1

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

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 V. la 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

Partitioning B⁺-tree in a Parallel and Distributed Computing Environment

by

K.W. Wong, Hons. B.C.S.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment 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
Decemeber, 1994

Copyright © 1994, K.W. Wong, Hons. B.C.S.

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 ACCORDE UNE LICENCE
IRREVOCABLE ET NON EXCLUSIVE
PERMETTANT A LA BIBLIOTHEQUE
NATIONALE DU CANADA DE
REPRODUIRE, PRETER, DISTRIBUER
OU VENDRE DES COPIES DE SA
THESE DE QUELQUE MANIERE ET
SOUS QUELQUE FORME QUE CE SOIT
POUR METTRE DES EXEMPLAIRES DE
CETTE THESE A LA DISPOSITION DES
PERSONNE INTERESSEES.

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 PROPRIETE
DU DROIT D'AUTEUR QUI PROTEGE
SA THESE. NI LA THESE NI DES
EXTRAITS SUBSTANTIELS DE CELLE-
CI NE DOIVENT ETRE IMPRIMES OU
AUTREMENT REPRODUITS SANS SON
AUTORISATION.

ISBN 0-315-98569-0

Canada

Name Dennis K.W. LUNG

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.

Applied Sciences, Computer Science

SUBJECT TERM

0984

U·M·I

SUBJECT CODE

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 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
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
General 0289
Ancient 0290
Linguistics 0291
Modern 0401
Literature 0294
General 0295
Classical 0297
Comparative 0298
Medieval 0316
Modern 0591
African 0305
American 0352
Asian 0355
Canadian (English) 0593
Canadian (French) 0311
English 0312
Germanic 0315
Latin American 0313
Middle Eastern 0314
Romance 0314
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
General 0770
Accounting 0454
Banking 0338
Management 0385
Marketing 0501
Canadian Studies 0503
Economics 0505
General 0508
Agricultural 0509
Commerce-Business 0510
Finance 0511
History 0358
Labor 0366
Theory 0351
Folklore 0578
Geography 0366
Gerontology 0351
History 0578
General 0578

Ancient 0519
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
General 0938
Criminology and Penology 0631
Demography 0628
Ethnic and Racial Studies 0629
Individual and Family 0630
Studies 0700
Industrial and Labor 0344
Relations 0709
Public and Social Welfare 0999
Social Structure and Development 0453
Theory and Methods 0453
Transportation 0453
Urban and Regional Planning 0453
Women's Studies 0453

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 0306
Biology 0287
General 0308
Anatomy 0309
Biostatistics 0379
Botany 0329
Cell 0353
Ecology 0369
Entomology 0793
Genetics 0410
Limnology 0307
Microbiology 0317
Molecular 0416
Neuroscience 0433
Oceanography 0821
Physiology 0778
Radiation 0472
Veterinary Science 0786
Zoology 0760
Biophysics 0425
General 0596
Medical 0596

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0596

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 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 0420
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 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences 0485
Chemistry 0749
General 0486
Agricultural 0487
Analytical 0488
Biochemistry 0488
Inorganic 0738
Nuclear 0490
Organic 0491
Pharmaceutical 0494
Physical 0495
Polymer 0754
Radiation 0405
Mathematics 0605
Physics 0986
General 0606
Acoustics 0608
Astronomy and Astrophysics 0748
Atmospheric Science 0607
Atomic 0798
Electronics and Electricity 0759
Elementary Particles and High Energy 0609
Fluid and Plasma 0610
Molecular 0752
Nuclear 0756
Optics 0611
Radiation 0463
Solid State 0346
Statistics 0984
Applied Sciences 0346
Applied Mechanics 0984
Computer Science 0984

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 0794
Materials Science 0548
Mechanical 0743
Metallurgy 0551
Mining 0552
Nuclear 0549
Packaging 0765
Petroleum 0554
Sanitary and Municipal 0790
System Science 0428
Geotechnology 0796
Operations Research 0795
Plastics Technology 0994
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 recommend to the Faculty of Graduate Studies
and Research acceptance of the thesis**

**Partitioning B⁺-tree in a Parallel and Distributed Computing
Environment**

submitted by

K.W. Wong, Hons. B.C.S.

**in partial fulfillment of the requirements for
the degree of Master of Computer Science**

Chairman, School of Computer Science

Thesis Supervisor

Carleton University

Decemeber, 1994

Abstract

With improving processor technology, the difference in data access time between disk storage and main memory continues to widen. This is even more so in multiprocessor systems and severely degrades the performance of a database system on a parallel machine, since databases are I/O intensive. A solution to this problem is to use multiple disk units and to scatter the data over them. The performance becomes dependent on the methodology of data partitioning. In this thesis, we present a new partitioning method based on an earlier key range method proposed by [Liv 87] in order to improve the performance of a parallel database using B^+ -tree indexed file. The key range partitioning method of a B^+ -tree for parallel or distributed system uses a global dictionary in which the records are distributed according to the key values. This approach is very efficient in handling range and exact match queries, and the performance of the design depends on whether a uniform data distribution can be maintained. One drawback of this design is that the dictionary is static. A priori knowledge of the data distribution during the initialization phase of the indexes is required in order to maintain an equal data size on each disk. Unlike the earlier method, our design does not require knowledge of the data distribution during the initialization phase. The global dictionary is rebuilt according to the data size on each disk as data imbalances appear. Using pruning and grafting operations on the B^+ -tree, data can migrate from one disk to the other according to the new ranges in the dictionary. As a result, a uniform data distribution is maintained dynamically. This and other partitioning methods are implemented in the Intel IPSC[®]/860 supercomputer, and some experimental results of their performances are compared.

Acknowledgements

I would like to thank my supervisor, Ekow J. Otoo, for his guidance, his invaluable comment and advice, and his patience while modifying and editing the thesis drafts, which made all things possible for me.

I would also like to thank Anton Saarimaki, with his advice, knowledge, and assistance to help me to develop an understanding on the hypercube machine. I would like to express my thanks to the technical staff in the school of computer science, especially Peter Choynowski, who always has to reboot the machine whenever my program caused it to fail.

I would also like to acknowledge the financial support from the School of Computer Science, Carleton University.

My thanks also go to the moral support of my numerous friends, without their encouragement and advice, I would not have been able to accomplish it. I would also like to acknowledge the help of my friends who proofread my manuscript.

Contents

Abstract	iii
Acknowledgements	iv
List of Tables	viii
List of Figures	ix
1 Introduction	1
2 Background and Related Work	10
2.1 System Overview	10
2.2 Definition of B-tree and B ⁺ -tree.....	12
2.2.1 B ⁺ -tree.....	14
2.3 Different Strategies for B-tree in a Parallel Environment.....	17
2.3.1 Key Range Declustering.....	18
2.3.2 Hash Declustering	20
2.3.3 Round Robin Declustering	21
2.3.4 Hybrid-range Declustering.....	21
2.3.5 Large Page B-tree.....	22
2.3.6 Distributed by Pages.....	23

2.4	Summary and Motivation.....	27
3	Design of the Dynamic Range Declustering Strategy	30
3.1	Detection State	30
3.1.1	Statistical Maintenance	31
3.1.2	Criteria for Global Imbalance and Strategy for Global Detection	39
3.2	Balancing State.....	46
3.2.1	Concept of Pruning and Grafting	47
3.2.1.1	Pruning Algorithm.....	48
3.2.1.1.1	Removal of Data Items.....	49
3.2.1.1.2	Rebalancing.....	54
3.2.1.1.3	Information Gathering.....	62
3.2.1.1.4	Cost for Pruning	65
3.2.1.2	Grafting Algorithm.....	65
3.2.2	Construction of a New Range Table	71
3.2.3	Data Migration Handling	73
3.2.4	Query Processing during Balancing Phase.....	77
4	Simulation Experiments	79
4.1	Objectives for Simulations.....	80
4.2	Simulation Model	84
4.3	Simulation Result	88
4.3.1	Response Time and Throughput under Insertion	88
4.3.2	Response Time and Throughput for Range Queries	96
4.3.3	Response Time and Throughput for Exact match queries	107
4.4	Summary of Results	112

5	Further Discussion	113
5.1	Query Processing under Dynamic Key Range Declustering	
	Strategy.....	113
5.1.1	Select Operation	114
5.1.2	Project Operation.....	116
5.2	Handling of Duplicate Records.....	116
5.3	Dynamic Key Range Declustering in Multiple Files Environmem	120
6	Conclusions	123
7	References	126

List of Tables

2.1	Comparison of the number of disk accesses used for each strategy based on B-tree.....	27
4.1	Comparison between different strategies for insertions under 8 nodes.....	95
4.2	Comparison between different strategies for insertions under 4 nodes.....	95
4.3	Comparison between different strategies for 0.05% range queries under 8 nodes	105
4.4	Comparison between different strategies for 0.05% range queries under 4 nodes	105
4.5	Comparison between different strategies for 1% range queries under 8 nodes	105
4.6	Comparison between different strategies for 1% range queries under 4 nodes	106
4.7	Comparison between different strategies for exact match queries under 8 nodes	110
4.8	Comparison between different strategies for exact match queries under 4 nodes	111

List of Figures

2.1	Logical view of a shared-nothing database system with n processors	11
2.2	Schematic structure of a page of the B-tree	13
2.3	An example of a B-tree of order $k = 3$	14
2.4	Schematic view of a B^+ -tree.....	15
2.5	An example of B^+ -tree of order $k = 3$	16
2.6	Possible partition of a relation under range declustering strategy in a five disks.....	19
2.7	A sample large-page B-tree of order 2 in a 2 disks system.....	23
2.8	Configuration of a local-balanced B-tree before the deletion	24
2.9	Configuration of a local-balanced B-tree after the deletion	25
3.1	A possible configuration of a ranked B-tree of order 3.....	33
3.2	Schematic view of a n slots sample table	35
3.3	Different cases for Global Imbalance Detection	43
3.4	B^+ -tree after a left pruning by key 55	54
3.5	Process of merging two pages P and P'	56
3.6	Process of rotation	57
3.7	The process of page merge and rotation for pages under different parents	60
3.8	Illustration of sibling from different branches	64
3.9	Possible outcome of grafting two trees of same height.....	67

3.10	Possible outcomes of grafting two trees of different height.....	69
4.1	The average response time of insertion for DRD strategy of different tolerance level (τ) using 8 nodes.....	89
4.2	The throughput of insertion for DRD strategy of different tolerance level (τ) using 8 nodes.....	89
4.3	The average number of disk accesses of insertion for DRD strategy of different tolerance level (τ) using 8 nodes.....	89
4.4	The average response time of insertion for DRD strategy of different tolerance level (τ) using 4 nodes.....	90
4.5	The throughput of insertion for DRD strategy of different tolerance level (τ) using 4 nodes.....	90
4.6	The average number of disk accesses of insertion for DRD strategy of different tolerance level (τ) using 4 nodes.....	90
4.7	The cost of balancing during insertion for DRD strategy of different tolerance level (τ) using 8 nodes.....	91
4.8	The cost of balancing during insertion for DRD strategy of different tolerance level (τ) using 4 nodes.....	91
4.9	The average response time of 0.05% range query for DRD strategy of different tolerance level (τ) using 8 nodes.....	97
4.10	The throughput of 0.05% range query for DRD strategy of different tolerance level (τ) using 8 nodes.....	97
4.11	The average number of disk accesses of 0.05% range query for DRD strategy of different tolerance level (τ) using 8 nodes.....	97
4.12	The average response time of 0.05% range query for DRD strategy of different tolerance level (τ) using 4 nodes.....	98

4.13	The throughput of 0.05% range query for DRD strategy of different tolerance level (τ) using 4 nodes.....	98
4.14	The average number of disk accesses of 0.05% range query for DRD strategy of different tolerance level (τ) using 4 nodes.....	98
4.15	The average response time of 1% range query for DRD strategy of different tolerance level (τ) using 8 nodes.....	101
4.16	The throughput of 1% range query for DRD strategy of different tolerance level (τ) using 8 nodes.....	101
4.17	The average number of disk accesses of 1% range query for DRD strategy of different tolerance level (τ) using 8 nodes.....	101
4.18	The average response time of 1% range query for DRD strategy of different tolerance level (τ) using 4 nodes.....	102
4.19	The throughput of 1% range query for DRD strategy of different tolerance level (τ) using 4 nodes.....	102
4.20	The average number of disk accesses of 1% range query for DRD strategy of different tolerance level (τ) using 4 nodes.....	102
4.21	The average response time of exact match query for DRD strategy of different tolerance level (τ) using 8 nodes.....	108
4.22	The throughput of exact match query for DRD strategy of different tolerance level (τ) using 8 nodes.....	108
4.23	The average number of disk accesses of exact match query for DRD strategy of different tolerance level (τ) using 8 nodes.....	108
4.24	The average response time of exact match query for DRD strategy of different tolerance level (τ) using 4 nodes.....	109
4.25	The throughput of exact match query for DRD strategy of different tolerance level (τ) using 4 nodes.....	109

4.26	The average number of disk accesses of exact match query for DRD strategy of different tolerance level (τ) using 4 nodes.....	109
5.1	Definition of EMPLOYEE relation.....	114
5.2	A sample select operation	115
5.3	A possible organization of duplicate records in a B ⁺ -tree.....	117

Chapter 1

Introduction

In a database processing environment, the speed of the I/O is the major determinant of the overall system performance. With rapidly changing memory and processor technology, the speed gap between the disk storage and main memory is becoming even wider. This raises the problem of I/O bottleneck in parallel database systems. Unless systems are equipped with sufficient amount of main memory to hold both the programs they run and the data required in a non-volatile fashion, the information must be transferred to the slow non-volatile devices like disk for persistence and to preserve data integrity. Even though there are systems based entirely on silicon memory like the Silicon Database Machine [Leland 85], the cost is just too high when compared to disk based system. This indicates that the speed of such non-volatile storage devices may still dictate the performance of an entire system. Many applications, specifically database applications, are I/O intensive, and the slowness of disk storage becomes a severe performance bottleneck. This discrepancy is further intensified as multi-processor systems become popular. Given that the performance of individual disk units is unlikely to improve significantly, there is a need to consider how to efficiently exploit multiple disk units to improve system performance. Furthermore, the cost of using a single large but fast disk is far too expensive and is difficult to incrementally expand. This thesis addresses the problem of I/O bottleneck with particular focus on the use of B^+ -tree index for data accesses. We examine how the techniques developed are used in processing various classes of queries; namely

exact match, range queries and update. Implementation of different operations in relational database are also investigated.

Performance

As an alternative to a single expensive disk, different strategies based on arrays of inexpensive disks have been exploited [Sal 86, Kim 86, Liv 87]. The first idea proposed to improve the I/O bandwidth is *disk striping* [Sal 86]. A group of disk units is *stripped* if each data block is stored, not on a single disk, but across all of the disks. In this way, the response time of any disk access can be reduced since one of the major components of disk activities, data transfer time, is greatly reduced due to the smaller transfers of data blocks from each disk. This idea of disk interleaving has also been successfully implemented in the Cray operating system [Mason] and some early Unix systems. However, there are two major drawbacks in increasing the number of disks. As more disk units are used, the total cost associated with the disks' seek and rotationary delay time increases since any disk access must involve all the disks. Moreover, a disk access has to wait for the slowest disk in the entire system. Concurrent accesses to the disk are also limited. These drawbacks limit the potential for expansion using this strategy.

While K. Salem [Sal 86] was trying to interleave the disk access at a sector level, similar idea of disk interleaving was being investigated by Kim [Kim 86] and Miron Livny [Liv 87] at a different granularity. Kim studied the effects of declustering by spreading the bytes in a data block over all disks in a round robin manner. The goal was to simulate a parallel readout drive using multiple disks in order to remove the wait on the slowest disk. But the hardware cost for constructing a synchronized disk is quite expensive and the idea

was not popular. In Miron Livny's paper, a file is the target of interest. If n is the number of disk units, a file under Livny's strategy would be divided into n (almost) equal sized sub-files, and each sub-file is then mapped to the n disks. The file is then said to be *declustered*. The traditional clustering method, i.e., storing a file in one single disk, was shown to be outperformed by the declustered strategy under both uniform and non-uniform access patterns in their simulation experiments [Liv 87]. They suggested three methods to partition a file: round-robin, hashing, and range. Depending on the partitioning method, a query request can be localized to a single disk, thus reducing the overhead associated with the seek time. These declustering methods became the corner stone for the later Gamma Machine [Dewi 84, Dewi 90] and Bubba database machine [Boral 88, 90].

Reliability

As disk access time is reduced by employing more disks, the reliability of the whole system drops accordingly. It is estimated that the reliability of a multiple disk system, Mean Time To Failure (MTTF), will decrease linearly as the number of disks increases. For example, the MTTF of a 100 disk system will be 300 hours provided that each disk in the system has a MTTF of 30,000 hours or 2 weeks. Clearly, something needed to be done to counter the reliability problem.

Several schemes based on ideas like disk mirroring, chained declustering, parity-code checking have been proposed [Patt 88, Gibs 89, Hsiao 90]. Among these schemes, RAID (Redundant Arrays of Inexpensive Disks) and disk mirroring are the two extremes in terms of extra disk space being used for data recovery. Disk mirroring is one of the most expensive options for improving data reliability. It achieves this by duplicating all disks, and replicating the data on the data disk. This means that every write to the original

data disk will be a write to the backup disk as well. When a failure occurs, the backup copy takes over. This scheme was used in the Tandem Non-Stop system [Borr 81].

To reduce the overhead in extra storage cost, RAID data storage scheme uses an array of small, inexpensive disks as a single storage unit termed a *group*. Instead of replicating data on different drives, this strategy stores check (parity) bytes for recovering from both random single byte failure and single disk failures. It interleaves (stripes) each data block, in units of one disk sector, across multiple disk drives such that all the sectors of a single block can be accessed in parallel, resulting in a significant increase in the data transfer rate¹.

When there is no failure, writing onto a sector requires 4 disk accesses: a read and write of the sector being updated, and a read and write of the check sector of the corresponding block. If a disk failure occurs, however, every disk in the same RAID group must be accessed each time an access is attempted to a sector on inoperative disk drive. The recovery procedure on a byte to byte basis is similar to that of error-correcting code such as parity coding. The details are shown in their paper [Pat 88] and is not described here.

One of the major problem with the RAID scheme is a potential increase of malfunction resulting from a component failure when the number of disks used increases. The large numbers of disks and support hardware (cables, controllers, etc.) will increase the probability of a component failure and increases the probability of unavailable data. Despite these problems, the low cost and good space utilization of the RAID scheme

¹This is the level 5 RAID in [Patt 88]. The level 1 is actually mirrored disk, and level 3 is the same as the synchronized interleaved disk [Kim 86] except for the end of fragment parity byte.

makes it very attractive as a good solution to improving data availability for low cost systems such as PCs and workstations.

Chained declustering [Hsiao 90] strategy has been proposed to provide data availability even in a situation of more than a single disk failure while maintaining a uniform load among all disks. With chained declustering, a file is partitioned into m (almost) equal sized fragments. The backup copy is stored at the adjacent disk of where the original copy is stored in a n -disks system. For example, the backup copy of a fragment original stored in disk i in a n -disks group will be stored in the $(i+1)\bmod(n)^{\text{th}}$ disk. Any update of the original fragment will imply an update of the backup copy. In this way, the possibility of unavailable data can be greatly reduced since the probability of two adjacent disk failures at the same time is only 0.0002 or once every 15.1 years [Hsiao 90]. To balance the work load for each disk during a disk failure, each disk shares part of the extra work load being placed on the disk where the backup copy of the lost fragment is stored. Provided that the file is partitioned into n fragments in a round-robin fashion, the failure of disk i will result in a workload redistribution for the fragment originally stored in disk j and the backup copy stored in disk $(j+1)\bmod n$ according to the following ratio:

$$\begin{aligned} \text{original fragment: } & \begin{cases} \frac{n-j+i}{n-1}, & \text{if } j < i; \\ \frac{j-i}{n-1}, & \text{if } j > i; \end{cases} \\ \text{backup copy: } & \begin{cases} \frac{j-i}{n-1}, & \text{if } j < i; \\ \frac{n-j+i}{n-1}, & \text{if } j > i; \end{cases} \end{aligned}$$

Using the above ratio, workload redistribution can be achieved by redirecting some of the query requests designated at the original fragments to the nodes storing the backup copies. In this way, all the backup fragments are used for processing requests, and the workload can be evenly distributed among all the remaining disks. This approach was very attractive

because it required neither disk I/O nor data movement. Only some of the bound values and pointers/indices in a memory resident control table were needed to be changed. Details of how this can be done for different file organizations are presented in [Hsiao 90].

Specific file organization under multiple disks

The general strategies on how to improve performance and reliability under a system of multiple disk units have been examined. It seems that the performance can always be improved by employing more disks and processors. However, in a shared-nothing multiprocessors database environment, the use of parallelism for all types of queries is not necessarily a good idea. In particular, when the overhead associated with using parallelism to process queries begins to constitute a significant fraction of the execution time of the query, the overall throughput of the system diminishes. On the other hand, for those queries that consume significant CPU and I/O resources, the use of parallelism will be advantageous since the overhead associated with using parallelism is not significant. These depend on the declustering strategy of how relations in a relational database system or files are partitioned among all disks.

Range declustering strategy was one of the three declustering strategies employed by Gamma [Dewi 84, 90]. It has the characteristic of being able to localize the execution of selection query that has a range predicate on the partitioning attribute, to as few processors as possible, thus reducing the overhead cost. A range partitioned relation will have the range of values of the partitioning attribute for each processor stored in a *range table*. Unfortunately, the original strategy was static and requires that the partitioning range value be known prior to its creation. When the access pattern does not fit the range table, an

imbalance in the workload of the system occurs and the performance of the system degenerates, where the performance is measured by the response time to queries and the number of requests processed per unit time.

In a database environment, relations are usually large and have to be stored physically as files on disks. A *relation* is defined as a subset of a cartesian product of a list of domains. The domain of an attribute is a set of permitted values the attribute can have. A file is organized logically as a sequence of records. These records are mapped onto disk blocks. Data in it must be moved from disk to main memory in order for the data to be operated on, since the entire database is stored as files on disk. After operations are performed, the data must be returned to disk. Efficient access and modification can be achieved if the attributes of relations are indexed. There are many indexing schemes that can be used to improve the performance of selection queries, join operation, etc. In general, most database management systems provide indexing method, such as B^+ -tree, to enhance the system performance. In a database of N records, it takes $O(\log N)$ time to locate a record with a given key value. It is also very efficient in processing range queries based on the indexed field since the data pages are linked sequentially. Just as indexes on a join attribute promote efficient join execution in a single processor environment, query processing on a parallel machine can be enhanced by providing a means to access data indexed on some attributes. As B^+ -tree remains one of the most widely used secondary index scheme due to its efficiency in handling exact match query and range query, there are only a few researches in the area of applying B^+ -trees in a parallel environment [Pram 90, Dewi 90, Seeger 91, Grand et al 90]. We believe the methods to partition an indexed file and the way to organize the indexes have a significant impact on the performance. The key range declustering strategy described has nice features in processing single record queries and small range queries since they can be localized. A B^+ -tree index can be easily adopted

by constructing an independent B^+ -tree for each file fragment. Assuming that the requests are generated continuously and are processed in a non-interactive fashion, the key range declustering strategy will be an ideal strategy to partition a relation if an even data distribution can be maintained. However, an even data distribution cannot be obtained if the access pattern is not known.

In this thesis, we propose a method to dynamically partition a relation based on a B^+ -tree file structure and the range partitioning method. Unlike the range partitioning method, the new strategy will modify the range table of each processor from time to time based on the data load of each processor. The redistribution of data allows a better data distribution among nodes, which translates into a better performance when answering queries. The contributions of this thesis are:

- to propose a new declustering strategy for a parallel database system which will monitor the data distribution of a B^+ -tree file and correct any discrepancy that may arise;
- to provide fast data distribution operations on B^+ -tree, like pruning and grafting, and to allow fast data migration between neighboring nodes;
- to have a data load balanced without a priori knowledge of the access pattern.

In the sequel, we make the assumption that the system is a shared-nothing environment and it is I/O bound. This is not an unlikely scenario given the recent increase in processor speed.

The organization of this thesis is as follows. In Chapter 2, we describe the overview of the database system and different strategies based on B^+ -tree. The new declustering

design will be described in Chapter 3 and the simulation model and result will be described in Chapter 4. The issues of query processing and the handling of duplicate records will be investigated in Chapter 5. Finally, we conclude in Chapter 6 and give some directions for future work.

Chapter 2

Background and Related Work

In this chapter we introduce the corresponding environment for which we build a partitioned file system; namely, a multi-processor system with multiple path and multiple data path. We describe the concept of shared-nothing architecture, and then describe the definition of B-tree and B⁺-tree. We will go on to describe different strategies using B⁺-tree in a parallel environment. A simple analysis of its effectiveness on a shared-nothing architecture is also given.

2.1 System Overview

The system is based on the concept of a shared-nothing architecture [Stone 86] in which processors do not share disk drives or random access memory, and can only communicate with one another by sending messages through an interconnection network. Mass storage in such architecture is generally distributed among the processors by connecting one or more disk drives to each processor as shown in Figure 2.1. We will refer to the processor and disk drive couple as a *node*. The shared-nothing system has some advantages over the other systems. First, there is nothing to prevent the architecture from scaling to thousands of processors, unlike shared memory machines for which scaling beyond 30-40 processors may be impossible. Second, a high aggregate I/O bandwidth can be achieved by associating a small number of disks with each processor and declustering each relation across the disk drives. A relation is assumed to be stored as a file on a disk. A system with

a large number of disk drives is ineffective in increasing I/O bandwidth unless custom disk controller is used as shown in [Cope 88]. Another advantage is that one can employ off-the-self mass storage technology to scale up the storage space.

Figure 2.1: Logical view of a shared-nothing database system with n processors. P_i represents a single processor, $0 < i < n$, with a number of disk drives (represented as a bucket) directly connected to each processor.

We further assume that each file is declustered using the available number of nodes in the system and has its own B^+ -tree indexes which served as a primary index. Therefore, accessing and updating the file will have to go through those operations of the corresponding B^+ -tree. If a file is partitioned into n fragments, an independent B^+ -tree will be built and stored at each node's disk in the system. The information in a B^+ -tree stored in node i is not shared by any other B^+ -tree in node j , $i \neq j$. Therefore, a query directed to node i can be processed independent of the state of node j . This idea is used in the query generation described in Chapter 4, where some simulation experiments are discussed.

To exploit the system capacity, the query is assumed to be generated continuously and processed in a batch fashion in each node. So we configure each node with a message queue so that the queries can wait inside the node. The query generation is also made to be independent of the idleness of the individual processor. The queries are assumed to be generated outside the system and come in through the interconnection network as shown in Figure 2.1. This can be thought of as independent query generated through the host. The host, acting as a query manager, is responsible for sending a query to the proper node/s for processing and monitoring its completion. The query is distributed according to the distribution of the records. In other words, each record is accessed with equal probability.

2.2 Definition of B-tree and B⁺-tree

The B-tree has been defined by Bayer and McCreight [Bayer 72], Comer [Com 79], and Knuth [Knuth 73] in slightly different ways. The following definition is adopted from the definition in [Knuth 73].

Definition 2.1

A B-tree of order k is a balanced multi-way tree with the following properties:

1. A page has at most k children;
2. Every page except the root page has at least $\lceil k/2 \rceil$ children;
3. The root may have at least two children except when it is a leaf page in which case it could be empty;
4. A non-terminal page with s children contains $s-1$ keys for $\lceil k/2 \rceil \leq s \leq k$;
5. All terminal pages appear at the same level, i.e., the path length from the root to any terminal page is the same.

The structure of a page in a B-tree is shown in Figure 2.2. The record pointers associated with the keys are ignored, and only the pointers to other index pages are shown. Let the keys and page-pointer pairs in a page Q be labeled as shown below.

$$(P_0, K_1, \alpha_1); (P_1, K_2, \alpha_2); \dots; (P_{s-2}, K_{s-1}, \alpha_{s-1}); (P_{s-1}, K_s, \alpha_s); P_s$$

Figure 2.2: Schematic structure of a page of the B-tree.

All the keys in a page are arranged in a non-decreasing order, i.e., $K_1 \leq K_2 \leq K_3 \leq \dots \leq K_s$, with K_s being the highest key value in the page Q . For any key that appears in the page Q , say K_i , we use the record pointer α_i to locate the record. Otherwise, it is probably located in the next level page pointed by the page-pointer P_{i-1} . The page-pointer P_{i-1} points to the page in the next level with key value in the range of K_{i-1} and K_i but is not equal to either K_{i-1} or K_i , where $i > 1$. If $i = 1$, then the page that P_0 pointed to has key value less than K_1 . For $i = s+1$, the page that P_s pointed to has key value greater than K_s .

It is shown by Bayer and McCreight [Bayer 72] that the minimum and maximum number of pages in a B-tree of order k are given by

$$N_{\min} = 1 + \frac{2}{v}((v+1)^h - 1) \dots \dots \dots (2.1),$$

and

$$N_{\max} = \frac{1}{2v}((2v+1)^h - 1) \dots \dots \dots (2.2),$$

where h is the height of the tree for $h \geq 1$, and $v = \lceil k/2 \rceil$.

It follows from equation (2.1) and (2.2) that the maximum height of the tree is $h = \log_v(N)$. As an example, a 2-3 tree, B-tree of order 3, is shown in Figure 2.3. Because

the records are embedded within a page, it takes $O(h.q.)$, where q , being the number of records falls in the interval, to process the range query.

Figure 2.3: An example of a B-tree of order $k = 3$, only key values and page-pointers are shown.

2.2.1 B⁺ tree

We have shown in the previous section that the cost for a retrieval is at most $O(\log_{\lceil k/2 \rceil}(N))$. To improve upon the cost of retrieving records in sequence order, the B⁺-tree was proposed by Knuth [Knuth 73]. This scheme achieves $O(1)$ (constant) operation to retrieve the next or previous record in sequence without compromising the random access retrieval cost. Most storage schemes use B⁺-tree since the range queries can be handled efficiently. The B⁺-tree may be defined as a standard B-tree except that the records (associated pointers to the actual records) reside only in the leaf pages. The leaf pages are linked in key sequence order and are called the *sequence set*. The internal pages contain values that serve as separators of the leaf pages and are duplicates of the lowest key values in the leaf pages. The internal pages of the B⁺-tree are structured as the standard B-tree index. This is termed the *index set*. The entries in the index set only serve to direct the traversal along the correct path to the leaf page where the target record of the search key

To delete or to insert a record, the record is simply removed from or added to the leaf page. When the number of children in a page violates the criteria specified, a balancing action takes place to ensure the criteria are met. For example, when there is an attempt to insert a record into a page having k children, a *split* will take place. The split will actually split the page with the new record into two pages, such that each page meets the criteria specified. Of course, this might cause an insertion in the higher level and the action might be propagated recursively up the tree. Similarly, when there is an attempt to remove a child from a page with only $\lceil k / 2 \rceil$ children, a *merge* with its left or right neighboring page may take place. Alternatively, the criteria within a page may be satisfied simply by *borrowing*

Figure 2.4: Schematic view of a B⁺-tree, where $\lceil k/2 \rceil < s \leq k$ and $\lceil k/2 \rceil < t \leq k$.

To delete or to insert a record, the record is simply removed from or added to the leaf page. When the number of children in a page violates the criteria specified, a balancing action takes place to ensure the criteria are met. For example, when there is an attempt to insert a record into a page having k children, a *split* will take place. The split will actually split the page with the new record into two pages, such that each page meets the criteria specified. Of course, this might cause an insertion in the higher level and the action might be propagated recursively up the tree. Similarly, when there is an attempt to remove a child from a page with only $\lceil k/2 \rceil$ children, a *merge* with its left or right neighboring page may take place. Alternatively, the criteria within a page may be satisfied simply by *borrowing*

keys or data from its neighbors. A merge actually combines two pages into one, which means a deletion of an index entry must take place in the parent level; while the borrowing moves some of the entries from the neighbors to the current page in order to satisfy the definition of the B^+ -tree page. The details of the algorithms for insertion and deletion are given in [Knuth 73] and we will describe them in details in the next chapter.

There is one major distinction between the B-tree and B^+ -tree in the process of deletion. If the key of the deleted record is a separator key, a copy of it is still retained in the index set. This is not immediately deleted until the pages for which it serves as the separator value, are merged. Figure 2.5 illustrates the structure of a B^+ -tree. In most literature, reference to the B-tree is in fact reference to the B^+ -tree. Hereafter, the terms B-tree and B^+ -tree will be used interchangeably.

Figure 2.5: An example of B^+ -tree of order $k = 3$.

2.3 Different Strategies for B-tree in a Parallel Environment

There are several different declustering methods for B⁺-tree that have been investigated in the past [Pram 90], [Dewi 90], [Ghand et al 90], [Seeger 91]. These can be categorized into two major classes according to how they partition a relation: 1) each page has its own B⁺-tree and the tuples of a relation are distributed by some kind of function, 2) the whole relation is viewed as a single B⁺-tree file and the problem is how to distribute the B⁺-tree over all disks. The Gamma machine uses three declustering methods: range, hash, and round-robin, plus the hybrid-range declustering strategy, all of which belongs to the first category; while the local balanced B-tree and the large page B-tree belongs to the second. Coincidentally, all the strategies belonging to the first class were originally intended to perform in a shared-nothing environment and those belonging to the second were for a shared memory system. We begin by introducing strategies in the first category. The first three strategies are those used by both Bubba [Boral 88] and Gamma [Dewi 90]. An analysis based on the performance of the different kinds of operations is also given. The performance is defined by the measure of the following:

- *system throughput*- the number of queries that can be performed per second, and
- *response time*- the time interval between the time a query starts and ends.

We also consider the possible effect of communication cost, which may greatly affect the performance. We define *large range query* as a query whose response set consists of at least $n*k$ records, n being the number of disks, and k being the maximum number of records in a page. We also define a *small range query* to be a range query whose response set does not exceed k records. These two types of range queries, and other basic operations of B⁺-tree are used as a basis of comparison of the various methods.

2.3.1 Key Range Declustering

This method defines a disjointed partition of the key range into n consecutive intervals $I_1, \dots, I_n$ during the initialization phase of the file. A record is inserted into the B^+ -tree on disk j , if its key value falls within the interval of I_j . For example, a possible configuration of a 5 disk system under the key range partitioning strategy is shown in Figure 2.6. An insertion of key 6 will be directed to node 1. Under this strategy, any single record operations like insertion, deletion and exact match query are localized to only one disk or tree. When processing a range query, the original range query can be distributed to only those processors that are involved. For example, the range query may involve more than a single disk, depending on the number of matched records falling into the range. Assuming that each disk carries approximately the same number of records of a file. One might expect the following:

Let $\bar{d}_n$ denote the expected number of disks,

q , denote the size of the range query,

n denote the total number of disks used, and

N_G denote the total number of records in a file;

Then

$$\bar{d}_n = \left\lceil n \times (q / N_G) \right\rceil$$

Figure 2.6: Possible partition of a relation under range declustering strategy in a five disks system. The relation has bounds of key values between 0 to 4999.

The advantages of this strategy are: i) it has good response time for single record operations; and ii) it is very good for processing small range queries as the query will be localized to only one disk.

The major problem with this approach is the difficulty in maintaining an even distribution of the data, i.e., nearly equal number of records on each disk. Statistical information on the access pattern must be known in advance in order to form a well-balanced range table. Furthermore, under the assumption that the query has the same distribution as the data, it is not efficient to process a large range query in an interactive manner because the query will tend to localize on a small number of nodes. As access pattern does not correspond to the setting in the range table, deviation of populations between nodes may exist. The nodes with more data are likely to receive more requests for processing than the nodes with less. This results in a longer completion time, which translates into a lower throughput and higher response time.

2.3.2 Hash Decustering

The idea is to use a hashing function on the key value of a tuple to assign the tuple to a specific node. A hashing function is applied to the key of the tuple and the resulting value indicates the id of the disk that the tuple should be stored. Thus any single record operation will be localized onto only one node. For example, in a 5 node system, using a simple hashing function of $(x \bmod n) + 1$, n being 5, x the key value, the key 6 will be inserted into disk 2. For any range query, not matter how small the range, at least one page access is required from each disk. This may involve all processors and disks and utilizes unnecessary system resource. The advantage of this scheme is that the data partitioning is well balanced, so there is no need to gather information to obtain a priori knowledge of access pattern if a properly designed hashing function is chosen. Since it uses all nodes for range query, it can provide maximal parallelism for large range query and thus achieve a better response time and high throughput. However, it has two serious drawbacks: i) it increases the response time for small range queries because the processing time in the slowest node in the system becomes the major determinant as all nodes must be involved in processing the query; ii) the response time of pure small range queries is significantly higher than that of key range strategy due to the overhead involved. As mentioned before, all the nodes must be involved to process a range query, this introduces unnecessary disk accesses if the selection query consists of less than a page. Furthermore, it increases the communication costs because a range query must be broadcast to all nodes and the results must then be collected from all nodes.

2.3.3 Round Robin Declustering

This scheme is similar to the hashing declustering strategy except that it distributes the records in a round-robin fashion instead of using a hashing function based on the key value of a record for disk assignment. The i^{th} record to be inserted is assigned to disk $(x \bmod n) + 1$ in a n -node system. However, for any other single record operations, all nodes must be involved because the knowledge of which node a record is located is not available. This gives the scheme an additional disadvantage over the hash strategy and will significantly increase the response time and overall request load on the system. The only advantage is that a perfectly balanced data load can be achieved if no deletion is allowed.

2.3.4 Hybrid-range Declustering

While the range declustering strategy has a definite advantage in processing small range queries, it falls short when processing large range queries because the queries are only processed by one or two nodes. To correct this problem, a hybrid range declustering can be thought of as a range declustering in p pseudo nodes while the actual system has only n nodes, for n not necessary equal to p . Each fragment is assigned to a real node in a round-robin fashion. The value of p depends on the number of optimal processors or disks needed to participate in the average selection queries and the size of the average query. The optimal value of processors is in turn dependent on the network communication cost, disk access time, the processor speed, access pattern, and additional overhead of an additional processor. A detailed derivation and the formula for the value p are given by [Ghand et al 90]. The p fragmented ranges can be kept in a range table and the way to handle each type of operation is similar to that of key range declustering strategy except that the i^{th} fragment is located at the $(i \bmod n) + 1$ disk. The advantage of this approach is that it avoids the

localization of a single node when processing large range queries, thus improving the overall throughput and response time under interactive processing. The only problem is that the *a priori* information on those parameters must be known in order to compute the optimal value p and the range table remains static in spite of changes in the access pattern.

2.3.5 Large Page B-tree

Recall that in order to access any data in a B^+ -tree, the pages starting from the root page to the leaf page must be accessed. The length of this tree path has a direct relationship with the number of disk accesses being used. This approach comes from the fact that both the height of the B^+ -tree and the probability of overflow must be reduced. This is done by enlarging the size of each B^+ -tree page, referred as a *superpage*, by a fraction of n , in a n -disk system. To balance the load and to make use of the available disks, it partitions a superpage into n subpages, where each subpage is stored on a different disk with the same page address. The keys are unsorted within the superpage, but are sorted in each subpage. This design ensures, when an insertion or deletion causes a split or merge in a subpage, that the modification can be made in the parent subpage directly without accessing the whole parent superpage. However, the entire superpage must be retrieved for any lookup operation. Any range query operation, according to the design², must either scan the leaf pages sequentially, or read the entire scope of the subtree within the range so that all the leaf's superpages can be read in parallel. As an example, the pages with addresses A, B, D, F must be accessed in order to perform a range query of [16, 67] on a large page B-tree in Figure 2.7. The advantage of this approach is the reduction in height of the tree and a good data load balance within the tree. It should have a better performance under large range query. However, as shown in [Salz 88], the tree height is rarely more than three and to

²No strategy for range query is shown in [Pram 90].

access any superpage requires n disk accesses, since all subpages belonging to the superpage must be accessed. This introduces unnecessary page reads. The number of disk accesses and the request load on the system are significantly increased. If synchronized disks are not used, the access of a superpage will then depend on the slowest disk in all disks. Recently, a similar idea has been investigated under a broadcast network [Matsl 90]. Their algorithm requires one broadcast message for each conceptual page accessed.

Figure 2.7: A sample large-page B-tree of order 2 in a 2 disks system. The D_i represents the disk the subpage is located. The page address of each page is represented by A-F.

2.3.6 Distribution by Pages

The basic idea of this strategy is to use an ordinary B^+ -tree, but to distribute the pages of the tree among the disks page by page. In a shared memory system or a single processor system, this requires that each index entry be expanded to include the information about which disk its child is located. In Seeger's paper [Seeger 92], the assignment of page to disk is done by a simple neighboring algorithm, which guarantees that any $\lceil n/2 \rceil$

consecutive pages at the same level will be all located at different disks in a n disks system. The above condition is used to ensure that consecutive pages would be scattered on different disks to guarantee local balancing. This is achieved by the following procedures. If a new page is created, the algorithm excludes all the disks where both the new page's left and right $\lceil n/2 \rceil - 1$ neighboring pages are located from the possible assignment. Then it assigns the new page to the disk with the least data load. In the case of page merging, the n consecutive pages centered on the merging page after the merge may violate the condition. It was suggested to move the conflicting page to a disk which its id is not the same as one of its $\lceil n/2 \rceil - 1$ neighboring pages on the left or on the right. Afterward, each of its $\lceil n/2 \rceil - 1$ neighboring pages, either on the left or on the right, must be examined and corrected in the same way. This is to ensure that the criteria of $\lceil n/2 \rceil$ consecutive pages in the same level are not located on the same disk. The disk assignment scheme is the same as the assignment of a new page. Notice that this indicates that at most $2x(\lceil n/2 \rceil - 1)$ disk accesses would be needed, with one write for the page movement and another for the update in the corresponding parent page. To illustrate, a possible configuration of local balanced B^+ -tree is shown in Figure 2.8. The result after deleting the key 30 is shown in Figure 2.9.

Figure 2.8: Configuration of a local-balanced B-tree before the deletion. The root page is stored in memory.

Figure 2.9: Configuration of a local-balanced B-tree after the deletion. The page in bracket is the page that has been moved to another disk. In this case, from disk 2 to disk 3.

The traversal and handling of local-balanced B⁺-tree are the same as in an ordinary B⁺-tree. It has the same number of disk accesses under each exact match query and insertion when there is no page split³. For deletion, more disk accesses may be needed as mentioned before. It is suggested that all the index pages in the tree be stored in main memory to speed up the operations. If this is the case, it uses at least $\lceil n/2 \rceil$ disks in performing large query, while using only a small number of disks to perform small range query (1 or 2 disk if size of query < size of a page). This improves both the throughput and response time significantly. This scheme assumes a shared memory multiprocessor environment.

Recall that a page merge or a page split in a B⁺-tree requires the knowledge of where the $\lceil n/2 \rceil$ sibling pages are stored. The means of knowing this is not clear in their paper. It will be quite costly if the information about the disk to store neighboring pages is

³One more disk access when there is a page split.

not maintained somewhere within the tree. To obtain such information, we have to scan not only the parent of the current page, but also the parent of the n neighboring pages. These neighboring pages or pointers must be memorized for each level to enable the operations of a page merge or page split. On the other hand, keeping the information further raises some problems. This information may have to be updated every time a neighboring page, within a distance of $\lceil n/2 \rceil$, splits. The information in at most $n-1$ pages may be affected, and consequently has to be updated. This issue becomes more serious if the number of disks used is larger than the order of the B^+ -tree.

A few variants of this approach were also found in [John 92] and [SK 90]. In Johnson's paper, instead of assigning a new page to a different disk, they keep the new page within the current disk unless the load is too high. In that case, the new page is assigned to the neighbors (left or right) with the least number of pages. They also propose a concurrency control method based on page replication so that a consistent copy of the common parent page can be maintained if multiple tasks are updating the same parent.

2.4 Summary and Motivation

We summarize the strategies described above in the following table using the number of disk accesses as a metric of measurement.

strategies	exact match query	update operation	small range query	large range query	platform running	A Priori Information Required?
Key Range	$O(h)$	$O(h)$	$O(h+S)$	$O(h+S)$	SN	Yes
Hashing	$O(h)$	$O(h)$	$O(h.n+S)$	$O(h.n+S)$	SN	No
Round-robin	$O(h.n)$	$O(h), O(h.n)^4$	$O(h.n+S)$	$O(h.n+S)$	SN	No
Hybrid- range	$O(h)$	$O(h)$	$O(h+S)$	$O(h.n+S)$	SN	Yes
Large Page B-tree	$O(n \lceil \frac{h^*}{n} \rceil)$	$O(n \lceil \frac{h^*}{n} \rceil)$	-	-	SM	No
LBT	$O(h^*)$	$O(h^*)$	$O(h^*+S)$	$O(h^*+S)$	SM	No
Johnson's tree	$O(h^*)$	$O(h^*)$	$O(h^*+S)$	$O(h^*+S)$	SN/SM	No

The parameters are defined as follows:

h denotes the height of a B-tree in a single disk,

h^* denotes the height of the global B-tree, $h^* \geq h$.

S represents number of pages in a range query, and

n represents the number of nodes/disks available.

SN stands for shared nothing environment, and

SM stands for shared memory system.

LBT stands for local balanced b-tree.

⁴This is for record removal.

Table 2.1: Comparison of the number of disk accesses used for each strategy based on B-tree.

A simple observation from the above suggests that the key range declustering strategy is the one that uses fewest disk accesses. It is very efficient and has good response time for all types of operations except for large range query. It is also easily scalable. However, it has some serious drawbacks, as mentioned in the previous section. The first drawback is that it is not efficient in handling large range query because only 1 node or disk is used. This may not be true from a different perspective. If a stream of large range queries arrives in the system, it is reasonable to believe that they will be scattered into different nodes assuming the data load is balanced and a message queue is available in each node to store the unprocessed queries. Now, each node should have approximately the same number of queries to be processed. Even though the key range declustering strategy does not use all the nodes to process *a single* range query, it does make use of all nodes for *a sequence* of range queries. This appears to have the same performance as the hybrid-range strategy. In addition, it eliminates some of the overhead in communication because fewer nodes are required for each query. However, this is only true if the range table reflects a balanced data distribution. This is the core idea behind the strategy we propose. We want to design the new strategy with the following four goals:

1. **Better data load balance:** The key range table remains a problem since the table is static and is not immune against changes in the data distribution. To resolve this problem, one might try to balance the data load by data redistribution when data imbalance occurs. That is, to shift records from the node which has excess data load towards the nodes that have less. This can be achieved by finding out a new

range table with good data load balance and then redistributing the records in each node accordingly.

2. **No a priori information is required:** In order to have a well-balanced key range, a priori information on the data distribution is needed, but the information may be costly to collect. To avoid this situation, we have a valid initial range table, which is not necessarily well-balanced. If any data imbalance appears, we go through the redistribution process to correct it. In this way, no a priori information on data distribution is required.
3. **Minimize the cost of balancing:** The redistribution will create lot of activities in communication, computation and disk read/write. This would be quite costly if the frequency of redistribution is high. The frequency of redistribution depends on how the data balance can be tolerated. If we want a consistently perfect balance, we will need to compute the data balance every time an insertion or deletion is carried out. The solution is to draw a balance between a balanced data load and the cost of balancing.
4. **Fast data redistribution:** The records are temporarily not available during the process of transmission from one node to the other. They remain unavailable until the records are fully integrated to the local B^+ -tree. How long the records are locked depends not only on how fast the data can be transferred, but also on the algorithm that perform the *pruning* and *grafting* operation on the B-tree. A fast algorithm shortens the delay introduced by the wait on those records, which in turn translates into a faster response time.

Chapter 3

Design of the Dynamic Range Declustering Strategy

The main idea of Dynamic Range Declustering Strategy (hereby called DRD) is to maintain a data balanced file among the nodes. This means that each node should contain approximately the same amount of data items of that file. As more modifications are done, the data distribution becomes uneven. A situation called *global imbalance* appears. To reduce such imbalance, data items are migrated from the node containing more data to the one containing less. This idea is referred to as *data balancing*. In order to do so, the tree file is subjected to two different phases: the detection phase and the balancing phase. The sole purpose of the first phase is to detect the existence of global imbalance for an individual tree file. If a global imbalance is detected, the system goes into the balancing phase where a new data balance is achieved. The remaining issues relate to how often data balancing is required, and how this can be done in an efficient manner. These issues are investigated in details in the following sections.

3.1 Detection Phase

To detect any data imbalance in a tree file, information on the *population* of a tree file in each node must be known. One way to obtain such information is to designate one of the node to be a *control* node where each node's population information on the tree file is maintained. The local node will only know of its population, the range of intervals, and a

copy of the estimated global population. A global imbalance appears when the data load on the nodes is uneven. Practically, this means that any change in the nodes' population must be reported to the control node. This induces severe usage on both the communication resource and the processing resource on the control node. To avoid such heavy utilization, a certain tolerance is specified so that a local node's population is allowed to deviate from the ideal value. When the population in a node is within the preset tolerance level, the changes in the population of this node are not reflected in the control node. On the other hand, when the population exceeds the pre-designated level, a *potential* global imbalance appears, and the control node is notified. Afterward, a further detection of global imbalance is started by the control node. By allowing such self-detection mechanism, each node is forced to share the workload during the detection phase. The methodology for global imbalance detection and the criteria for determining the global imbalance is further investigated in Section 3.1.2. By using a self-detection scheme, the discrepancy between the population information stored in a local node and that of the control node will always exist. This discrepancy is eliminated when the control node receives a notification of potential imbalance from a local node. After the control node decides there is a global imbalance, the tree then enters into a balancing phase. A new range table is built, and actual data migrates so that a new data balance can be obtained. These are further described in Section 3.2. We start by discussing the strategy on how to maintain statistical information at each node.

3.1.1 Statistical Maintenance

The idea of global detection demands the maintenance of statistics information on the population of the B⁺-tree. This mechanism should be able to answer a query about the rank of a key and to locate a key of certain rank. We used the terms *rank finding* and *ranked*

key finding for these operations respectively. The purpose is to establish a new data balanced dictionary.

There are two possible ways to store the rank information in a B^+ -tree. The first approach is to embed the statistic information within each page of the tree. This idea is called *ranked B^+ -tree* and was investigated in [Antos 92]. The second method is a *sampling table* approach. The idea is to maintain a fixed size table of sample keys selected from the tree. The table also consists of the corresponding sample sizes for each sampling key. The *sample size* is defined as the number of data items in the tree between any two consecutive sampled keys. The sampled keys in the table are sorted, and the aggregate number of samples is set to be the actual tree population. The two approaches are discussed in details.

The key idea of the ranked B^+ -tree is to store the population of any child in the parent of that child. For this, extra space in each page must be provided in order to maintain the population information about its descendants. This indicates that there must be structural changes in the B^+ -tree. Not only will the child pointers and the keys to locate each child be maintained in each page, but the population of each child sub-tree is also stored. The population of any child in a leaf page is assigned a population of one. As an example, a possible configuration of a ranked B-tree of order 3 is shown in Figure 3.1.

Figure 3.1: A possible configuration of a ranked B-tree of order 3. The small printed number indicates the population associated with the child pointer.

The rank of a given key can be easily answered by finding the given key in the tree and accumulating the population of the sibling to its left. The algorithms on rank finding and ranked key finding are summarized as follows:

Rank Finding

1. Assign the current page C to be the root and initialize Rank to be zero;
2. Pick the correct branch for the given key K^* ;
3. Add the population of all children which are the left siblings of the child found to Rank;
4. If C is not a leaf, set C to be the child found, go to step 2;
5. If the child has the same key value, add one to Rank;

Ranked Key Finding

1. Assign the current page C to be the root;
2. Find child: accumulate the population of all the children's population of C until it is just less than R^* ;

3. Reduce R^* by that amount and pick the child to be the right neighbor of the child accessed;
4. If C is not a leaf, set C to be the child found, and go to 2;
5. The child found is the one with the given rank;

As can be observed, this method allows a quick location of the ranked key as well as fast rank finding operation. However, maintenance of the ranked B^+ -tree incurs a high cost of updating populations along the entire root-to-leaf path for each update operation. This high maintenance cost limits any usage of the ranked B^+ -tree only to the case of low update rate environment.

The second approach to determine the rank of a key is to use a sampling table. The key idea of the sampling table is very similar to a scaled down version of the ranked B^+ -tree. A certain number of key values are picked in random from the tree, and they are arranged in the same order as they appeared in the tree. Instead of maintaining statistics on all pages, only the relative populations between any two consecutive key values are maintained. Formally, we define the combination of the relation population between any two keys (X, Y) to be the sample sizes for the key Y , and we denote the combination of the sampling key Y and its corresponding sample sizes to be a sampling slot. A sampling table can be conceived as a table consisting of a fixed number of sampling slots. To maintain the true population in the tree in a sampling table, zero number of samples can be assigned to the first slot, and the start off sampling key value is assigned to have the same key value as the leftmost descendant in the tree. The last sampling key will then be the key of the rightmost descendant in the tree. In this way, the sum of all slots' population will be equal to the population of the entire tree. The schematic view of a sampling table is shown in Figure 3.2.

$$[(K_0, 0), (K_1, P_1), (K_2, P_2), (K_3, P_3), \dots, (K_n, P_n)]$$

Figure 3.2: Schematic view of a n slots sample table. Each bracket indicates a slot and K_i , P_i represent the sampled key and sample size in the i th slot.

Notice that the minimum number of slots in the table is at least two because the first and the last slot are used to represent the boundary key values of the tree. An insertion of key value K will add one to the sample size in the slots i , where $K_{i-1} < K \leq K_i$. Similarly, the sample size in slot i will decrease by one if a tuple with key K is removed. The strategy for rank finding and ranked key finding is similar to that of ranked B-tree. Only the version where the keys are maintained in ascending order is shown. The methods for rank finding and ranked key finding for the tree in descending order will be similar. They are described as follows.

Rank Finding

1. Find the slot S_i such that the sampling key in that slot, K_i is just smaller than the given key K^* , i.e. $K_i \leq K^* < K_{i+1}$, and set $R = \sum_{j=1}^i P_j$; $1 \leq i \leq n$;
2. Find the leaf page that contains the key value K_i ;
3. Scan siblings to the right until the key K^* is found or a key K has a greater value than K^* is found, counting how many records has been scanned and adding the number to R ;
4. The rank of key K^* is R .

Ranked Key Finding

1. Find the slot S_i such that the cumulative population $P^* = \sum_{j=1}^i P_j \leq R^* < \sum_{j=1}^{i+1} P_j$; for R^* is the desired rank and $1 \leq i \leq n$;
2. Find the leaf page that contains the key value K_i

3. Scan to the right until the R^*-P^* th record is found or no more record can be scanned;
4. The ranked key is the key in the record last scanned;

The effectiveness of the two operations depends heavily on two factors. The first is the number of slots used and the second is the deviation of relative populations between slots. Assuming that the sample sizes among the slots are roughly the same, then the number of slots has a direct impact on the performance since the number of pages needed to be scanned in step 3 of both operations are inversely proportional to the number of slots used. The number of pages between the sampling keys of each neighboring slot is reduced as the number of slots increases. A large number of slots provides better performance. However, a large sample table has disadvantages as well. As the number of insertion or deletion increases, the deviation between each sample size P_i may increase. Moreover, in an n -slots sampling table, there are some difficulties in representing a tree with population less than n .

To ensure a small deviation, a table restructuring scheme is used in which a detection algorithm similar to the one for detecting global imbalance is used. The sampling table is reorganized whenever the following criterion is met.

$$100 \times \left| 1 - \frac{n \times P_i}{\sum_{i=1}^n P_i} \right| > \text{preset percentage} \dots\dots\dots (3.1)$$

for $1 \leq i \leq n$, where n is the number of slots.

The term on the left of the inequality (3.1) represents the percentage deviation of the sample size of the i^{th} slot from the ideal value. The preset percentage to the right of the inequality represents the percentage deviation allowed. The lower the preset percentage, the higher the

probability and frequency for restructuring the sampling table. The restructuring strategy can be best described in two phases. In the first phase, the sample size in each slot is copied to a population table consisting of n slots. The sample size values in the population table are adjusted so that the inequality in (3.1) holds for every slot, and the aggregate changes in relative populations are minimum. The resulting values in the population table will become the new sample sizes in the new sampling table. In the second phase, the population table and the old sampling table are used to construct the new sampling table's keys. Each key can be found by the following procedure. By cumulating all the previous slots' sample size in the new sampling table, the rank of a slot's sampling key can be found. By applying the rank finding operation, the new sampling key value can be obtained.

This table restructuring method is not only used to ensure the efficiency of a statistical query, but also is used when the tree is undergoing pruning or grafting⁵ operations. Since the tree pruning or grafting is either at the leftmost branch or the rightmost branch, one can just subtract the number of data items removed or added by these operations to either end. If a large deviation in the table is found, a table restructuring can be started to reorganize the table.

To be able to represent any tree of population less than n , a sampling table with a variable number of sample slots up to n is chosen. In this way, the size of a sample table can grow or shrink as the number of records increases or decreases. To do so, the following criteria for modifying the number of slots are used to determine when to grow or shrink the sampling table:

⁵The pruning operation *removes* part of the B^+ -tree while the grafting operation *merges* two B^+ -trees into one. The detail definition is listed in Definition 3.1 and 3.2.

I) Condition for table contraction- $m \times P_{min} > \sum_{i=1}^m P_i$, where $m > 2$ (3.2)

II) Condition for table expansion- $\sum_{i=1}^m P_i > m \times P_{max}$, where $m < n$; (3.3)

where m is the number of slots currently in use in the sampling table;

The inequality in (3.2) uses a minimum slot population parameter P_{min} to determine when the average population of all slots falls below the acceptable level. The condition, $m > 2$, for the inequality in (3.2) ensures that at least two slots are presented in the sampling table. Similarly, the inequality (3.3) uses a maximum slot population parameter P_{max} to determine when the average slot's population is above the acceptable level. The condition, $m < n$, in equation (3.3) ensures that the number of slots will not grow out of the maximum possible number of slots.

For table expansion, the slot with the highest population in the population table is split into two slots. If the condition for expansion is still met, the procedure for the slot splitting is repeated. For table contraction, the two adjacent slots with the lowest combined relative population are merged into one slot. As in the case of table expansion, this slot merging process is also repeated until condition for contraction is not met. Then a table restructuring is applied for both table expansion and table contraction to rebuild a new sampling table. Notice that the number of merging and splitting can only be applied at most n times.

The advantages of this approach when compared to the ranked B⁺-tree approach are the following:

- 1) There is some control over the time to update the statistical information, and
- 2) It does not incur a high cost when an update operation is done on the B^+ -tree.

There are two disadvantages of the sampling table approach. Restructuring the sampling table in the worst case may require a scan of every record, thus incurring a high cost. Secondly, compared with the ranked B-tree approach, the costs of both rank finding and ranked key finding operations are higher. Assuming that the frequency of restructuring the table and using those two operations is low, we are better off with the sampling table approach under update operations.

Two possible ways to storing statistical information about the B^+ -tree have been discussed. The sampling table approach was adopted because the cost for maintenance is significantly less than the first one under update operations.

3.1.2 Criteria for Global Imbalance and Strategy for Global Detection

One way to measure the data load imbalance is by comparing the local population on a node with the ideal local population and to see how much it has deviated from the ideal local population. The term *population* here means the population of a node with respect to a single B^+ -tree file, not the population of the entire node. This deviation can be measured by calculating the percentage deviation of the local population from the ideal population. The ideal local population can be defined as the mean population of all the nodes as follows.

$$N_i = \frac{N_G}{n_p} \dots\dots\dots (3.4)$$

where N_i stands for the ideal local population,
 N_G stands for the global population, and
 n_p stands for the number of nodes used.

When the percentage deviation is higher than a certain preset level, the data balancing process is initiated. The value of the preset deviation value allowed becomes the *tolerance level*. The definition for tolerance level and the condition for local detection is as follows:

$$\tau_i = 100 \times \left| 1 - \frac{N_i}{N_G} \right|;$$

Using equation (3.4), we have

$$\tau_i = 100 \times \left| 1 - n_p \times \frac{N_i}{N_G} \right|; \dots\dots\dots (3.5)$$

where
$$N_G = \sum_{i=1}^{n_p} N_i$$

τ_i is the tolerance level for node i ;

$$\text{Condition for imbalance: } \tau_i > \text{preset tolerance value, } 1 \leq i \leq n_p; \dots\dots\dots (3.6)$$

Since the actual global population and local population are not known precisely at any node, the estimated values are used instead. The modified definition of the tolerance level is shown as follows:

$$\bar{\tau}_i = 100 \times \left| 1 - n_p \times \frac{\tilde{N}_i}{\tilde{N}_G} \right| \dots\dots\dots (3.7)$$

The estimated value was adopted for the following reason. From the point of view of a local node, the precise value of the global population is not known at any given time.

so it is estimated. Obviously, statistics about the population on a local node can be collected and maintained at the local node from time to time, so a local node's population is precise. Suppose one of the nodes or an outside host is responsible for the collection of all local nodes' population information to form a global population. From the moment a node requests information from all other nodes to the time that it finally receives all the necessary information, the local population at any nodes may have changed. As a result, the global population that is collected does not reflect the actual global information at any instant in time unless the actual changes are known in advance. From the view point of the control node, which is the sole determinant in global imbalance detection, both the local and global population are only estimated values.

The tolerance level has two effects on the entire system. On one hand, the frequency of data balancing can be controlled by varying the value of the preset tolerance level. A higher preset value implies less chance for detecting data imbalances. A higher tolerance level implies that we allow a greater deviation of data population among the nodes. This may affect the system performance because data is not distributed evenly over the nodes.

To share the workload for imbalance detection, an estimated global population can be used at each local node to self examine the tolerance level. This is done whenever an update is made. A *potential imbalance* appears when the tolerance level calculated exceeds the preset value. The knowledge of the potential imbalance can be shared with the control node by message passing. These scenarios are depicted in Figure 3.3a and Figure 3.3b. Figure 3.3a depicts the situation when a local node i detects a potential global imbalance according to its copy of global population $\tilde{N}_G$ and local population $\tilde{N}_i$. When a potential imbalance message is sent to the control site as illustrated in Figure 3.3b. The control site

updates its own local population value about node i and the global population from $\bar{N}_i$ to $\bar{N}_i'$ and $\bar{N}_G$ to $\bar{N}_G^*$ respectively. The reason for carrying local population $\bar{N}_i'$ is that when the control node has this node's local population information, the control node can update its copy of the local population for node i . This means that the control node can perform another imbalance test on the updated values of both the local and the global populations. This reduces the frequency of unnecessary data collection when a true imbalance does not exist. If the result of the imbalance test is negative, then an updated global population is sent back to the node that detected the imbalance. This is illustrated in Figure 3.3c. The new global population $\bar{N}_G^*$ in the control node will be sent to node i and node i will change its global population from $\bar{N}_G$ to $\bar{N}_G^*$. The effect of an updated global population provides the local node a better estimate for detecting a true potential imbalance. On the other hand, if the imbalance result is positive, a request for local population will be broadcast to every node. This is again illustrated in Figure 3.3d. Another imbalance test for every node will be performed after all the local population information are collected and a new global population is constructed. This time, if the test is positive, the system will undergo data rebalancing, and the control node will broadcast this information to every node. Otherwise, a message of the updated global population will be sent to all local nodes to notify them that no imbalance exists. However, the local copies of the estimated global population are updated. This is similar to the case shown in Figure 3.2c.

Figure 3.3: Different cases for Global Imbalance Detection.

Besides the condition for imbalance described previously, a minimum population requirement on a local node, a *threshold* population, is imposed to ensure that a very small database will not undergo any data balancing. The reason is that the advantage of using more nodes may not justify the cost for data balancing. For example, an empty file is initialized only a specified number of nodes. Only when the global population reaches a certain level, will more nodes be used. Similarly, processing queries in a file with less than n_p pages can be done using less than n_p nodes while achieving the same performance levels. Fewer number of nodes should be used when the global population falls below a certain level. To take these into account, the information about the number of nodes currently in use must be kept in all the nodes and in the control site. The local node can only signal the control node when its local population exceeds the threshold population or

falls below the minimum page size. The condition for imbalance detection for the local and control node can be summarized as follows:

At the control node:

Let n_p denote the total number of nodes available and let $\tilde{N}_i$ denote the estimated population in node i . Let the estimated global population be $\tilde{N}_G = \sum_{i=1}^m \tilde{N}_i$. Assuming that the file during the initial phase expands to utilize all nodes before rebalancing is carried out over all the nodes, the criteria for increasing the number of nodes utilized, m , and for global rebalancing of the tree are given as:

- i) The number of nodes is increased whenever $\tilde{N}_G \geq m \times \bar{N}_{Threshold}$ for $m < n_p$. $\bar{N}_{Threshold}$ is the minimum population for the local node.
- ii) A global rebalancing is initiated whenever

$$\tilde{\tau}_i = 100 \times \left| 1 - m \times \frac{\tilde{N}_i}{\tilde{N}_G} \right| > \tau_{threshold} \dots \dots \dots (3.8)$$

where $\tau_{threshold}$ is the preset percentage,

$$1 \leq i \leq n_p; \text{ and } n_p = m;$$

This condition follows from equation (3.7).

Similarly, when the file undergoes data removal, the file may progressively contract to use less number of nodes. The condition for contraction is stated as follows.

- iii) Contract the file to use m^* nodes, $m^* < m$, from m nodes whenever
- $$\bar{N}_G \leq m \times \lceil k / 2 \rceil \dots\dots\dots (3.9)$$

where k is the order of the B⁺-tree;

and $m > 1$.

Criteria for detecting potential imbalance at local node i :

A potential imbalance exists at a local node i , whenever one of the following conditions holds:

- i) $\bar{\tau}_i = 100 \times \left| 1 - n_p \times \frac{N_i}{\bar{N}_G} \right| > \tau_{threshold}$, where $n_p = m$;
- ii) $N_i > \bar{N}_{Threshold}$ where $m < n_p$;
- iii) $N_i < \lceil k / 2 \rceil$, where $m > 1$.

One remaining question is how to elect the control node among all the nodes. The control node can be either the host computer, or found by a simple election algorithm, e.g., node with lowest id. In the distributed leader election algorithm, all nodes participate in electing a control node among them. This is necessary in case of node failure. Irrespective of the method for selecting the control node, the information about all the nodes passes to the new control node.

Summary

Two possible statistics maintenance systems for storing population information within the B⁺-tree have been described. Both have the capabilities for rank finding and ranked key finding. The sampling table approach is adopted mainly because it is simple and has a lower update cost. The strategy for detecting global imbalance by means of

comparing the percentage deviation of the local population from the ideal population has also been discussed. The frequency of data balancing can then be adjusted according to the value of tolerance level allowed. We will see in the next section how a new balance of data distribution is achieved.

3.2 Balancing Phase

After the control node detects that there is a global imbalance for the B⁺-tree, the file enters a balancing phase. In this phase, the following processes occur before the database is restored:

- a new range table using the knowledge of the global population derived from all local nodes is constructed by the control node;
- tuples are redistributed according to the boundaries of the new range table.

The idea of data balancing is to modify the data distribution among the nodes so that they are nearly equal and that they satisfy the criteria for balanced data distribution given in Section 3.1. After data redistribution, the tree returns to the normal phase of operation so that further imbalance can be detected. The cycle of detection and data balancing enables the system to react to the changes in data distribution.

In order to allow data migration between nodes, we carry out pruning and grafting operations of the B⁺-tree. These are discussed in detail in Section 3.2.1. These two operations enable the B⁺-tree to *move* part of the sub-tree to its neighboring node/s and to *combine* the sub-tree from neighboring node/s. These serve as the primitive operations for data migration. The knowledge of which part of the sub-tree should be moved is derived by

comparing the boundary key values in the new and old range table. The new range table is constructed first. The process for constructing a new range table will be described in Section 3.2.2. The detailed strategy for determining and handling the actual data migration is presented in Section 3.2.3. Some of the records in a tree during the balancing phase may become temporary inaccessible. The reason is that they may be in the process of migrating to another node. The pending queries to be processed are handled differently. The strategy for handling these exceptional cases is investigated in Section 3.2.4.

3.2.1 Concept of Pruning and Grafting

The idea of pruning and grafting in the B^+ -tree, as mentioned before, is to allow data migration between any two B^+ -trees of different nodes. Pruning and grafting are defined as follows:

Definition 3.1

Given a key K^* and a direction Δ , where $\Delta = '\leftarrow'$ or $'\rightarrow'$, a *pruning* operation on node i removes all data items with keys $k < K^*$ if $\Delta = '\leftarrow'$ (or $k \geq K^*$ if $\Delta = '\rightarrow'$), located in the leaf nodes and sends them to the left (or right nodes)⁶.

⁶The data item K^* will be removed in a right pruning operation. The reason is to ensure that node i will always include the data item with its lower bound key value in the range table within the node.

Definition 3.2

Let A and B be two B^+ -trees of the same order k . A *grafting* operation merges the two trees A into B or B into A , to generate a simple B^+ -tree of the same order.

These operations are carried out with the prime objective of preserving the B^+ -tree properties and under the assumption that there are no duplicates within the tree. The grafting operation cannot be applied if the smallest key in one of the trees is not larger than all the keys in the other. Fortunately, with key range declustering, all the keys of a tree in one node are either always larger or smaller than that of the other. The second goal is to ensure that the operations can be done in an efficient manner during data migration. Since the basic idea for a right pruning is similar to that of a left pruning, left pruning is used to illustrate how the pruning operation works. We further assume that the keys are arranged in an ascending order in a B^+ -tree, and the key K^* is used as the pruning key. The key K^* , by Definition 3.1, is termed the *pruning key*.

3.2.1.1 Pruning Algorithm

The pruning algorithm uses three major steps to remove data items from the B^+ -tree and to reorganize it back to B^+ -tree format after the data removal. The first step is to collect information from the B^+ -tree that is used in the later reshaping step. The second step is the removal of data items. The last step uses the information collected in the first to reshape the tree to satisfy the definition of the B^+ -tree. We call this the *rebalancing* process of the B^+ -tree. We describe the second step next and later the third step. The information required to be collected in the first step will become clear after the discussion of the rebalancing step.

3.2.1.1.1 Removal of Data Items

A simple removal of all keys and records in the leaf pages with a specified value K^* will leave the leaf pages empty. A straightforward solution is to remove not only the keys and records, but also the empty leaf pages. However, these empty pages cannot be removed, because they are being referenced by their parent pages. This problem is resolved if the *removal* is carried out *recursively* from the leaf level to the root level. More precisely, the following strategy, based on a post-order traversal, is used to execute the removal:

Data Removal algorithm for left pruning

Assuming h is the height of the tree.

1. Visit the root page and set $i = 1$;
2. Store the current page as C_i ;
3. Find the correct child for the desired key K^* , and increment i by 1;
4. Store the child page as C_i ;
5. Set the current page to be the child found;
6. If current page is not a leaf, repeat step 3;
7. Set the current page to be the root;
8. Push the page visited onto a stack;
9. Set the current page to be the leftmost child of the current page;
10. If current page is not a leaf, repeat step 8;
11. If the stack is not empty
 12. Pop the last page from the top of the stack;
 13. If current page is a leaf
 14. Remove child with key which is less than the desired key K^* ;
 15. Remove the page if the entire page is empty and it is not C_h ;
 16. Remove the link to its left neighbor;

17. Otherwise,
18. Remove the leftmost key if its key value is less than the desired key value K^* ;
19. Remove the leftmost child pointer if the child is not one of the C_i ;
20. Remove current page if it is empty;
21. If there are still keys less than the desired key left
22. Push current page into stack;
23. Push the leftmost child into stack;
24. Go back to step 12.

The purpose of Steps 1 to 6 is to memorize the pages along the access path of the pruning key K^* . The pages along the access path of the pruning key for each level i are denoted as C_i , where $1 < i < h$, and h is the height of the tree. C_1 is the root, and C_h is the leaf page containing the key K^* . These pages are used to detect where the removal will be terminated. Steps 7 to 11 are similar to the initial steps for post-order traversal of the B⁺-tree. The only difference is that the leftmost pages visited are pushed onto the stack to enable a post-order visit and removal. Steps 12 to 24 are the detailed description of the post-order removal. The idea can be illustrated as follows. Pruning removes all pages at each level i , starting from the leftmost page to the left sibling page of C_i . Since there is no link between intermediate pages at the same level, the page visited in Step 7 has to be remembered. In this way, the leftmost sub-tree of the root can be examined and removed first, then the second leftmost and so on. The only difference between a post order traversal and our removal algorithm is that the visiting and removing process stops at the page C_i , rather than the rightmost page in the current level. Step 21 enforces this condition. By memorizing C_i , we are able to detect which child should not be removed. This is done in Steps 15 and 19. The stack is used to store the intermediate pages so that its children can be examined first. The size of the stack will vary, according to the level of the current page.

The maximum size of the stack is at most the same as the height of the tree. Notice that Step 19 can be done in $O(1)$ if we keep track of the level of the current page.

The right pruning is very similar to the left pruning with only two differences. The removal of the sub-tree is started from the leftmost page to the *right* of the access path of the pruning key K^* . This guarantees that the leaf pages are always examined and removed in ascending order of key value despite the direction of pruning. The second difference is that the data item with the pruning key K^* as its key will be removed. The data removal algorithm is given as follows:

Data Removal algorithm for right pruning

1. Visit the root page and set i to be 1;
2. Push the page visited, C_i , into a stack;
3. Find the correct child for the desired key K^* ;
4. Set the current page to be the child found;
5. Increment i by 1;
6. If current page is not a leaf, repeat step 2;
7. Memorize the current page as C_i ;
8. If the stack is not empty
9. Pop the last page from the top of the stack;
10. If current page is a leaf
11. Remove children with keys which are greater than or equals to the desired key K^* ;
12. Remove the page if the entire page is empty and it is not C_h ;
13. Remove the link to its right neighbor;
14. Otherwise,
15. Remove the leftmost key if its key value is greater than or equals to the desired key value K^* ;
16. Remove the leftmost child if it is not one of the C_i ;
17. Remove current page if it is empty;

18. If there are still keys greater than or equals to the desired key left
19. Push current page into stack with the key and child;
20. Push the leftmost child into stack;
21. Go back to step 9.

The cost of data removal depends on the number of pages needed to be read. We show in the following lemma that the numbers of page read or write in disk will be of order $O(\frac{N_d}{k} + h)$, where N_d is the total number of data items to be removed, k is the order of the tree, and h is the height of the tree.

Lemma 3.1

The data removal can be done in $O(\frac{N_d}{k} + h)$ disk accesses provided that h pages can be stored in memory.

Proof:

Suppose we have sufficient memory to hold the pages stored in the stack in main memory, any page that is already read and stored in the stack does not require another page read. The number of pages read will simply be the total number of pages accessed by the data removal algorithm. This is the sum of the number of intermediate pages required in order to access these leaf pages, the number of pages along the access path of the pruning key K^* , and the number of leaf pages which have the key values less than the pruning key. For each intermediate page except the root, we can access at least $\lceil k/2 \rceil$ children, provided that the order of the tree is k . Assuming that the total number of data items to be removed is N_d , the maximum number of intermediate pages and the leaf pages traversed is bounded by the following:

Let N_p denote maximum number of pages visited by the data removal algorithm, and N_d denote the number of data items being removed.

Then,

$$\begin{aligned}
 N_p &= \left\lceil \frac{N_d}{\lceil k/2 \rceil} \right\rceil + \left\lceil \frac{N_d}{\lceil k/2 \rceil^2} \right\rceil + \cdots + \left\lceil \frac{N_d}{\lceil k/2 \rceil^{h-1}} \right\rceil + \left\lceil \frac{N_d}{\lceil k/2 \rceil^h} \right\rceil \\
 &\leq \frac{N_d}{\lceil k/2 \rceil} + 1 + \frac{N_d}{\lceil k/2 \rceil^2} + 1 + \cdots + \frac{N_d}{\lceil k/2 \rceil^{h-1}} + 1 + \frac{N_d}{\lceil k/2 \rceil^h} + 1 \\
 &= \frac{N_d}{\lceil k/2 \rceil} + \frac{N_d}{\lceil k/2 \rceil^2} + \cdots + \frac{N_d}{\lceil k/2 \rceil^{h-1}} + \frac{N_d}{\lceil k/2 \rceil^h} + h \\
 &\leq \frac{N_d}{k/2} + \frac{N_d}{(k/2)^2} + \frac{N_d}{(k/2)^{h-1}} + \frac{N_d}{(k/2)^h} + h \dots\dots\dots (3.10)
 \end{aligned}$$

The first part is a geometric sum, and the equation (3.10) can be rewritten as

$$\begin{aligned}
 N_p &\leq \frac{N_d}{(k/2)} \times \frac{(1 - (2/k)^h)}{(1 - (2/k))} + h \\
 &= \frac{N_d}{(k/2)} \times \frac{k^h - 2^h}{k^h} \times \frac{k}{k-2} + h \dots\dots\dots (3.11)
 \end{aligned}$$

Assuming $k > 2$, then $\frac{k^h - 2^h}{k^h} < 1$, and equation (3.11) becomes

$$N_p \leq \frac{N_d}{(k/2)} \times \frac{k}{k-2} + h \dots\dots\dots (3.12)$$

The term $\frac{k}{k-2} < 2$ as $k > 2$. The above equation becomes

$$N_p = c \cdot \frac{N_d}{k} + h \dots\dots\dots (3.13)$$

where c is a constant.

The total number of pages is then $O(\frac{N_d}{k} + h)$.

#

As an illustration, consider the removal of all records with key values less than 55 from the B⁺-tree in Figure 2.6. Figure 3.4 shows the tree after the removal. Notice that

there is an empty space in the page at each level of the tree as we follow the access path of the key K^* . This certainly violates the definition of a B⁺-tree. To solve this problem, a rebalancing process is needed to restructure the tree back to a B⁺-tree.

Figure 3.4: B⁺-tree after a left pruning by key 55.

3.2.1.1.2 Rebalancing

Observing that the only possible violation of B⁺-tree lies along all the pages in the access path of the pruning key K^* , the rebalancing process can restructure the tree back into B⁺-tree by fixing those pages. This can be done a page at a time, starting from the one at the leaf level. The page restructuring process is very similar to the process of rebalancing when the tree undergoes record deletion, and it happens to have insufficient number of keys in the node to satisfy the B⁺-tree condition. In the simplest case of deletion, the page after the deletion does not violate the definition of B⁺-tree, the keys and pointers may simply be shifted. Similarly, when the remaining children in a page is at least $\lceil k/2 \rceil$, the

empty space in a page caused by the data removal can be adjusted by shifting all the remaining keys and reference pointers to the leftmost end of the page⁷. In this way, the empty space in a page is eliminated. However, when the number of remaining children falls below the required limit, $\lceil k/2 \rceil$, a *page merge* or *rotation* may be needed⁸ after shifting. The idea of a rotation is to borrow children from its right sibling to satisfy the required lower bound on the number of children in a page. The page rotation can only apply when the combined number of children in both pages exceeds the limit of one page. When a rotation is not possible, the current page is merged with the sibling page to become one page. In both cases, the common parent needs to be updated. These operations are very similar to the merging process described in [Knuth 73] except that there is only one possible sibling page for each page that requires modification. The existence of empty space in the corresponding parent page is temporary allowed. The Figure 3.5 and 3.6 illustrate how page rotation and page merges are done.

Figure 3.5 illustrates the process of a page merge when either page P' or P has insufficient number of children, i.e., an *underflow* condition, is detected. The page merge is performed only when a rotation is not possible. That is, when the combined number of children of both P and P' is less than k for an order k B^+ -tree. Notice that the key Y_j originally belongs to the parent page Q is moved down to the page P to serve as a separator key between the child P_i and P'_0 . The pointer P'_j (which points to page P') is also removed from the parent page Q . This may induce an underflow in the parent page Q . As a result, a page merge or rotation will be applied to the parent page Q . Figure 3.6 illustrates the process of how a page rotation is carried out. Assuming that page P is the page with the underflow condition and it contains only $\lceil k/2 \rceil - d$ children. The children P_0 to P'_{d-1} are

⁷No shifting is required for the rebalancing process of a right pruning operation.

⁸There is only one case in which rotation or page merge can not be applied. That is, the current page has no sibling page. This means that the current page is the root after rebalancing.

borrowed from its sibling page P' . The separator key Y_j in the parent page Q is now used to separate P_j and P'_0 , and the separator key X^* is promoted to become the separator for page P and P' . Since there is no removal of children in the parent page Q , no underflow is induced by this rotation of the page P and P' . The parent page Q also lies along the access path of the pruning key K^* , and it is examined and modified in the recursive process of the rebalancing. One important result after the merge or rotation process is that the possible underflow induced by page P has no effect on any other pages except its ancestors.

Figure 3.5: Process of merging two pages P and P'

Figure 3.6: Process of rotation.

However, the page merge or rotation operation requires that the two pages P and P' are adjacent brothers, i.e., that both pages have the same parent. An ordinary page merge or rotation cannot be applied when the current page is the only remaining child of its parent. This is the case when page Q has page P as its only child. To be able to apply the same strategy as before, all the *right* sibling pages from top to bottom level, and the separator keys for all levels are kept in memory. The separator key is defined as the smallest key in the right sibling sub-tree. When a page merge or rotation is needed and a common parent is not available, one could imagine that the parent of the sibling page has a temporary extra space to accept the current page as its leftmost child using the separator key defined. If a page merge is required, all the information on the current page will be merged with the

sibling page, so that the current page can be removed from its parent page. The empty parent page **Q** will be removed in the next iteration of page adjustment. As in the case of ordinary page merge, the removal of the parent page causes the parent page to be empty if it has only one child. This page removal process will continue until either the root is reached or an ancestor page containing more than one child is reached. In that case, an ordinary page merge or page rotation is applied.

Suppose instead of a page merge, a rotation is required, the current page then borrows a certain number of children from its sibling page in order to maintain at least $\lceil k/2 \rceil$ children. One possible scenario is depicted in Figure 3.7a, assuming that the page to be corrected is **P**. Since both the sibling page and the current page do not share a common parent, the separator key used by the rotation process has to be known in advance. The separator in this case is Y^* . In addition to knowing the separator key, the usage of Y^* in a rotation actually induces changes in the value of the separator key between the branch rooted at **R'** and **R''**. The new separator key will be X^* if d children are borrowed from the page **P'**. Therefore, the separator keys with the value Y^* in the previous levels must be modified to X^* . This is to ensure the correct update when the parent page undergoes page merge or rotation. As an example, we will consider applying a non-common parent rotation and merge on page **P** in Figure 3.7.

a) Configuration of the tree before any operation on P is applied. b) Result after a rotation on P and its sibling P' has applied.

c) Result after a page merge between page P and P' is applied.

Figure 3.7: The process of page merge and rotation for pages under different parents.

Figure 3.7a shows a possible configuration before a merge or rotation. Notice that the page Q has page P as its only child. If Q has another child, say P'' , P 's sibling will be P'' and not P' . In this case, an ordinary rotation or a page merge is used. Apart from the condition that P and P' do not share the same parent, the condition for rotation and page merge are the same as that of the ordinary rotation and merge respectively. To allow a rotation or a page merge between pages in different branches, the separator key between the child pointer P_s and P_0^* must be known. The value of the key of the leftmost descendent of the child pointer P_0^* now becomes the separator key between the branches rooted at R' and R'' . The separator keys are given for each level as shown in the table to the right of the diagram. The procedure to find these values will be described later. If a rotation is performed, and page P is assumed to require d children to satisfy the minimum $\lceil k/2 \rceil$ children of an order k B^+ -tree, the resulting tree is the configuration shown in Figure 3.7b. The separator key Y^* now separates P_s and P_0^* , and all the occurrences of the key Y^* are

replaced by X^* . The reason for the replacement is that the new separator for the branch R' and R'' is now X^* instead of Y^* . This enables us to perform another rotation or merge of the ancestors of P if it is required.

On the other hand, if a page merge is needed, the result of such a merge is shown in Figure 3.7c. Here the page P is merged with P' and all the occurrences of Y^* remain. The reason is that the parent page Q has no child after the merge. The separator key Y^* is only useful when a further merge or rotation is required. In this case, all the ancestors of Q up to S' will have only one child except S' , which may have more than one. Once Q is deleted after the merge, the parent page of Q has no child, and it too can be removed. This argument applies to all ancestors except for S' . For S' as the parent page, the page S' is a *common* parent for the sibling page R'' and the underflow page R' . Since R' is an empty page, the pointer to R' and the key Y^* are simply removed from S' . Therefore, there is no need for the key replacement in the table.

In ordinary deletion the root can degenerate if the only two children the root has are merged with each other. The root after the merge will have a single child, which violates the definition of B^+ -tree. A simple solution similar that of the deletion algorithm is to remove the current root and to make its only child the new root. However, the new root may degenerate, so this process may have to continue until either a non-degenerate root is found or a leaf page is reached. If the leaf page is degenerate, the whole tree may actually be removed. In this case, the empty tree is represented by a single empty page.

The page and its descendants after rotation or page merge satisfies the defined properties of a B^+ -tree. The only possible violation will be on the ancestor level along the access path of the pruning key K^* . In this case, the rotation and page merge can be applied

as the process restructures the pages level by level. In the end, the entire tree is transformed to a B⁺-tree again. The cost of the rebalancing process, which is measured in the number of page accesses required, is proportional to the tree height, $O(\log(N_i))$, where N_i is the population of the tree at node i . This is using the following lemma and is based on the assumption that all the sibling pages and the corresponding separator keys are available.

Lemma 3.2

The maximum number of disk accesses required by the rebalancing process at a node i is bound by $O(\log(N_i))$.

Proof:

The number of rotations and merges required is proportional to the height of the tree h , since a rotation and merge of any level does not affect any descendants in the current level. Each rotation and merge only modifies the current page, the parent and possibly the sibling page. A constant number of disk accesses is therefore needed each time we examine a page backward along the access path of the pruning key K^* . The total number of disk accesses is $O(h)$. Since the height of the tree $h = \log_{\lceil k/2 \rceil}(N_i)$, the result $O(h) = O(\log(N_i))$ follows. #

3.2.1.1.3 Information Gathering

The rebalancing process requires that the separator key and the sibling page be known if a page with fewer keys than the minimum has no sibling in its parent. This information can be obtained when we traverse the tree along the access path of the pruning key K . The idea is to find the next level's separator key and sibling page as we traverse

toward the leaf level. The following steps are used to find and store the information about sibling pages and separator keys:

Sibling and Separator keys Finding

1. Visit the root page;
2. Push the page visited into the stack;
3. Find the correct child pointer P_i in the current page using the desired key K ;
4. Find the right sibling page for page pointed by P_i , it should be pointed by P_{i+1} ;
5. If P_{i+1} exists
 6. Store the sibling page pointed by P_{i+1} ;
 7. Store the key K_{i+1} that separate the sibling page and the child page;
8. else
 9. The sibling page is the leftmost child of the sibling page for the parent level;
 10. Store the sibling page;
 11. Store the separator key as the one in the parent level;
12. Set the current page to be the child pointed by P_i ;
13. Push the current page into the stack;
14. If current page is not a leaf, repeat step 3;

Step 6 to 7 are for the case when a common parent can be found. If no common parent exists, there are two possibilities. The first is that the current page is the root, so no sibling page exists. The second is that the sibling page and the current page are on different branches. In this situation, the separator key is the same as the one in the parent level, and the sibling page is the leftmost child of the sibling page in the parent level. An example of sibling page and separator is illustrated in Figure 3.8.

Figure 3.8: Illustration of sibling from different branches.

Suppose both pages, **A** and **B**, are along the access path of the pruning key K^* , the sibling pages for these two pages can be found using the sibling finding algorithm. Their sibling pages are shown in Figure 3.8. The separator key for the page **A** is the key K ; while the one for the page **B** is also K . The case of page **B** illustrates the situation of separator key when the current page and its sibling page are from different branches.

Assuming the stack has sufficient memory to store h sibling pages, the cost for information gathering will take at most $O(h)$ disk accesses according to the following Lemma:

Lemma 3.3

The total cost for information gathering for a particular node i will take at most $O(h)$ page accesses, where h is the height of the tree in node i .

Proof:

In the information gathering algorithm, we search the separator keys and sibling pages level by level starting from the root to the leaf. If a page is along the access path of pruning key K^* , and its right sibling page has the same parent, the separator key is the key that separates the two pages. One extra page access is required to bring the sibling page in and store it in memory. On the other hand, if the sibling page is not located within the same parent, we will locate its parent from the memory in step 7. The separator key remains the same as that of the parent level since they follow different branches. Another page access is required to bring the sibling page in. The total number of page accesses is at most $2h = O(h)$. #

3.2.1.1.4 Cost for Pruning

The cost for pruning in a single node i is examined here. The cost for data removal according to Lemma 3.1 is of order $O(\frac{N_d}{k} + h)$. The cost for rebalancing is $O(\log(N_i))$ according to Lemma 3.2. The cost for information gathering takes $O(h)$ as well according to Lemma 3.3. The total cost for the pruning is then $O(\frac{N_d}{k} + h)$.

3.2.1.2 Grafting Algorithm

Assuming that the data items arrive in the order of the removal from left to right, a B^+ -tree can be constructed in time $O(N_d)$, where N_d is the number of data items moved. The remaining problem is when and how to combine the two trees. We address the second problem by presenting the grafting algorithm. We assume that the keys are arranged in ascending order.

Grafting can be described into two steps. In the first step, the two trees are rearranged into left and right trees according to their key values. Let these be denoted by R_L and R_R respectively, where the one with the smaller key values is the left tree. The heights or the number of levels of the two trees are known. Let these be h_L and h_R respectively. After connecting the leaf pages between these two trees, there are two possible cases of grafting. In the first case, both trees have the same height. This is shown in Figure 3.9a. If a page merge of the roots of both trees is possible, with the separator key being the smallest key in the right tree, the two trees are combined into one single tree by simply merging the roots, as shown in Figure 3.9b. Otherwise, a new root is created with R_L and R_R as its children. The key that separates the children, R_L and R_R , is the smallest key in R_R . This is illustrated in Figure 3.9c. Rotation may be needed, if one of the pages, R_L or R_R , has less than $\lceil k/2 \rceil$ children in a B⁺-tree of order k . The configurations of the tree before and after a page merge and where a new root is constructed are presented in Figure 3.9. We assume that the smallest key value in the tree rooted at R_R is X^* .

a) Two trees rooted at R_L and R_R before grafting.

b) Result of grafting if the total number of children is within k , i.e., $(s+t+2) \leq k$.

c) Result of grafting if the total number of children is more than k , i.e., $(s+t+2) > k$.

Figure 3.9: Possible outcome of grafting two trees of same height.

Suppose one tree has a height higher than the other. The taller tree is assumed to be the one on the right, i.e., R_x . The strategy for grafting is shown below.

Grafting Algorithm

Condition: when trees are not of the same height

1. Set $P \leftarrow R_x$;
2. Scan to a possible common parent: set $P \leftarrow$ leftmost child of P for $h_x - h_L - 1$ times;
3. Insert R_L into P with the separator key X^* .

The separator key X^* is the smallest key in the right tree. The insertion in step 3 is shown in Figure 3.10. This is very similar to the previous case of merging trees of the same height except that it is done on the index page. The tree before the insertion is shown in Figure 3.10 a. If a merge is possible, the result after the merge between page P' and R_L will be as in Figure 3.10 c. If a page merge between R_L and P' is not possible, The pointer to R_L is added to the page P . The result is shown in Figure 3.10 b. Notice that the insertion of the pointer to R_L in page P may cause an overflow in page P , and a rotation may be needed between pages R_L and P' if R_L has less than $\lceil k/2 \rceil$ children. The overflow in page P can be handled by page splitting method used in insertions and is described in [Knuth 73]. As in the case of insertions, the page split may be applied recursively all the way up to the root page R_R .

a) Trees rooted at R_L and R_R before grafting.

b) Result after a rotation is applied.

c) Result after a page merge is applied.

Figure 3.10: Possible outcomes of grafting two trees of different height.

The cost for the grafting is of order $O(h_R)$, where h_R is the height of the tree rooted at R_R , and it is stated as the following Lemma:

Lemma 3.4

The total cost of grafting two trees of height h_R and h_L , $h_R \geq h_L$, is of order $O(h_R)$.

Proof:

The cost, in terms of page accesses, is evaluated as follows. In order to examine the height of the two trees, to find the value of the separator key between the two trees, and to connect the leaf pages between the two trees, h_R and h_L page accesses are required. If one tree is taller than the other, another $h_R - h_L$ page access is needed to find out the insertion page, P . If the root page R_L can be merged with the child of P , i.e., page P' , no update is needed in the parent page P . However, if a rotation between R_L and P' is needed, page R_L is inserted into the page P . This may cause a page split in P . At most another $h_R - h_L$ page accesses is required if the page split affects all the ancestor pages all the way up to the root page R_R . As a result, the total cost, in this case, is $O(h_R)$, since $h_R > h_L$.

On the other hand, if both trees have the same height, either a rotation or a page merge is applied to the root pages, R_L and R_R . A rotation will require the creation of a new root page with R_L and R_R as its children. A page merge will combine the contents of the two root pages and retained this as R_R . These two operations require constant number of page accesses. Therefore, the cost for grafting is denominated by the cost of traversing both trees, which is of order $O(h_R)$. #

3.2.2 Construction of a New Range Table

After the control node receives all the local node's population information and decides to perform global balancing, a new global balanced table is computed by assigning an equal number of tuples to each of the m nodes, where m is the number of nodes currently in use. Since the lower boundary key of the range table can be found easily, the problem now is how to find the upper boundary key values for the new table.

The upper boundary keys in the new range table is computed by first finding the global rank of each boundary key using the newly assigned population of the nodes, and then locating them from the m nodes. The detailed strategy is described as follows.

Determining the Boundary Key Values:

Input: The previous local population for each node $\tilde{N}_i$;

The balanced local population $\tilde{N}_i^*$;

m , the number of nodes currently in use;

1. Global rank finding: the estimated rank of boundary key for node i , K_i , is
$$\tilde{R}_i = \sum_{j=1}^i \tilde{N}_j^*;$$
2. Node locating: key K_i will be located at node j where
$$\sum_{i=1}^{j-1} \tilde{N}_i < \tilde{R}_i \leq \sum_{i=1}^j \tilde{N}_i;$$
3. Local rank finding: the rank at node j for K_i is
$$LR_i = \tilde{R}_i - \sum_{i=1}^{j-1} \tilde{N}_i;$$
4. Key collection: request each ranked key K_i of rank LR_i from node j in descending order.

The requests for the ranked key finding in step 4 must be processed in descending order within a local node. The reason is that during the balancing process, the node that processes the ranked key finding requests may be processing a sequence of deletion operations. In this case, the requested rank may not be available since the population of the local tree falls below the requested rank. In this situation, the requested ranked key is assigned to the key with the highest rank available. However, there is a possibility of having more than one ranked key finding operation, and all ranks requested are higher than the population in that node. In order to maintain a range table with unique boundary keys, some policies concerning deletion and ranked key processing should then be imposed. The first one is to store the *actual rank* of the last ranked key finding operation in a node. A unique returned key value can then be guaranteed by replacing the original ranked key finding request with a rank less than the rank stored, if the requested rank is higher than the *actual rank* found by the previous request. To ensure all ranked key finding operations is answered by the any node in the system, the population of a local node will be maintained at P^* so that n_p ranked key requests can be answered. The minimum value of P^* is n_p as stated in the following Lemma:

Lemma 3.5

The minimum population of a node P^* must be at least n_p in order to construct a range table with distinct boundary key values.

Proof:

To find the minimum value of P^* , we assume that n_p requests are directed to a local node. Recalls that the rank finding operations are processed in decreasing order of the requested ranks, if the actual rank returned by the current ranked key finding operation is

R , the highest rank possible for the next ranked key finding operation, R^* , is then $(R-1)$. The rank requested for the next ranked key finding operation, R' , is replaced by R^* if $R' > R^*$. This is to ensure that distinct boundary key values are found in constructing the new range table. The value of R^* now becomes $(R-2)$. Suppose each rank requested for all the remaining ranked key findings is higher than the value of R^* , then the key returned by the last ranked key finding would have an actual rank of $(R - n_p + 1)$. The smallest positive value for R^* would be 1. In this case, the value of R is n_p . Therefore, the minimum population of a tree must be at least n_p . #

During balancing, there may be many deletion requests waiting for processing when the local tree has a population of n_p . In this case, these deletion operations are simply delayed to guarantee unique boundary keys for the new range table. Since the condition for delaying deletions is valid only when the population reaches the minimum, the chances for postponing operations are minimized.

By adopting these policies and the strategy for ranked key finding described in Section 3.1.1, a new range table with unique boundary keys is built by combining the results in the key finding into the new population table.

3.2.3 Data Migration Handling

After the new range table is constructed in the control node, it is immediately broadcast to every node. A node that receiving the new range table determines what data should be moved and to which node. It also knows which nodes it should receive the data from. This decision is made by comparing the old range table with the new range table. Any record where its key value falling within the boundary values of the node j as defined by the new

range table, is moved to node j . Similarly, a node i receives records from a node j if node j contains some records that belong to node i . The criteria to determine who sends or receives data is summarized as follows:

Using the old range $[K_{i-1}, K_i)$ in node i and the new set of range keys $[K_0, K_1, K_2, \dots, K_m]$:

- The records with key values falling in the range $[K_{i-1}, K_i) \cap [K_{j-1}, K_j)$ at node i will be sent to node j if $[K_{i-1}, K_i) \cap [K_{j-1}, K_j) \neq \emptyset$, for $i \neq j$;
- Node i receives records with key values in the range $[K_{j-1}, K_j) \cap [K_{i-1}, K_i)$ from node j if $[K_{j-1}, K_j) \cap [K_{i-1}, K_i) \neq \emptyset$, for $i \neq j$.

After the data are sent out, the node waits for the data it expects to receive. A new independent B^+ -tree is built for each node it expects to receive data from. All B^+ -trees at the local node will share the same file fragment. When all the B^+ -trees are grafted with the existing one to form a single B^+ -tree, the entire balancing process is completed at the local node. The control node is notified afterward. When all nodes completed their balancing phase, and the control node has received notifications from all the nodes, the balancing is then completed for the entire system.

There are several issues that need to be addressed. The first relates to the order of grafting. This is important because the basic requirement for grafting is that the keys in one tree must precede the keys in the other. An incorrect order of the grafting operations will lead to unsorted key order in the B^+ -tree. The second issue relates to how to build a B^+ -tree for each data stream, assuming that they arrived in the order of sending. The last issue concerns the timing in completing the balancing phase. From the view point of a local

node, the balancing phase is completed when it has completed the sending of data, and has completed the integrating of the data received. Since not all the nodes complete at the same time, we need to synchronize the completion of a round of balancing. This is to ensure that a node will not initiate another balancing phase until the pending one is globally completed.

The simplest method to ensure the ordering of data within a B^+ -tree is to start the grafting of multiple trees for a node i in the following ways:

- Graft with the trees formed by data from node j in descending order, where j is the largest node id which is smaller than i ;
- Graft with the trees formed by data from node k in ascending order, where k is the smallest node id which is larger than i .

In this way, the order of keys in the B^+ -tree can be preserved. The data integration can be done independently of the tree building process. It is possible that data may be still arriving from another node l , while node i is grafting the tree formed by the data from node j .

To build a B^+ -tree, assuming that the messages that store the records are received in order from the leftmost to the rightmost, the data is inserted to the leaf page last accessed. The level above is modified only when a new leaf page is required. This demands that we remember the rightmost page for each level. As a record is inserted to the rightmost leaf page P , two cases can arise. In the first case, there is still room in the page, and the record is simply inserted at the end of the page along with its key value. In the second case, the page is full, and a new leaf page is created. After the record is inserted into the new page along with its key K , a reference to the new page P' is needed from its parent. So the reference pointer for P' and the separator key value between P and P' are inserted into the

rightmost page in the level above. This may cause another page split if this index page is already full. The modified page split can be applied level to level, until an index page above has enough room.

To get a better utilization rate, a page split can be forced when the utilization rate in a page is over the preset percentage. However, in exchange for a better utilization, the definition of the B⁺-tree is violated because the number of children in the rightmost page at each level may be less than required. This is resolved simply by applying a rebalancing algorithm similar to the one already discussed for the pruning operation. The difference is that the rebalancing process for the right pruning is used here, and the only requirement is the information about the left sibling pages for each of the rightmost pages and the corresponding separator keys must be known. Finding the sibling pages is similar to the information gathering step in the pruning operation. The information for all levels can be obtained as follows:

Sibling Page Finding

Assuming the data is from other nodes and is to be used for grafting. Let the rightmost pages from top to bottom level be denoted as $T_1, \dots, T_l$ respectively.

1. Set $i \leftarrow 1$
2.
 - The sibling page for T_{i+1} is the second to last child in T_i if T_i has at least two children and the separator key is the last key value in T_i ;
 - Otherwise, it is the rightmost child of the sibling page in the parent level and the separator key is the same as the one for previous level;
3. Set $i \leftarrow i + 1$ and go to step 2;

To ensure the nodes will be in the same phase of balancing, the control node will simply delay the processing of any potential imbalance signal from the local node. In

this way, the earlier completed local node will have to wait for all the others in order to enter another balancing phase.

3.2.4 Query Processing during Balancing Phase

One side effect during the data balancing is that the records arriving from other nodes are temporarily not available until they are integrated into the local tree. All the single record operations directed to the unavailable regions cannot be processed immediately, and have to wait for the completion of the data integration. For the range queries, part of the queries may be processed, provided that the requested range is also directed to the available range of records. Instead of waiting for the entire range to become available, one possible method to alleviate this problem is to partition the original range queries into two or more sub-queries, so that some of the sub-queries can be answered. Other unprocessed sub-queries are put back to the end of the message queue to be processed later.

Another problem during the data balancing is that queries are directed to the ranges of data which are designated to be sent to other nodes. If the records have not been sent to the designated node, the queries are processed. Otherwise, single record operations are transferred to the designated node according to the new range table. A range query is partitioned and transferred to other node/s according to the following rules:

- For a range query of the form $\langle r_1', r_2' \rangle$, where ' $\langle$ ' = $\{[, (]$, and the boundary values for the new range table is $\{r_0, r_1, r_2, r_3, \dots, r_m\}$, the query is partitioned into $\langle r_1', r_i' \rangle, [r_i, r_{i+1}), \dots, [r_j, r_2']$ for $0 \leq i \leq j \leq m$ if $r_1' < r_i \leq r_2'$;
- The sub-query $\langle r_1', r_2' \rangle$ is processed by node $i-1$, sub-queries of type $[r_i, r_{i+1})$ are processed by node i , and sub-query $[r_j, r_2']$ are processed by node j ;

- For a range query of $\langle r_1', r_2' \rangle$, where ' $\langle$ ' = $\{[, (]\}$, and the boundary values for the new range table is $[r_0, r_1, r_2, r_3, \dots, r_m]$, the query is partitioned into $\langle r_1', r_i' \rangle, [r_i, r_{i+1}), \dots, [r_j, r_2' \rangle$ for $0 \leq i \leq j \leq m$ if $r_1' < r_i < r_2'$;
- The sub-query $\langle r_1', r_2' \rangle$ are processed by node $i-1$, sub-queries of type $[r_i, r_{i+1})$ are processed by node i , and sub-query $[r_j, r_2' \rangle$ are processed by node j .

The idea is similar to the case where part of the range is temporarily unavailable, except now the data is located somewhere else. By applying these rules, the range queries can be transferred to the appropriate nodes for processing.

Summary

Various strategies on how to achieve a new data balance have been described. The control node uses its knowledge on the populations of each local node to construct a new data balanced range table. The new range table is then broadcast to all nodes. By comparing the old and new range table, the local node can tell where the data should migrate to. Efficient algorithms for pruning and grafting operations have also been discussed. These operations enable a fast data integration between the migrating data and the existing data in a B⁺-tree. The cost of balancing depends on the size of the data moved. Various schemes on query processing during balancing have also been described to enhance the performance of the system and to protect the data integrity.

Chapter 4

Simulation Experiments

The performance of a file system can be measured by its response time and the throughput (or number of requests processed per unit time) to a series of requests. Response time to any request consists of the processing time in memory, disk access time, communication and queuing delay. Since the disk access time is much greater than the data processing time in memory, the effect of disk access time on response time is obviously more dominant than that of processing cost irrespective of the size of disk caching used. The only dominant factors are the time spent on disk accesses, communication and queuing delay. The disk access time can be measured by the number of disk read, write, and seek because the amount of data transferred is fixed. This is often measured in units of either a page or disk block. Measuring both the response time and the time spent on disk activities allows us to compare the communication overhead in different strategies. The throughput can be computed by measuring the time used in completing a specific number of requests.

The reason for using the term 'simulation' is twofold. First, queries are usually divided into categories like insertions, range queries, etc., in order to compare and measure the merits of different strategies. This influences the system behavior. As a result, it is different from a real world system which generates and processes a mixture of concurrent requests. Furthermore, the files we generate only have fixed length records and have a specific data type as key field. These factors make the system deviate from a real world

system. The difficulty in analyzing the communication and queuing overhead also makes the comparison on simulation results more attractive.

4.1 Objectives of the Simulations

The following strategies have been implemented for experimentation: key range declustering, hash declustering, round-robin declustering, and the new dynamic declustering strategy. The implementation for the B⁺-tree described in [Otoo 90] was selected to implement the underlying tree structure. In the experimental set-up, only the cases of file insertions are considered, i.e., the records are inserted into the file, but are not deleted. Furthermore, each record is assumed to be accessed with equal probability. The following values are observed when processing any request from a file:

- RT_{Avg} : average response time of a request.
- D_{Avg} : average number of disk access. We consider a disk access as either a disk read or write.
- T_{Com} : time of completing all requests.

An additional value, T_{Bal} , is observed when creating an empty initial file or inserting records into a file. T_{Bal} is defined as the average time a node spends in pruning and grafting. This can be measured by summing the total time each node spends in pruning and grafting and computing the average. It is defined as the following:

$$T_{bal} = \frac{\sum_{i=1}^n \sum_{j=1}^m B_j^i}{n} \dots\dots\dots(4.1)$$

where B_j^i is the cost for pruning and grafting for the j^{th} balancing of node i .

n is the number of nodes.

m is the number of balancing phases;

The value of T_{bal} gives us an idea of how costly the global balancing is. The value of the tolerance level in the dynamic range declustering strategy has a direct impact on T_{bal} . The higher the tolerance level allowed, the higher the cost for balancing, and consequently the higher the value of T_{bal} .

The throughput of the system is defined as the number of completed requests per unit time. For a specific sequence of requests, it can be determined by dividing total number of completed requests N^R by the completion time T_{Com} . The throughput of all strategies under insertions will likely to be the same, except for the dynamic range declustering strategy. The throughput of the dynamic range declustering is expected to be lower than the others because of the balancing cost involved. The tolerance level should also have an adverse effect on the throughput because it allows a greater percentage of data distribution deviation.

Based on an assumption that a request is generated at a local node and is sent to an appropriate node for processing, the average response time can be computed by calculating the sum of the response time of all requests and dividing this sum by the number of requests. That is,

$$RT_{avg} = \frac{\sum_{j=1}^n \sum_{i=1}^{N^j} RT_i^j}{N^R} \dots\dots\dots(4.2)$$

where $N^R = \sum_{j=1}^n N^j$.

n is the total number of nodes used.

RT_i^j denotes the response time of the i^{th} request initiated by the node j .

N^j denotes the total number of requests generated from node j .

The reason for such request generation is to avoid a particular node becoming the bottleneck of the entire system. Each node measures the time between a request generation and completion. The average response time for each strategy is expected to be the same except for dynamic declustering strategy. For a lower tolerance level, the higher balancing process incurs an additional delay in processing insertions. On the other hand, a higher tolerance level has the potential to generate a larger deviation of data load from the expected. This means that the nodes with more data receive and process more requests, which causes an increase in queuing delay. At the same time, the nodes with less data may have very little to process. As a result, the response time may increase as tolerance level increases.

The average number of disk accesses for a record insertion consists of three parts: one for searching for the data page that the record should lie in, the second for inserting the record into the data page, and the third for possible sampling table restructuring. The first part is equivalent to an exact match search in a single tree. The second part may involve a page split, in which a new page is created. In both cases, the number of disk accesses depend on the height of the tree. Since all strategies should distribute the data evenly among the nodes, the number of disk accesses used in those two parts will be expected to be

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

Applied Sciences, Computer Science
 SUBJECT TERM

0984

SUBJECT CODE

U·M·I

Categories

HUMANITIES AND SOCIAL SCIENCES

EDUCATIONS AND THE ARTS

Education 0729
 Art 0377
 Sciences 0900
 Secondary 0378
 Social Sciences 0357
 Science 0723
 Special 0391
 Teacher Training 0399
 Communications 0708
 Tests and Measurements 0413
 Communication 0459
 Vocational 0465

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 0515
 Continuing 0514
 Continuing 0516
 Continuing 0517
 Continuing 0273
 Continuing 0282
 Continuing 0688
 Continuing 0275
 Continuing 0727
 Continuing 0518
 Continuing 0524
 Continuing 0277
 Continuing 0519
 Continuing 0680
 Continuing 0745
 Continuing 0520
 Continuing 0278
 Continuing 0521
 Continuing 0279
 Continuing 0280
 Continuing 0522
 Continuing 0998
 Continuing 0523

General 0679
 Ancient 0289
 Linguistics 0290
 Modern 0291
 Literature 0401
 General 0294
 Classical 0295
 Comparative 0297
 Medieval 0298
 Modern 0316
 African 0591
 American 0305
 Asian 0352
 Canadian (English) 0355
 Canadian (French) 0593
 English 0311
 Germanic 0312
 Latin American 0315
 Middle Eastern 0313
 Romance 0314
 Slavic and East European

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
 General 0770
 Accounting 0454
 Banking 0338
 Management 0385
 Marketing 0501
 Canadian Studies 0503
 Economics 0505
 General 0508
 Agricultural 0509
 Commerce-Business 0510
 Finance 0511
 History 0358
 Labor 0366
 Theory 0351
 Folklore 0578
 Geography 0351
 Gerontology 0351
 History 0578
 General

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
 General 0938
 Criminology and Penology 0631
 Demography 0628
 Ethnic and Racial Studies 0629
 Individual and Family Studies 0630
 Industrial and Labor Relations 0700
 Public and Social Welfare 0344
 Social Structure and Development 0709
 Theory and Methods 0999
 Transportation 0453
 Urban and Regional Planning 0453
 Women's Studies

SCIENCES AND ENGINEERING

AL SCIENCES

Al 0473
 Al 0285
 Al 0475
 Al 0476
 Al 0359
 Al 0478
 Al 0479
 Al 0480
 Al 0817
 Al 0777
 Al 0746
 Al 0306
 Al 0287
 Al 0308
 Al 0309
 Al 0379
 Al 0329
 Al 0353
 Al 0369
 Al 0793
 Al 0410
 Al 0307
 Al 0317
 Al 0416
 Al 0433
 Al 0821
 Al 0778
 Al 0472
 Al 0786
 Al 0760

Geodesy 0370
 Geology 0372
 Geophysics 0373
 Hydrology 0388
 Mineralogy 0411
 Paleobotany 0345
 Paleocology 0426
 Paleontology 0418
 Paleozoology 0985
 Palynology 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 0420
 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
 Atmospheric Science 0798
 Atomic 0759
 Electronics and Electricity 0609
 Elementary Particles and High Energy 0610
 Fluid and Plasma 0752
 Molecular 0756
 Nuclear 0611
 Optics 0463
 Radiation 0611
 Solid State 0463
 Statistics

Applied Sciences 0346
 Applied Mechanics 0984
 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 0794
 Materials Science 0548
 Mechanical 0743
 Metallurgy 0551
 Mining 0552
 Nuclear 0549
 Packaging 0765
 Petroleum 0554
 Sanitary and Municipal 0790
 System Science 0428
 Geotechnology 0796
 Operations Research 0795
 Plastics Technology 0994
 Textile Technology

PSYCHOLOGY

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

2 of/de 2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

approximately the same. For the table restructuring, if a larger number of size and large tolerance value are chosen, the probability of restructuring will be quite small, and the cost should be negligible.

In general, we will expect all strategies to perform slightly better than the dynamic range declustering strategy under insertions.

The simulation runs use the following types of requests to evaluate the performance: exact match and range queries. Range queries include the small, medium, and large range queries. For the exact match query, at most one record can be found in the design since duplicate records are not stored in the B⁺-tree. The average number of disk accesses will be expected to be much higher with round-robin strategy than the others due to its inability to locate a record. For the same reason, the response time for round-robin should be higher, and the throughput should be lower. As in insertion, we expect that the increase in queuing delay will increase the response time and decrease the throughput for the dynamic range declustering strategy with increasing tolerance level.

Let q_r be the number of records found in a range query, let N_G be the total number of records in the file, and let $\bar{N}_p$ be the average number of records in a page. The number of disk accesses for a range query in a B⁺-tree consists of two parts: the cost for initializing the range query, and the cost for reading the data pages within the requested range. The number of data pages is independent of how the data is partitioned, and is bounded by $\left\lceil \frac{q_r}{\bar{N}_p} \right\rceil$. The cost of initializing the range query depends on how the data is partitioned, i.e., how many nodes are used to process the range query. For each node used, the node has to reach the starting point of the range query. Assuming the page utilizations are the

same for different strategies, all strategies should use the same number of disk accesses to read the $\left\lceil \frac{q_s}{N_p} \right\rceil$ data pages for the same range query. The cost for reaching the starting point of the query is similar to that of an exact match query since the lower boundary value has to be searched first. The total number of pages used should be

$$\left\lceil \frac{q_s}{N_p} \right\rceil + m \times h \dots\dots\dots(4.3)$$

where m is the number of nodes used in processing the query,
and h is the average height of the B⁺-tree in all nodes.

The second term in equation (4.3) indicates that as more nodes are involved, the overhead for the range query increases. For both hash declustering and round-robin declustering strategy, the range queries have to be distributed to all the nodes for processing. The overhead in disk accesses grows as the number of nodes used increases. Therefore, both strategies are expected to have a higher number of disk accesses, lower throughput, and larger response time when compared to the key range and dynamic key range declustering strategy. As the value of q_s increases, the time used in processing becomes the dominating factor. We expect the difference in the number of disk accesses between the round-robin and key range strategies to decrease as the range size increases. This situation is expected to be the same between the hash strategy and the key range strategy.

4.2 Simulation Model

These simulations are aimed at studying the effect of the tolerance level on the dynamic range declustering strategy, and comparing its performance with the other strategies under different numbers of nodes.

We assume every page of the tree resides in disk and has to read from a paged memory system. A LRU (least recently used) page replacement policy is used with a 20 slot cache in memory. A page fault occurs if a referenced page is not residing in memory and has to be fetched from secondary storage. To avoid the contention in using only one physical disk in the Intel IPSC/860[®] machine, the tree T_i related to the fragment in node i is in the main memory of processor P_i , and written out to the secondary storage only when all requests are done. In this way, multiple disk units are emulated by memory in each node and no physical disk activities are actually carried out during request processing.

The query requests are generated at each local node and an incoming message queue, that can hold up to 100 queries, is established at each node. A finite sized queue is used to reflect the fact that there is limited memory in each node. Since the files are resided in memory, a single query generator will not generate queries fast enough to all the nodes. This is the case because most of the nodes will be idle and any query that arises will be processed immediately. Consequently, measurements made will be inaccurate. By embedding the query generation within the local node, these problems are avoided, since a node is either processing or generating a request. The introduction of the message queue allows the queries to be processed in a batch fashion, which further exploits the capacity of the system.

To illustrate the effect of the tolerance level under different configurations, two sets of files, file (F1) to (F11) and file (F12) to (F18), are generated. Each of the files in the first set contains 120,000 records distributed over 8 nodes, and each of the files in the second set has 80,000 records distributed over 4 nodes. Each record is a 32 bits integer. Our aim is to test the performance of the dynamic key range declustering strategy using different

levels of tolerance values and using different system configurations. All key values are assumed to be generated under uniform distribution. In the following, τ is the tolerance value of the global file, n is the number of nodes used. The value of τ is defined as the preset tolerance value allowed as in formula (3.8).

$$(F1) \quad \tau = 0.2, n = 8;$$

$$(F2) \quad \tau = 0.5, n = 8;$$

$$(F3) \quad \tau = 1.0, n = 8;$$

$$(F4) \quad \tau = 2.0, n = 8;$$

$$(F5) \quad \tau = 4.0, n = 8;$$

$$(F6) \quad \tau = 6.0, n = 8;$$

$$(F7) \quad \tau = 8.0, n = 8;$$

$$(F8) \quad \tau = 10.0, n = 8;$$

$$(F9) \quad \tau = 12.0, n = 8;$$

$$(F10) \quad \tau = 15.0, n = 8;$$

$$(F11) \quad \tau = 20.0, n = 8;$$

$$(F12) \quad \tau = 0.2, n = 4;$$

$$(F13) \quad \tau = 0.5, n = 4;$$

$$(F14) \quad \tau = 1.0, n = 4;$$

$$(F15) \quad \tau = 2.0, n = 4;$$

$$(F16) \quad \tau = 4.0, n = 4;$$

$$(F17) \quad \tau = 6.0, n = 4;$$

$$(F18) \quad \tau = 8.0, n = 4;$$

The experiments are performed on the 18 files generated above. The exact match query and range query are separately tested. The exact match query is performed by randomly producing 20,000 records in the entire data space for each node. The range query (k_L, k_U) is produced by randomly picking the lower bound k_L and generating the upper bound of the range k_U using the range size. The range size is given as a percentage of the size of the entire data space. With the same boundaries, four different range queries based on inclusion or exclusion of the boundary values can be generated by random selection. The following query files depict the number of queries generated at each node to demonstrate the query performance:

- EQ1 : 20,000 exact match query;
- RQ1 : 1,000 range queries of 1%;
- RQ2 : 5,000 range queries of 0.05%;

Assuming a utilization rate of at least 75%, the RQ2 type of query will match approximately 40 - 60 records, which is within the size of a page. These types of queries are considered as small range queries. Each RQ1 type query will match about 800 - 1200 records, which are about 10 - 20 pages, and are referred to as large range query.

To compare the performance with other strategies, six more files are generated for key range declustering, hash declustering and round-robin declustering strategy. The six files are as follows:

- (F19) : 120,000 key range declustering on 8 nodes;
- (F20) : 120,000 round-robin declustering on 8 nodes;
- (F21) : 120,000 hash declustering on 8 nodes;

- (F22) : 80,000 key range declustering on 4 nodes;
- (F23) : 80,000 records with round-robin declustering on 4 nodes;
- (F24) : 80,000 records with hash declustering on 4 nodes;

The query files EQ1, RQ1, RQ2 are also used for each of the above files. We can then compare their performance under 4 nodes and 8 nodes respectively. The key range declustering strategy is given a perfect data balanced range table to generate the file, so that it may measure how well the dynamic declustering strategy performs.

4.3 Simulation Result

4.3.1 Response Time and Throughput under Insertion

As discussed in section 4.1, the performance of a particular strategy can be measured by the response time and the throughput. Since main memory is used instead of actual physical disk, the average number of disk accesses is measured by the number of page reads or writes in memory. In this way, the actual number of disk accesses made can be observed. The dynamic key range declustering (DRD) strategy allows a data load imbalance when the deviation is within a preset tolerance percentage. The increase in the value of the tolerance level τ increases the deviation in the data load, which leads to a decline in the throughput. Increasing the tolerance level also decreases the frequency of global balancing, which increases the throughput. These two factors counteract each other, but a lower throughput is expected as the tolerance level becomes closer to zero or closer to a very high tolerance level.

Figure 4.1: The average response time of insertion for DRD strategy of different tolerance level (τ) using 8 nodes.

Figure 4.2: The throughput of insertion for DRD strategy of different tolerance level (τ) using 8 nodes.

Figure 4.3: The average number of disk accesses of insertion for DRD strategy of different tolerance level (τ) using 8 nodes.

Figure 4.4: The average response time of insertion for DRD strategy of different tolerance level (τ) using 4 nodes.

Figure 4.5: The throughput of insertion for DRD strategy of different tolerance level (τ) using 4 nodes.

Figure 4.6: The average number of disk accesses of insertion for DRD strategy of different tolerance level (τ) using 4 nodes.

Figure 4.7: The cost of balancing during insertion for DRD strategy of different tolerance level (τ) using 8 nodes.

Figure 4.8: The cost of balancing during insertion for DRD strategy of different tolerance level (τ) using 4 nodes.

The results for the insertions of 120,000 records into the files (F1) to (F11) under the 8-node configuration are shown in Figure 4.1 to Figure 4.3. Measurement of the response time, throughput, and average number of disk accesses per node are illustrated

respectively. The average response time measures the average duration between generating a query and completing the processing of that query. For each query, it includes the time spent in the queue and the time used in processing the query. The throughput ratio measures the number of completed queries per unit time for each node. The total number of disk accesses made for an average node is observed by the value of average number of disk accesses. Each node generates queries at a rate faster than the processing rate of the node. This query generation is only stopped temporarily for a preset time interval when a query is returned to the sender. This situation occurs when the message queue at the receiver node is full. By allowing the query generation rate higher than the processing rate of a node, the processing power of the system can be measured. One side effect of this query generation is that the number of queries waiting in the message queue of any node is always close to the maximum queue size. Consequently, this increases the waiting time in the queue for any query and contributes to a much higher response time.

The results for the insertions of 80,000 records into the files (F12) to (F18) over the 4-node configuration are shown in Figure 4.4 to Figure 4.6. The measurements shown are for similar parameters as in Figure 4.4 to Figure 4.6. A very low throughput value for the file with tolerance level 0.2% when compared with that of 0.5% can be observed from both Figure 4.2 and Figure 4.5. As predicted, a general trend towards lower throughput as we increase the tolerance level can be found in the above two figures. In addition, the following can be observed for the DRD strategy during the insertion:

1. The cost of balancing decreases as the tolerance level τ increases. This pattern can be observed in both Figure 4.7 and Figure 4.8. A higher percentage of the average completion time is spent on pruning and grafting operations when the tolerance level is lower. A lower percentage is observed as the tolerance level increases. This

reflects the fact that the frequency of global data balancing is lower with a higher tolerance level value and higher with a lower tolerance level value.

2. The cost of data balancing during pruning and grafting operations is quite low when compared to the average completion time. From Figure 4.7 and Figure 4.8, the cost for pruning and grafting only contributes to 3% or 4% of the average completion time.
3. The average number of disk accesses required per node to accomplish all the insertion operation does not increase when the tolerance level increases. The average number of disk accesses varies from 46,000 to 55,000 for an 8-node system, and from 49,000 to 52,000 for the 4-node system, as observed in Figure 4.3 and 4.6 respectively. The variation in the number of disk accesses is at most 10%. The main reason why the average number of disk accesses does not increase with the tolerance level is that the tolerance level does not affect the 'average' number of disk access during the insertion. The data load imbalance causes some nodes to take in more insertion requests, which increases the number of disk accesses among these nodes. However, the difference in the number of data items between a node with more data and the one with less may not be sufficient to cause a significant difference in the height of the B^+ -trees stored in these nodes. It can then deduced that the number of disk accesses required for each insertion request is close to a constant, and the increase in the number of disk accesses at one node is offset by the reduction in the number of disk accesses at some other nodes. This explains why the average number of disk accesses does not increase with the tolerance level.

4. The skewness in the data load does not occur to the DRD strategy in the range of tolerance value from 1% to 8%. This is true for the files distributed over 8 and 4 nodes. The value of the throughput remains fairly stable during this interval.
5. The average response time has an inverse relationship with the throughput value. The response time decreases as the throughput value increases, and increases as the throughput value decreases, with only a few exceptions. The reason for such exceptional cases is due to the limited space in the message queue. When an incoming request arrives at a node with no space in its message queue, the request is sent back to the sender to acknowledge the situation. For each bounced back message received, the original sender halts the generation of request for a fixed interval of time, and then sends the message back. During the pruning and grafting operations, any incoming insertion requests are delayed until the operations are completed. This increases the queuing delay of those requests. If the original sender is undergoing data balancing when the returned requests arrive, the returned requests also wait. The queuing delay, associated with fixed size buffer and data balancing, leads to a longer completion time and a less predictable pattern of response time.

The best and worst performances of the DRD declustering strategy for insertions in both 8-node and 4-node configuration are compared with the performance of other strategies, file (F19) to F(24). These are shown in Table 4.1 and Table 4.2 below.

Declustering Strategies	Average number of disk accesses	Throughput (in operations per sec.)	Average Response Time (in sec.)
hash (F21)	48115	5.63	13.7
key range (F19)	47984	5.75	14.5
round robin (F20)	49127	5.71	14.1
best of DRD	46594 (.5%)	5.57 (4%)	22.5 (10%)
worst of DRD	54907 (15%)	4.25 (15%)	32.3 (15%)

Table 4.1: Comparison between different strategies for insertions under 8 nodes.

Declustering Strategies	Average number of disk accesses	Throughput (in operations per sec.)	Average Response Time (in sec.)
hash (F24)	53352	4.77	21.1
key range (F22)	53227	4.69	15.0
round robin (F23)	54151	4.75	22.3
best of DRD	49775	4.77	25.8
worst of DRD	55326	4.38	35

Table 4.2: Comparison between different strategies for insertions under 4 nodes.

We comment on some observations made in comparing the values in each table:

1. The key range declustering strategy, hash declustering strategy and the round-robin declustering strategy have a similar performance in terms of average number of disk accesses, throughput, and average response time. This is as expected from the evaluation in Section 4.1.
2. The dynamic key range declustering strategy has a substantially high response time and a low throughput for all tolerance level values compared to other strategies. The

DRD strategy with tolerance value in the range 0.2% to 10% has a throughput value above 5 as shown in Figure 4.2.

3. The average number of disk accesses remains competitive for the DRD strategy. When the number of disk accesses is compared with all other strategies in Table 4.1 and Table 4.2, the increase in disk accesses, in the worst case, is at most 10%. This verifies the previous observation that balancing, i.e., grafting and pruning, do not contribute much disk access cost to the system.

4.3.2 Response Time and Throughput for Range Queries

For range queries, both sets of test files, (F1) to (F11) and (F12) to (F18), were tested under RQ1 and RQ2 type queries. The result for the 40,000 0.05% range queries under 8 nodes is shown in Figure 4.9 to Figure 4.11, and that for the 20,000 0.05% range queries under 4 nodes is shown in Figure 4.12 to Figure 4.14. The effect of the tolerance level to the DRD strategy for small range queries are summarized below:

Figure 4.9: The average response time of 0.05% range query for DRD strategy of different tolerance level (τ) using 8 nodes.

Figure 4.10: The throughput of 0.05% range query for DRD strategy of different tolerance level (τ) using 8 nodes.

Figure 4.11: The average number of disk accesses of 0.05% range query for DRD strategy of different tolerance level (τ) using 8 nodes.

Figure 4.12: The average response time of 0.05% range query for DRD strategy of different tolerance level (τ) using 4 nodes.

Figure 4.13: The throughput of 0.05% range query for DRD strategy of different tolerance level (τ) using 4 nodes.

Figure 4.14: The average number of disk accesses of 0.05% range query for DRD strategy of different tolerance level (τ) using 4 nodes.

1. The variations in average response time and throughput are quite low when the tolerance level varies within the range of 0.5% to 8%. It shows that the data load imbalance does not have a significant impact on both the average response time and the throughput within this range of tolerance level. However, the file with tolerance level of 0.2%, file (F1) and (F12), does not perform as well as the one with 0.5%, file (F2) and (F13). The reason that they have a slightly lower throughput and higher response time is possibly due to the imbalance distribution of query requests. The query requests are generated by randomly selecting the starting point of the range and calculating the end value by adding a fixed distance to the value of the starting point. For the data item selected as integer, if the starting value is too close to the boundary value of integer, it is rejected and a new value is generated. This implies that there will be less number of queries distributed over last nodes in the system, and more requests for the other nodes. This imbalance in the distribution of requests leads to an increase in completion time, since the completion time depends on the node which finishes last. This explains why the files with a tolerance value of 0.2% are outperformed by those with 0.5%, as shown in Figure 4.10 and Figure 4.13.
2. Under an 8-node configuration, as shown in Figure 4.9, the average response time increases gradually and the throughput declines gradually as the tolerance level approaches 20%. This indicates that the increase in data imbalance has some impact on the performance. A larger data imbalance leads to imbalance in the distribution of requests since the nodes with more data items are more likely to receive more query requests. In this way, the completion time is delayed by these nodes, and a higher response time follows.

3. The variations on the average number of disk accesses is quite stable. The deviation is at most 4% between the lowest number and the highest number for both 4 and 8 nodes. This situation is similar to the situation for insertion. The data imbalance only increases the number of requests directed to those nodes with more data. This can be thought as a transfer of query requests from the node with less data to the one with more. Notice that this will not change the average number of disk accesses because the number of disk accesses for processing a single query in any of the nodes is roughly the same.

For the large range query, 8,000 range queries of type RQ2 are run on test files (F1) to (F11) under 8 nodes, and 4,000 range queries of the same type are run on test files (F12) to (F18) under 4 nodes. The results for the first set of files are shown in Figure 4.15 to Figure 4.17. Figure 4.18 to Figure 4.20 show the results for the second set. We observe the following for the DRD strategy:

1. The throughput value remains quite stable for the tolerance level in the range of 0.2% to 8% in both Figure 4.16 and Figure 4.19. The difference between the highest and the lowest value is 0.02, which is about 3% of the lowest value. In Figure 4.16, the throughput value starts to drop when the tolerance value goes beyond 8%. This reflects the effect of data skewness on the system when the tolerance level value is high.

Figure 4.15: The average response time of 1% range query for DRD strategy of different tolerance level (τ) using 8 nodes.

Figure 4.16: The throughput of 1% range query for DRD strategy of different tolerance level (τ) using 8 nodes.

Figure 4.17: The average number of disk accesses of 1% range query for DRD strategy of different tolerance level (τ) using 8 nodes.

Figure 4.18: The average response time of 1% range query for DRD strategy of different tolerance level using 4 nodes.

Figure 4.19: The throughput of 1% range query for DRD strategy of different tolerance level (τ) using 4 nodes.

Figure 4.20: The average number of disk accesses of 1% range query for DRD strategy of different tolerance level (τ) using 4 nodes.

2. As in the case of the small range queries, we have fairly stable average number of disk accesses for the DRD strategy through out the entire range 0.2% to 20%. The difference between the highest and lowest value are less than 400 for the 4-node system, and 1,000 for the 8-node system.
3. The throughput value does not seem to have a big impact on the response time, as can be seen in Figure 4.15 and Figure 4.18. Apart from the influence of the throughput value, the downward movement of the response time can be explained as follows.

First, we realize that the response time is not equivalent to the actual processing time. The response time of a query is influenced by how long a query has to wait before it begins execution. If the queries arrive at the same rate, the response time should vary as the throughput varies. However, some of the nodes are awakened earlier than the others. Suppose we call the set of early awakened nodes **A** and the rest of the nodes **B**. This means that the nodes in **A** will send requests to the rest of the nodes in set **B** before any node in **B** actually starts sending out query request. To ensure the communication network is not flooded with messages, the generation of requests is temporarily halted if the incoming message queue is full. The rationale behind this is that other nodes will have their message queues filled with requests, and there would be no need to generate a message that only increases the network flow. If more data items are located at the set of nodes in **B**, further generation of requests at these nodes are more likely to be suspended, since their message queues are most likely full. When all requests needed to be generated by the set of nodes in **A** are completed, it is possible that the nodes in set **B** are still generating requests. These remaining requests have a much

shorter queuing delay time since there are more nodes to process them. As a result, the average response time is significantly reduced. This accounts for the downward trend of the response time when the throughput decreases.

This effect is more apparent for a larger range size. The reason is that the request generation can only be done either at the beginning or at the end of processing a request. In the initial phase of the system, a longer processing time for a query causes more messages to wait in the queue, and consequently increases the likelihood of the message queue being full for a node in **B**. A longer processing time also reduces the frequency of checking the status of the message queue. These factors reduce the chance of a request being generated from a node in **B**. A reduction of requests generated from the nodes in **B** means that other nodes in **A** receive fewer requests. The chances of their message queue being full is smaller, and the nodes in **A** continue to send requests to the nodes in **B**. The message queues of nodes in **B** will then continue to be filled. This phenomenon, coupled with a non-uniform distribution of range requests previously described, contributes to the fluctuation of response time for both the small range and large range queries.

A comparison of the performance of the best and the worst case of the DRD strategy with the other strategies is also done for both small range queries and large range queries. The result of small range queries for the 8 and 4 nodes are shown in Table 4.3 and Table 4.4 respectively. The result of large range queries for 8 and 4 nodes are shown in Table 4.5 and Table 4.6.

Declustering Strategies	Average number of disk accesses	Throughput (in operations per sec.)	Average Response Time (in sec.)
hash	77616	0.84	139.2
key range	14385	3.62	16.1
round robin	76930	0.82	133.7
best of DRD	14191 (0.5%)	3.86 (2%)	16.8 (0.2%)
worst of DRD	14532 (0.2%)	3.13 (20%)	21.7 (20%)

Table 4.3: Comparison between different strategies for 0.05% range queries under 8 nodes.

Declustering Strategies	Average number of disk accesses	Throughput (in operations per sec.)	Average Response Time (in sec.)
hash	40604	1.43	72.0
key range	12860	4.21	14.2
round robin	40250	1.45	75.0
best of DRD	12647 (2%)	4.11 (2%)	13.1 (1%)
worst of DRD	13038 (6%)	3.75 (6%)	15.8 (4%)

Table 4.4: Comparison between different strategies for 0.05% range queries under 4 nodes.

Declustering Strategies	Average number of disk accesses	Throughput (in operations per sec.)	Average Response Time (in sec.)
hash	33977	0.31	240.0
key range	20886	0.437	140.2
round robin	33896	0.29	255.6
best of DRD	20916 (2%)	0.433 (2%)	135.3 (2%)
worst of DRD	21590 (20%)	0.334 (6%)	161.0 (8%)

Table 4.5: Comparison between different strategies for 1% range queries under 8 nodes.

Declustering Strategies	Average number of disk accesses	Throughput (in operations per sec.)	Average Response Time (in sec.)
hash	20290	.441	137.5
key range	14482	.558	78.8
round robin	21393	.438	145.1
best of DRD	14370 (6%)	.556 (1%)	74.8 (6%)
worst of DRD	14655 (0.5%)	.547 (0.5%)	88.8(1%)

Table 4.6: Comparison between different strategies for 1% range queries under 4 nodes.

We make the following observations:

1. For both the small range queries and large range queries, the performance of DRD strategy, measured in terms of the throughput, average number of disk accesses, and average response time, closely matches that of the key range declustering strategy.
2. The performance of range queries for both the hash and the round-robin declustering methods are significantly lower than that of the DRD and key range declustering strategies. As predicted in Section 4.1, there is an overhead cost in both the disk accesses and communication, which are reflected in the values measured. Compared with the key range declustering, which usually only localizes the range query to a single node, both strategies have to distribute a range query over all nodes for processing. There are additional disk accesses and processing time in starting a range query. The cost of such overhead is comparatively bigger when the range size is smaller. As the range size increases, the time spent on reading the data pages increases, while the overhead cost per query remains unchanged. For these

reasons, the effect of the overhead cost diminishes as the range size gets larger. The performance of both the hash and the round-robin declustering strategies should match that of the key range declustering strategy as the range sizes increase. As shown in Table 4.3 and Table 4.5, the difference in performance between the key range declustering and the round-robin declustering methods under 8 nodes is reduced as the range size increases from a 0.05% range query to a 1% large range query. This decrease in the difference in performance between the two strategies is also observed under a 4-node configuration, as shown in Table 4.4 and Table 4.6. A similar observation can be made on the performance difference between the key range and the hashing declustering methods under both 8 and 4 nodes.

By comparing the values in Table 4.3 and Table 4.4, and also those of Table 4.5 and Table 4.6, the difference in performance is smaller under a 4 node system. This confirms the fact that overhead cost depends on the total number of nodes used. The cost is reduced when there are fewer nodes.

4.3.3 Response Time and Throughput for Exact Match Queries

For the experiments in processing exact match queries, 160,000 requests were done on a 8 node system using the test files (F1) to (F11), and 80,000 were done on a 4 node system using the test files (F12) to (F18). The results of the 8 node system for response time, throughput and average number of disk accesses are plotted in Figure 4.21, Figure 4.22, and Figure 4.23 respectively. The results of the 4 node system for response time, throughput and average number of disk accesses are plotted in Figure 4.24, Figure 4.25, and Figure 4.26 respectively. We observed the following for the DRD strategy:

Figure 4.21: The average response time of exact match query for DRD strategy of different tolerance level (τ) using 8 nodes.

Figure 4.22: The throughput of exact match query for DRD strategy of different tolerance level (τ) using 8 nodes.

Figure 4.23: The average number of disk accesses of exact match query for DRD strategy of different tolerance level (τ) using 8 nodes.

Figure 4.24: The average response time of exact match query for DRD strategy of different tolerance level (τ) using 4 nodes.

Figure 4.25: The throughput of exact match query for DRD strategy of different tolerance level (τ) using 4 nodes.

Figure 4.26: The average number of disk accesses of exact match query for DRD strategy of different tolerance level (τ) using 4 nodes.

1. A fluctuation of response time within the tolerance level from 0.2% to 8% is observed in both Figure 4.21 and Figure 4.24. This is due to the phenomenon associated with the request generation as described in previous section. The effect of this phenomenon is offset by the effect of data load imbalance, as we increase the tolerance level. This is the case observed where by as the tolerance value is increased from 10% to 20%, the response time also progressively increases. In particular, the highest response time is measured at a tolerance level of 20%.
2. A general declining trend of the throughput is observed as the tolerance level increases. This is illustrated in both Figure 4.22 and Figure 4.25. This indicates that a higher tolerance level contributes to a lower throughput value due to data load imbalance.
3. In observing both Figure 4.23 and Figure 4.26, we find very low variations in the average number of disk accesses as the tolerance level is varied. The difference is at most 1% of the lowest value. This is similar to the situation of processing range queries.

Declustering Strategies	Average number of disk accesses	Throughput (in operations per sec.)	Average Response Time (in sec.)
hash	36834	7.29	10.1
key range	36049	7.38	15.3
round robin	287200	0.85	177.25
best of DRD	36250 (2%)	7.08 (0.5%)	9.21 (4%)
worst of DRD	36548 (15%)	5.73 (20%)	18.2 (20%)

Table 4.7: Comparison between different strategies for exact match queries under 8 nodes.

Declustering Strategies	Average number of disk accesses	Throughput (in operations per sec.)	Average Response Time (in sec.)
hash	18779	6.49	11.8
key range	18778	6.51	8.72
round robin	73566	3.02	42.3
best of DRD	18752 (0.5%)	6.45 (0.2%)	7.2 (1%)
worst of DRD	18884 (6.0%)	6.21 (8.0%)	11.85 (2%)

Table 4.8: Comparison between different strategies for exact match queries under 4 nodes.

Two tables are used to compare the performance of different strategies under exact match queries for both 4 and 8 nodes. Table 4.7 shows the results for the test files over 8 nodes, while Table 4.8 shows the results over 4 nodes. Each of the hash, key range and dynamic key range declustering strategy has similar performance in the disk activities, request response time, and the throughput ratio. This indicates that the DRD strategy closely matches the best one in this category. However, the round-robin declustering performs significantly poorly under exact match queries. Its cost is about 8 times the number of disk accesses of other strategies in a 8 node configuration, costs about 4 times the cost of other strategies in a 4 node configuration. The throughput ratio is only about 1/8 of that of other strategies in an 8 node configuration, and about 1/4 that of the others in a 4-node configuration. From the discussion in Section 4.1, the result is as expected.

Since the round-robin declustering strategy has no way to locate where the searching key is, it has to broadcast the query to every node in the system. This increases the disk accesses and communication cost unnecessarily, which accounts for the poor performance.

4.4 Summary of Results

The following points can be made from the results in Section 4.4:

- The performance of key range declustering strategy is either better than or equivalent to that of the round-robin or hash declustering strategy. This is the case when a good data balanced range table is given.
- The performance of dynamic key range declustering strategy is closely matched with the performance of key range declustering strategy when the tolerance level is below 10%. A slightly higher cost is observed for the insertion operations.
- A low cost of balancing is found during the insertion in both a 4-node and an 8-node configurations. This cost is higher with a lower tolerance level, and declines as the tolerance level increases.
- Increasing the tolerance level has a tendency to lower the throughput in processing all types of queries.
- The response time is not a good measure of performance in our implementation. The response time is affected by the data distribution, queuing delay caused by the network flow, and the protocol used in controlling request generation. It does not show a direct relationship with the system throughput.

Chapter 5

Further Discussion

In this chapter we discuss three issues. The first addresses the use of distributed files to accelerate the computation of the relational algebra. Given that relations are stored as the B^+ -tree indexes on the projection and selection attributes using the declustering method discussed, how do we evaluate selection and projection in such relations? The second is how to handle records with duplicate keys? The last one addresses the issue of balancing the data load when multiple files are used.

5.1 Query Processing under Dynamic Key Range Declustering Strategy

Relational operations are the primitive operations in answering queries for relational databases. Since one of our objectives is to use files to represent relational databases in a parallel environment, the translation of relational operations to the existing operations is now considered. The select and project operations for the relational algebra are investigated in the following sections. We further assume that the indexed attribute is contained in the predicates under both operations.

5.1.1 Select Operation

For the *select* operation, if part of the predicate is based on the indexed variable, it is replaced by a sequence of range queries and exact match queries. Suppose the indexed file is indexed on attribute 'Salary' for the relation EMPLOYEE defined below.

(Employee_Number, Employee_Name, Salary)

Figure 5.1: Definition of EMPLOYEE relation.

The *select* operation with predicate '(Salary > 10000) and (Salary < 20000)', i.e., $\sigma_{(Salary > 10,000) \wedge (Salary < 20,000)}(EMPLOYEE)$, can be translated into the range query (10000, 20000) for the index file on 'Salary'. In general, the following algorithm can be applied to developing a sequence of range queries and exact match queries for a *select* operation:

Translation Algorithm for Select Operation

Assumptions:

Query optimization is assumed to be done prior to processing this algorithm. Let x be the indexed attribute, c_1, c_2 are constants on the attribute x , where $c_1 < c_2$, x_L be the smallest extreme value for the attribute x , and x_U be the largest extreme value for the attribute x .

1. Each group of 'or' ($\vee$) predicate conditions, in the form of $A \vee B$, is partitioned into two sequences of predicates, A and B ;
2. For each group of predicate conditions connected by the 'and' operator ($\wedge$), $A \wedge B$, the common region that all conditions applied is found. The condition is translated

into a sequence of inequalities. The inequalities are then changed into range queries or exact match queries by applying the following replacement for each condition:

$(x > c_1), (x \geq c_1)$:

using the given value c_1 as the lower bound with the upper extreme value of the indexed attribute, x_U , as the upper bound, the corresponding range queries are $(c_1, x_U]$ and $[c_1, x_U]$ respectively.

$(x < c_1), (x \leq c_1)$:

using the given value c_1 as the upper bound with the lower extreme value of the indexed attribute, x_L , as the lower bound, the corresponding range queries are $[x_L, c_1)$ and $[x_L, c_1]$ respectively.

$(x = c_1)$:

the corresponding query is an exact match query on key value c_1 .

$(c_1 < x < c_2), (c_1 \leq x < c_2), (c_1 < x \leq c_2), (c_1 \leq x \leq c_2)$:

they are equivalent to the range queries $(c_1, c_2), [c_1, c_2), (c_1, c_2],$ and $[c_1, c_2]$ respectively.

As an example, the predicate of the select operation in Figure 5.2 will be first translated into a series of inequalities $(20,000 > \text{Salary} > 10,000)$ and $(30,000 > \text{Salary} > 25,000)$. After applying step 2, it is translated into the sequence of range queries, $(10000, 20000)$ and $(25000, 30000)$, on the indexed attribute Salary.

$$\sigma_{(((\text{Salary} > 10,000) \wedge (\text{Salary} < 20,000)) \vee ((\text{Salary} > 30,000) \wedge (\text{Salary} > 25,000)))}(\text{EMPLOYEE})$$

Figure 5.2: A sample select operation.

5.1.2 Project Operation

If no duplicate record is allowed, the project operation on the indexed attribute is just a selection of all the records followed by a sort. Sorting is not required for the key range based strategy, since the B^+ -tree is already sorted on the projection attribute. The project operation is simply translated into a range query with the lower and upper extreme values of the indexed attribute x as its lower and upper bound, i.e., x_L , x_u respectively.

When duplicate records are allowed, not only is every record read, but records with the same values on the attributes of project operation have to be eliminated. For example, if the project operation is on attribute x and y , and the relation has index on attribute x , only the records with distinct values on both x and y are selected. In other word, all records with the same x values must be checked to eliminate any duplicate.

In practice, most of the project operations may involve more than one attribute. For key range declustering strategy, all duplicated records are located in the same node, the comparison can therefore be made on the local node. On the other hand, round-robin declustering strategy requires that we compare duplicate records over multiple nodes. On the basis of the cost of communication, the key range based declustering is clearly more efficient in handling the project operation.

5.2 Handling of Duplicate Records

One way to represent duplicate in the B^+ -tree indexes is shown in Figure 5.3. The records with the same key values are grouped together. If a page is overflowed, the pages are then linked in the level for the data pages. In this way, there is no change in the handling

The diagram illustrates a B-tree structure. At the top is a rectangular internal node. Two arrows point from it: one to a middle internal node below and to the left, and another to a large triangle on the right. A dotted line connects the top node to the middle node. The middle node is labeled "Leaf page" and has two arrows pointing to two leaf pages below it. Each leaf page is a rectangle divided into slots, with an arrow pointing from the last slot to the next leaf page. A large triangle is on the right, with an arrow pointing from the top node to its apex. Ellipses (...) indicate additional nodes and pages.

The sampling table requires unique key values. Since multiple records may share the same key value, expanding the number of slots used may not be appropriate. For example, the entire population of the tree may already fulfill the condition for expansion in (3.3). If the number of slots before expansion is m , and the number of unique keys N_k is also m , an expansion in the number of slots to $\tilde{m}$, $\tilde{m} > m$, would result in at least two sampling slots sharing the same sample keys. This case is rare, but it reveals that additional conditions are required. To eliminate the existence of duplicate sample keys, the number of

unique keys must be at least the same as the number of slots after the expansion. As a result, the following condition is added for table expansion:

Let N_k^i denotes the number of distinct key values in the node i ,
and let $\tilde{m}$ denote the projected number of slots for table expansion, then

$$N_k^i \geq \tilde{m} \dots\dots\dots (5.1)$$

The table expansion requires knowledge of the number of distinct key values. This information is obtained by examining the total number of keys in the tree. Since the number of resulting slots, $\tilde{m}$, for the table expansion can be calculated prior to any modification being done on the table, finding m distinct keys is sufficient to satisfy condition in (5.1). Reading $\tilde{m}$ keys from the leaf level will take at most $\frac{\tilde{m}}{\lceil k/2 \rceil} + h$ pages accesses. The cost is relatively small since the maximum number of slots used is much smaller than the total number of records in the tree.

Duplicate sample keys arise under an alternate situation. It can occur if the condition for restructuring the sampling table in (3.1) is met. One possible case is that the restructuring is caused by too many samples in a particular slot. In this situation, some of the samples from this slot is moved to other slots. However, if there is only one distinct key value in the entire slot, there is a chance for obtaining duplicate sample keys. This occurs when the two new sample keys happen to be located in the slot with only one key. The problem is resolved by replacing the duplicate key with a key of different value but has a rank just lower than the current one. One of the drawbacks of this approach is that the deviations on the number of samples may increase slightly. The replaced sample key may

coincide with the another existing sample key found in the restructuring process. The process for the key replacement is applied recursively until no duplicate key exists.

To find a distinct key with a lower rank, it can be done using the ranked key finding operation with the following modifications. When a requested rank is requested by the ranked key finding operation, the rank of the first occurrence of the desired key, say R^* , is returned along with the key value. In this way, a distinct key with a lower rank is found simply by replacing the key requested in the ranked key finding operation with the rank $R^* - 1$.

The strategy for statistical maintenance is very similar to the strategy for data balancing and data load imbalance detection. To avoid the boundary keys in the range table share the same value when expanding the number of nodes in use, the number of unique keys in the distributed file must be at least the same as the projected number of nodes after the expansion. This additional criterion for expansion, similar to the one for sampling table expansion, is summarized as follows.

Assuming that the global population is $N_k = \sum_{i=1}^n N_k^i$, where

n is the number of nodes before the expansion.

Suppose $\tilde{n}_p$ is the projected number of nodes after the expansion, then

To utilize $\tilde{n}_p$ nodes, we must have $N_k \geq \tilde{n}_p$.

A similar strategy for the handling duplicate sample keys is applied here if some new boundary keys found are the same. There is one major difference. If the duplicate key found on a node i is the same as the lower boundary key value in this node, the location for the replacing key with lower rank is located at node $i-1$.

In constructing a new range table, an equal number of data items is assigned to each node. However, with duplicate records, the data size assignment can not be the average value. This may lead to increase in the frequency for data balancing, and an increase in balancing cost.

5.3 Dynamic Key Range Declustering in Multiple Files Environment

When more than one file is involved in the database processing, the cumulated data load imbalance in multiple files causes differences in disk utilization between nodes. To ensure a better system performance, the disk space available should be considered as well. As a result, our objective becomes that of maintaining the disk space available as close to the average as possible. When there is no conflict with this objective, a data load balance within a file shall also be maintained. These objectives are achieved by modifying the detection-balancing strategy described in Chapter 3. This requires changes in the criteria for imbalance detection, and in the method for assigning data when creating the new range table. These changes are used to reduce the differences in disk utilization between nodes.

The idea of maintaining a balanced disk utilization is achieved by disallowing any increase or decrease of data in a node with higher or lower utilization respectively. This is expressed in the following condition:

$$|\tilde{D}_i - \bar{D}_s| < \delta \dots\dots\dots(5.1)$$

where $\tilde{D}_i$ is the disk utilization on node i ;

n is the number of nodes in the system,

δ is the difference allowed,

$$\text{and } \bar{D}_s = \sum_{i=1}^n \frac{\tilde{D}_i}{n}.$$

To compute this, each node maintains a copy of the estimated utilization of all the nodes. This eliminates potential data imbalance caused by insufficient or excess data at a node, which also has a higher or lower utilization. In this way, an imbalance message is received by the control node only when the data size deviates from the average and the difference in disk utilization is beyond the acceptable limit.

When the control node receives a potential imbalance message, it will request information on the disk utilization in addition to the population information. After collecting the information, the condition (5.1) is tested for all nodes. Regardless of the result of the test, a new global population and an updated copy of the nodes' disk utilization are sent to all the nodes. If a global imbalance appears, a new range table is built with the addition of the following rules:

1. Data must be non-increasing for the node with higher storage utilization;
2. Data must be non-decreasing for the node with lower storage utilization.

To simplify the table construction, we partition the nodes into two sets of nodes. Denoted by X , this is the set of the nodes which satisfy either condition 1 or 2. The

remaining nodes are put into the set **Y**. We only consider data movement within the nodes in set **Y** that cause data balancing. The nodes in set **X** serve as transitional nodes, their function is to pass the data from a node in the set **Y** to another node in the set **Y**. A temporary increase or decrease of data on a node in set **X** occurs during the data balancing phase. After data balancing, the data size in these nodes should be roughly the same. The new range table is constructed by computing the average data size of all nodes in the set **Y**. The range table is computed by assigning this population value to all nodes in set **Y** without changing the population value in any node in set **X**. The data distribution of the nodes in set **Y** after the balancing will have lower deviation. The difference in disk utilization between nodes in set **Y** is also reduced after the balancing. In this way, the difference in disk utilization is gradually corrected.

Chapter 6

Conclusions

In this thesis, we have presented a new declustering strategy based on the B⁺-tree structure. The dynamic key range declustering strategy has the inherent advantages of the key range declustering strategy. Compared with other declustering strategies, it has a good system throughput and response time when processing range and exact match queries. Our simulation demonstrates that the dynamic key range declustering strategy exhibits good performance measures as does the key range declustering with a perfectly balanced range table under different queries. Especially for batched queries processing, both the key range declustering strategy and dynamic key range declustering strategy outperform the hash and round-robin declustering strategies for exact match query, small and large range queries.

The data load imbalance has an adverse effect on the system performance. As revealed by the simulation experiment, an increase in the deviation of the data load decreases the system throughput for all types of query operations. The range table of the key range declustering strategy cannot be updated to reflect changes in the data distribution. As a result, the performance degrades. Our design overcomes this drawback by redistributing the data when the deviation from an average data size is greater than a preset tolerance level. The data redistribution is achieved by using the pruning and grafting operations for the B⁺-tree. The preset tolerance level gives a control over the

frequency and the cost of data balancing. A higher tolerance level decreases the system throughput but reduces the cost of data migration.

The pruning and grafting operations provide the efficient operations that allow fast data migration from one disk to the other. Simulation experiments confirm that the cost of these operations accounts for a small percentage of the overall completion time. However, the update requests are blocked during data balancing. This factor combined with the method for generating insertion requests increase the queuing delay for all update requests. As a result, a lower throughput and higher response time are obtained when compared with other strategies.

A high cost may be involved in order to acquire the information on the data distribution prior to the initialization phase of the file. A simple statistic maintenance based on the sampling table gives us an estimate on the data load information on each node. It also enables the local node to detect any potential data load imbalance, which leads to the detection of data load imbalance by a designated control node. The sampling table also provides some basic operations such as rank finding and ranked key finding. These enable the construction of a new data load balanced range table without a priori knowledge of the data distribution.

The B⁺-tree indexes can be used for processing relational queries. By storing relations as files, different types of relational operations like select and project can be translated into range queries and exact match queries. Files with duplicate keys are also considered. Since a file system may process more than one file at a time, consideration of the disk usage is addressed by allowing the data to move to the disk with more space and to disallow data to move out from a disk with less space.

We believe that the contribution of this thesis is a step forward towards the design of distributed partitioned file organization in where the data distribution is prone to changes. Our experiments are limited to a memory based file system with only up to 8 nodes. To fully comprehend the capacity of the dynamic declustering strategy, further investigation on the dynamic declustering strategy on a larger scale is necessary. Also, a distributed protocol in detecting data load imbalance will be required if the fault tolerance is a major concern. Future works should also include the design and implementation of parallel join based on the new strategy.

References

- [Antos 92] Antonshenkov, G.: *Random Sampling from Pseudo-Ranked B^+ -trees*, Proc of VLDB, B.C., Canada 1992.
- [Bayer 72] Bayer, T., McCreight, E.: *Organization and Maintenance of Large ordered indices*, *Acta Informatica* 1, 3, 173-189, 1972.
- [Boral 88] Boral, H.: *Parallelism in Bubba*, International Symposium on Database in Parallel and Distributed Systems,(1988), 68-71.
- [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 Transactions on Knowledge and Data Engineering, Vol. 2, No.1, March 1990, 4-23.
- [Borr 81] Borr, A.: *Transaction Monitoring in Encompass [TM]: Reliable Distributed Transaction Processing*, Proc. of VLDB, LA., August 1988.
- [Com 79] Comer, D.: *The Ubiquitous B-Tree*, Computing Surveys, Vol. 11, No.2, 121-137, 1979.

- [Cope 88] Copeland, G., Alexander, W., Boughter, E., Keller, T.: *Data Placement in Bubba*, ACM SIGMOD on Management of Data,(1988), 99-108.
- [Dewi 84] DeWitt, D.J., Katz, R.H., Olken, F., Shapiro, L.D., Stonebaker, M.R., Wood, D.: *Implementation Techniques for Main Memory Database Systems*, SIGMOD records,(1984), 1-8.
- [Dewi 90] DeWitt, D.J., Ghandeharizadeh, S., Schneider, D.A., Bricker, A., Hsiao, H., and Rasmussen R.: *The Gamma Database Machine Project*, IEEE Transactions on Knowledge and Data Engineering, Vol.2, No.1, March 1990, 44-61.
- [Ghand et al 90] Ghandeharizadeh, S., DeWitt, D.J.: *Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines*, Proc. 16th International Conference on Very Large Data Bases,(1990), 481-492.
- [Gibs 89] Gibson, G., Hellerstein, L., Karp, R., and Patterson, D.: *Failure Correction Techniques for Large Disk Arrays*, Proc. of ASPLOS III, Boston, MA., March 1989.
- [Hsiao 90] Hsiao, H., DeWitt, D.J.: *Chained Declustering: a New Availability Strategy for Multiprocessor Database Machines*, IEEE Parallel Processing, (1990), 456-465.
- [John 92] Johnson, T., Colbrock, A.: *A Distributed Data-Balanced Dictionary Based on the B-linked Tree*, Proc. 6th International Parallel Processing,(1992), 319-324.

- [Kim 86] Kim, M.Y.: *Synchronized Disk Interleaving*, IEEE Transactions on Computers, Vol.C-35, No.11, Nov. 1986, 978-988.
- [Knuth 73] Knuth, D.E.: *The Art of Computer Programming, Vol 3: Sorting and Searching*, Addison-Wesley Publ., Reading, Mass. 1973.
- [Leland 85] Leland, M.D., Rorrie, W.D.: *The Silicon Database Machine*, Database Machines,(1985), 169-189.
- [Liv 87] Livny M., Khoshafian, S., Boral, H.: *Multi-Disk Management Algorithms*, Proc. ACM-SIGMETRICS Conference, May 1987, 69-77.
- [Mason] Mason, D.: Personal Communication.
- [Matsl 90] Matsliach, G. and Shmueli O.: *Distributing a B⁺-Tree in a Loosely Coupled Environment*, Information Processing Letters, Vol. 34,(1990), 313-321.
- [Otoo 90] Otoo, E.J.: *Efficient Concurrency Control Protocols for B-tree Indexes*, Technical Report, SCS-TR-166, School Of Computer Science, Carleton University, January 1990.
- [Patt 88] Patterson, D.A., Gibson, G., and Katz, R.H.: *A Case for Redundant Array of Inexpensive Disks (RAID)*, Proc. ACM SIGMOD Conference on Management of Data,(1988), 109-116.

- [Pram 90] Pramanik, S. and Kim, M.H.: *Parallel Processing of Large Node B-Trees*, IEEE Transactions on Computers, Vol. 39, No.9, Sept. 1990, 1208-1212.
- [Sal 86] Salem, K.: *Disk Striping*, IEEE International Conference on Parallel Engineering, (1986), 336-342.
- [Salz 88] Salzberg, B.: *File Structures: an analytical approach*, Prentice Hall, 1988.
- [Seeger 91] Seeger, B. and Larson P.: *Multi-Disk B-Trees*, ACM SIGMOD New York, 1991, 436-445.
- [SK 90] Seeger, B., Keiegel, H.: *The Buddy-Tree: an efficient and robust access method for spartial data base systems*, Proc. Conf. on VLDB, (1990), 590-601.
- [Stone 86] Stonebraker, M.: *The Case for Shared-nothing*, Database Eng., vol.9, no.1, 1986.

END

3 1-05-95

FIN