosely-Coupled Systems", *FT Systems Newsletter*, Issue No. 101/102, January/February 1991.
- [Shoens 85] Shoens, K., Narang, I., Obermarck, R., Palmer, J., Silen, S., Traiger, I., and Treiber, K., "The Amoeba Project", *Proc. IEEE Compcon Spring '85*, San Francisco, February 1985, pp. 102-105.

- [Stonebraker 86] Stonebraker, M., "The Case for Shared Nothing", *Database Engineering*, 9, 1, March 1986, pp. 4-9.
- [Stonebraker 88] Stonebraker, M., *Readings in Database Systems*, San Mateo, Calif.: Morgan Kaufmann, 1988.
- [Strickland 82] Strickland, J. P., Uhrowczik, P. P., and Watts, V. L., "IMS/VS: An Evolving System", *IBM Systems Journal*, Vol. 21, No. 4, 1982, pp. 490-510.
- [Tandem 87] 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, pp. 60-104.
- [Tandem 88] The Tandem Performance Group, "A Benchmark of Non-Stop SQL on the Debit Credit Transaction", *Proc. ACM SIGMOD Conf.*, Chicago, IL, June 1988, pp. 337-341.
- [Tay 85] Tay, Y., Goodman, N., and Suri, R., "Locking Performance in Centralized Databases", *ACM Trans. on Database Sys.*, 10(4), Dec. 1985, pp. 415-462.
- [Teng 84] Teng, J. Z., and Gumaer, R. A., "Managing IBM Database 2 Buffers to Maximize Performance", *IBM Systems Journal*, Vol. 23, No. 2, 1984, pp. 211-218.
- [Teradata 88] Teradata, "DBC/1012 Database Computer System Manual Release 2.0", *Document No. C10-0001-02*, Teradata Corp., November 1985.
- [Thacker 88] Thacker, C. P., Stewart, L.C., and Satterwaite, E.H., Jr., "Firefly: A Multiprocessor Workstation", *IEEE Trans. Computers*, Vol. C-37, No. 8, August 1988, pp. 909-920.