1

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

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

Notice - Notice

Notice - Notice

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.

Ameliorating the Drawbacks of the Grid File

by

Ying Lin

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

Master of Computer Science

**Ottawa-Carleton Institute for Computer Science
School of Computer Science**

Carleton University

Ottawa, Ontario

April, 1993

© Copyright

1993, Ying Lin

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 library - votre bibliothèque

Cher lecteur - Notre bibliothèque

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

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

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

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

ISBN 0-315-84096-X

Canada

Name

Ying Lin

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

Computer Science

SUBJECT TERM

0984

SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0906
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
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

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 0460
Gerontology 0383
History 0386
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
Criminology and Penology 0938
Demography 0631
Ethnic and Racial Studies 0628
Individual and Family Studies 0629
Industrial and Labor Relations 0630
Public and Social Welfare 0700
Social Structure and Development 0344
Theory and Methods 0709
Transportation 0999
Urban and Regional Planning 0453
Women's Studies

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture 0473
General 0285
Agronomy 0475
Animal Culture and Nutrition 0476
Animal Pathology 0359
Food Science and Technology 0478
Forestry and Wildlife 0479
Plant Culture 0480
Plant Pathology 0817
Plant Physiology 0777
Range Management 0746
Wood Technology
Biology 0306
General 0287
Anatomy 0308
Biostatistics 0309
Botany 0379
Cell 0329
Ecology 0353
Entomology 0369
Genetics 0793
Immunology 0310
Microbiology 0307
Molecular 0317
Neuroscience 0416
Oceanography 0433
Physiology 0821
Radiation 0778
Veterinary Science 0472
Zoology
Biophysics 0786
General 0780
Medical

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0905

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0995
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 0380
Obstetrics and Gynecology 0354
Occupational Health and Safety 0381
Therapy 0571
Ophthalmology 0419
Pathology 0572
Pharmacology 0382
Pharmacy 0573
Physical Therapy 0574
Public Health 0575
Radiology
Recreation

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences

Chemistry 0485
General 0747
Agricultural 0486
Analytical 0487
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
Statistics

Applied Sciences

Applied Mechanics 0346
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 System Science 0790
Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

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

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis,
Ameliorating the Drawbacks of the Grid File
submitted by
Ying Lin
in partial fulfilment of the requirements
for the degree of Master of Computer Science

Thesis Supervisor

Director, School of Computer Science

Carleton University

May, 1993

Abstract

The grid file is a symmetric, dynamic and multidimensional data organization scheme which guarantees two disk accesses to locate a record. It is efficient in processing range queries with respect to all the attributes. It can be used as an efficient physical storage scheme for relational and spatial databases. Unfortunately, it exhibits some drawbacks that do not have trivial solutions. First, the use of the conventional array mapping function for the directory requires reorganizing the directory storage at each expansion step. This causes performance degradation in real-time applications. Second, the growth of the directory appears to be non-linear for uniform distributions, i.e., $O(N^\zeta)$, where $\zeta = 1 + (d - 1)/(d \times b + 1)$, N is the number of records, d is the number of attributes and b is the page capacity. When data distribution is heavily non-uniform, the growth of the directory is exponential in the number of pages. Third, arbitrary deletions sometimes generate deadlocked empty regions that are impossible to eliminate as they occur. This thesis offers solutions for resolving these limitations. First the one level extendible directory grid file is presented to avoid reorganizing the directory at each expansion step. Second to offset the exponential growth of the directory, a multilevel grid file is proposed. The deadlock problem is resolved by applying a merging policy of reversing the splitting operation. We describe their design concepts, data organization and algorithms in this thesis. Some experimental results that justify our design are also given. Finally, we discuss how the grid file system stores spatial objects and efficiently performs the traditional relational join operation.

Acknowledgements

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

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

My thanks also go to my parents for their love, education and encouragement in so many years.

Contents

Abstract	iii
Acknowledgements	iv
List Of Tables	viii
List Of Figures	ix
1 Introduction	1
2 Background and Related Work	8
2.1 Concept of the Original Grid File and Its Drawbacks	8
2.2 Two-Level Grid File	15
2.3 Interpolation Based Grid File	16
2.4 BANG File	21
2.5 Nested Interpolation Based Grid File	23
2.6 Multilevel Grid File	25
2.7 Splitting Policies of the Grid File	29
2.8 Summary and Motivation	30

3	Grid File Designs: One Level and Multilevel	33
3.1	Basic Data Structures, Splitting and Merging Policies	33
3.1.1	Scales	33
3.1.2	Directory Organization	38
3.1.3	Data Pages	39
3.1.4	Splitting and Merging Policies	39
3.2	Design of the One Level Extendible Directory Grid File	43
3.2.1	The Directory Mapping Function	43
3.2.2	Expansion and Contraction of the Directory	45
3.2.3	Query Processing Strategy	50
3.2.4	Summary	51
3.3	Design of the Multilevel Grid File	51
3.3.1	Multilevel Grid File Structure	52
3.3.2	Splitting and Merging Operations	55
3.3.3	Query Processing Strategy	56
3.3.4	Two Techniques for Improving Directory Page Utilization . . .	57
3.3.5	Summary	58
4	Implementation and Software Modules	62
4.1	The One Level Extendible Directory Grid File	62
4.1.1	Data Structure	62
4.1.2	Insertion and Deletion Algorithms	64
4.1.3	Query Algorithm	70
4.2	The Multilevel Grid File	72
4.2.1	Data Structure	72
4.2.2	Insertion and Deletion Algorithms	73
4.2.3	Query Algorithm	78
4.3	Cache and Opened Grid File List	80

4.4	Software Modules and Dependencies	83
5	Simulation Experiments	85
5.1	Objectives of the Simulation	86
5.2	Simulation Model	89
5.3	Simulation Results	90
5.3.1	Visualization of the Geometry of Partitions	90
5.3.2	Utilization of Data Pages	94
5.3.3	Utilization of Index Pages	95
5.3.4	Disk Accesses for Insertion	99
5.3.5	Query Performance	100
6	Further Discussion	102
6.1	Methods of Object Storage Using the Grid File System	102
6.1.1	Reducing Object Storage to Point Storage	103
6.1.2	Clipping Method	104
6.1.3	Overlapping Method	105
6.2	An Application for Storing Maps in the Grid File System	107
6.2.1	Storing Base Map Data	108
6.2.2	Storing Layered Data	110
6.3	Grid File Join	111
6.3.1	The Grid File Join Model And Two Major Join Methods . . .	111
6.3.2	An Algorithm for Grid File Join	114
6.3.3	Join of Two One Level Extendible Directory Grid Files	118
6.3.4	Join of Two Multilevel Grid Files	120
7	Conclusions	121
	References	123

List of Tables

5.1	Utilization of Data Pages	94
5.2	Average Disk Access for Insertion of a Record	99
5.3	Summary of Query Performance	101

List of Figures

2.1	Data Structure of the Grid File	9
2.2	A Grid Directory Split By Using a New Boundary	11
2.3	A Grid Directory Split By Using an Existing Boundary	11
2.4	A Spatial Deadlock Caused by Buddy or Neighbor Merging Policy . .	13
2.5	Numbering Grid Cells in the Interpolation Based Grid File	17
2.6	Identifying Page Regions in the Interpolation Based Grid File	18
2.7	Directory Organization in the Interpolation Based Grid File	20
2.8	Partitions of a Non-Uniform Data Distribution	21
2.9	Representation of Page Regions in BANG File	22
2.10	Representation of Page Regions in Multilevel Grid File	26
2.11	Directory Organization in Multilevel Grid File	27
3.1	Binary Tree Structured Scales in the Grid File	34
3.2	Buddy Identification in BTS-Scales	37
3.3	Comparison of Two Splitting Policies	41
3.4	Schematic Storage Layout	46
3.5	Initial Situation of the Grid File	47
3.6	An Example of Directory Expansion	47
3.7	An Example of Directory Contraction	49
3.8	Structure of a 2-dimensional Multilevel Grid File	54
3.9	Limitation in the Balanced Multilevel Grid File	60

4.1	A Cache Structure	81
4.2	Module Dependencies in the Grid File Implementation	84
5.1	Visualization of Partitions of a Uniform Data Distribution	91
5.2	Visualization of Partitions of a Diagonal Data Distribution	92
5.3	Visualization of Partitions of a Circular Data Distribution	93
5.4	The Directory Fan-out in the 2-Dimensional Tests	95
5.5	The Directory Fan-out in the 3-Dimensional Tests	96
5.6	The Directory Page Utilization in a 2-Dimensional Space	97
5.7	The Directory Page Utilization in a 3-Dimensional Space	98
6.1	Approximation: A Technique to Represent Spatial Objects	103
6.2	A Partition of Spatial Objects in the Grid File	104
6.3	Overlapping Method of Storing Spatial Objects in the Grid File	106
6.4	A Multilevel Grid File Model for Base Map Data	109
6.5	Superimposed Method for the Grid File Join	115

Chapter 1

Introduction

The interest in various fields of data processing continues to seek the development of new multidimensional data organizations and access mechanisms. For example, the physical organization of database systems and new applications, such as computer aided design and geographic information system, require data organizations that provide multikey access to records. Multidimensional data organizations are characterized by the following properties:

1. Each record is identified by multiple independent keys and the domain of each key is large and linearly ordered.
2. All keys should be treated symmetrically, i.e., the multidimensional data organization does not favor the primary key at the expense of secondary keys. Therefore queries that involve different keys are processed with equal efficiency.
3. The data is dynamic in the sense that new records are constantly being inserted, deleted and modified. The data organization should be able to store multidimensional data to preserve some order while the database grows and shrinks.

With these characteristics in mind, we focus on the following desirable goals of multidimensional data organizations.

1. Good storage utilization of data page.
2. Good storage utilization of index page. The index should be significantly smaller than the actual data stored. We use the *fan out* to identify the property of data organizations. The *fan out* is defined as the ratio of the number of index entries to the number of data pages. A low *fan out* is expected.
3. Fast exact match search.
4. Good clustering on all keys so that reasonable fast range searches can be achieved.
5. Simple insertion and deletion algorithms -- gradual incremental growth and shrinkage of the data space.

These guiding principles should hold for all data distributions, for all insertion orders and for all query patterns.

In a traditional database, one or more secondary indices or a specialized form of hashing function are used when the queries are allowed on multiple keys. This method suffers from inherent redundancy because access to a record via more than one key involves multiple indices with associated update overheads. This comes as no surprise, since most of the one dimensional organizations depend on a total ordering of the single key value, but natural total orders of multidimensional data do not exist.

There is a selection of data organization methods available for managing multidimensional data. These include the *Hashing Method* [3, 10, 12, 14, 20, 22, 25, 27, 31] and the *Tree Structured Method* [1, 2, 7, 8, 23, 24, 33, 36]. These two classes, however, have a common geometric perspective of the file.

Consider that in a file $F = (r_1, r_2, \dots, r_N)$ of N records, each record $r_i = (k_0, k_1, \dots, k_{d-1})$ is defined by d attribute values. There may be some associated

attribute information corresponding to each record. This is omitted in the subsequent discussion. A geometric interpretation of a file F is as a set of points in a d dimensional space. Each attribute corresponds to an axis in the space and the natural ordered distinct attribute values define discrete points along the respective axes. The presence of a record $r_i = (k_0, k_1, \dots, k_{d-1})$ is denoted by a point at the coordinate point defined by the values of $k_0, k_1, \dots$, and k_{d-1} . Such a representation is referred to as a multidimensional attribute space model U^d of the file [25].

A query can be expressed by specifying a region R called the query region in a multidimensional space. It is convenient to think of a region as a cross product B of intervals $[l_i, u_i]$ ($0 \leq i < d$). An exact-match query requests the retrieval of a point whose coordinates k_i ($0 \leq i < d$) match some d specified values x_i . A partial-match query specifies values x_i ($i \in Q, Q \subset \{0, 1, \dots, d-1\}$) and requests the retrieval of all points that match the specified values in the specified axes in Q . A range query requests the retrieval of all points whose coordinate values lie in given intervals $[l_i, u_i]$ ($0 \leq i < d$).

Given such a file representation, one easily sees that the proposed data organizations for efficient query processing essentially partition the attribute space rectilinearly into rectangular regions or cells. The number of points in a rectangular cell is no more than b , where b is a predefined page capacity. Differences exist in the manner of splitting the space, and in the techniques for storing and retrieving the records whose point images fall in the cells.

In the tree structured methods, the partitioning boundary values of the space serve as discriminating values for the internal nodes of an index tree whose terminal nodes are pointers to the data pages holding the records. Such design methods are exemplified by the heterogeneous K-D tree [1, 2] and Quad tree [7]. Quad tree is the most straightforward d dimensional generalization of binary search tree, but it is impractical because tree nodes become large and contain many nil pointers.

These problems are avoided in K-D trees and binary TRIEs [23], both of which can be thought of as an efficient implementation of the d -dimensional generalization of quad trees. K-D trees share many properties with binary search tree.

Since balancing techniques for one dimensional data are well known (e.g., B-tree), it has been suggested by a number of researchers that multidimensional data be organized by several generations of B-trees. Robinson [33] describes the K-D-B-tree which combines properties of K-D trees with B-trees. It is expected that the multidimensional search efficiency of balanced K-D trees and I/O efficiency of B-trees should be approximated in the new K-D-B-tree. The multidimensional B-trees, suggested in [36], orders the records lexicographically on the key fields, with the more significant attributes placed toward the higher end of the sorting field. The retrieval performance depends on the levels of keys in the multidimensional B-trees, therefore all keys are not treated symmetrically.

The hashing methods with the exception of Dynamic Multipaging [20] are all based on one or the other of two 1-dimensional dynamic hashing, namely: Extendible Hashing [6] and Linear Hashing [18]. The characteristics of Extendible Hashing are: a) no overflow records are organized in a special overflow area and b) a record retrieval is guaranteed in at most two disk accesses. The disadvantage of Extendible Hashing is that the directory size can grow exponentially in the number of record insertions unless the keys are uniformly distributed over the directory space. Therefore, Extendible Hashing may generate a bad index page utilization and high *fan out*. A number of techniques have been proposed in [28, 38] to overcome this problem and also to achieve an order preserving hashing scheme to support range queries. In Linear Hashing, a dynamically changing function is used to compute the address of the data pages. The main advantages of the Linear Hashing function are: a) the size of data file grows and shrinks gracefully; b) there is no index and c) the utilization of allocated storage can be controlled. One major disadvantage of this scheme is that it requires the use of a good collision resolution method in

order to avoid poor record retrieval times. There is a trade-off between the data page utilization and retrieval times.

The above two hashing techniques, however, emphasize only single-key cases. These techniques must be extended to the design of multidimensional data organization. *Interpolation Based Index Maintenance* [3] and *Storage Mapping for Multidimensional Linear Dynamic Hashing* [31] are multidimensional file structures which are derived from Linear Hashing. In a manner similar to Linear Hashing, these methods assign regions to data pages by means of mapping functions and use collision resolution by separate chaining to handle overflow. Both methods are sensitive to non-uniform data distribution because the guarantee of a predefined minimum load factor results in overflow chains that may be long and therefore reduce performance. Their solutions have been proposed in [10, 12]. In order to completely avoid the overflow problem, Multidimensional Extendible Hashing [25] and the Extendible Cell Method [39] were proposed. These are based on Extendible Hashing. As in Extendible Hashing, these two methods are related by the fact that they both maintain directories whose entries point to data pages of corresponding regions of the partitioned attribute space. Collision resolution is not necessary since no overflow can occur. However, non-uniform data distribution may cause the directory to grow at an exponential rate in the number of records. Many entries in the directory point to the same data page. Some solutions of multidimensional extendible hashing have been reported in [25, 27].

The grid file [22] is a symmetric, dynamic and multidimensional hashing scheme that has a guaranteed two-disk access to locate a record. It is efficient in processing range queries with respect to all the attributes. Some implementations of the grid file are discussed in [4, 5, 9, 15, 29, 30]. We will review these in Chapter 2. Most of them are not significant and lead to more complicated searching and updating algorithms. A theoretical analysis of some aspects of the performance of the grid file can be found in [32]. Applications of the grid file for storing and processing

geometric objects are presented in [9, 37]. Additionally, the grid file can achieve better performance on relational join operation ([34, 19]) in comparison with other file structures. This is because the grid file can limit the extent of a relational scan very precisely by simultaneously considering the constraints on all the attributes specified by the query.

Unfortunately, the implementation of the grid file may pose many difficulties. In the following discussion, we assume that d is the number of attributes being used, N is the number of records, and b is the page capacity. First, the use of the conventional array mapping function for the directory requires $O(N^{1-1/d})$ disk accesses for reorganizing the directory at each expansion step ([32]). This causes performance degradation in any real-time application. Second, the growth of the directory appears to be non-linear for uniform distributions, i.e., $O(N^\zeta)$, where $\zeta = 1 + (d - 1)/(d \times b + 1)$ ([32]). When data distribution is heavily non-uniform, the growth of the directory is exponential in the number of pages. Third, arbitrary deletions sometimes generate deadlocked empty regions that are impossible to eliminate as they occur. The goal of this thesis is to ameliorate these drawbacks in the grid file.

The main results achieved in this thesis are:

1. A new mapping function between grid cells and data pages is used to develop the one level extendible directory grid file. It avoids reorganizing the directory at each expansion or contraction step. This improves the insertion and deletion times of the grid file.
2. To offset the exponential growth of the directory, we present a multilevel grid file method which builds a hierarchical grid directory. This method allows linear directory growth and also reduces the disk accesses for insertions and deletions. Range queries can be processed efficiently under this method.

3. The linear scales are replaced by binary tree structured scales. As a result, the process of partitioning the space is explicitly represented.
4. The splitting policy which partitions the space entirely according to data distribution reduces the directory size dramatically.
5. The merging policy of reversing the splitting operation avoids the spatial deadlock problem.

The outline of the rest of this thesis is as follows. We present a summary of the concept of the grid file in the next chapter and introduce a variant of proposed implementations of the grid file. In Chapter 3, we present our designs, the one level extendible directory grid file and the multilevel grid file. A precise description of the algorithms for insertion, deletion and query, and the software modules follow in Chapter 4. In Chapter 5, we report on the performance of implementations of our designs. In Chapter 6, we further discuss the issues of spatial object storage and relational join operation in the grid file system. We conclude and present suggestions for future work in Chapter 7.

Chapter 2

Background and Related Work

This chapter describes the concept of the original grid file and its implementations. It then discusses the drawbacks of these implementations that motivated this thesis work. The solutions presented later are intended to overcome the shortcomings of these earlier approaches.

2.1 Concept of the Original Grid File and Its Drawbacks

The *grid file* is a variation of the fixed-grid method, which partitions a d -dimensional data space from which the data points (or records) are drawn using an orthogonal grid. Its goal is to retrieve records with at most two disk accesses and to handle range queries efficiently. This is done by using a *grid directory* that consists of *grid cells*, where all records in one cell are stored in the same *data page*. Generally, three kinds of data structures are needed to define a grid file structure.

scales: A set of d one-dimensional arrays which define an orthogonal grid. Each boundary in a scale represents a $(d - 1)$ -dimensional hyperplane.

grid directory: A dynamic d -dimensional array whose entries correspond to grid cells.

Figure 2.1: Data Structure of the Grid File

data pages: Data pages that contain the data points that lie in the corresponding grid cell.

To avoid low page occupancies, several grid cells may share a page as long as the union of these grid cells forms a d -dimensional rectangle in the space of records. Such a set of grid cells is called a **page region**. Although these page regions are piecewise disjoint, together they span the data space. Figure 2.1 shows these concepts of the grid file. The dash line in the grid directory specifies a page region consisting of two grid cells. The purpose of the grid directory and the scales is to maintain a dynamic correspondence between page regions in a data space and data pages. The grid directory is likely to be large and must therefore be kept on disk, but the linear scales are small and can be kept in memory. To access a record, the d coordinates of a data point are converted into interval indices without any disk accesses by searching the scales. These indices provide direct access to the correct element of the grid directory on disk, where the page address is located. In the second access, the correct data page is read from disk.

As records are inserted and deleted, grid directory, scales and data pages are modified. An overflow occurs when a record is being inserted into a full data page; an underflow occurs when a record is being deleted from a data page with the predefined minimum number of records. If inserting or deleting a record does not cause the corresponding data page to overflow or underflow, only the data page is modified. If a data page overflow occurs, the grid directory and the scales may have to be modified in two possible ways.

In the case of a page region covering only one grid cell, the grid has to be extended by a $(d-1)$ -dimensional hyperplane that cuts the page region into two. This is achieved by inserting a new boundary into one of the scales and maintaining the one-to-one correspondence between the grid defined by the scales and the grid directory. For instance, if an overflow happens to occur in data page 3 in Figure 2.1 and dimension x was selected as the splitting dimension, this would lead to the new grid directory and the scales in Figure 2.2.

In the case of a page region covering more than one grid cell, we merely need to select an existing boundary enclosed by the page region and adjust the mapping between grid cells and data pages. For instance, an insertion in the leftmost lower grid in Figure 2.2 creates an overflow but does not modify the embedding. The next grid directory and the scales are represented in Figure 2.3.

No matter what page region is originally covered, either one grid cell or more, a new data page is attached to one of the two page regions by updating the grid directory, and the records stored originally in the overflowed page are distributed between the new page and the existing one. During update of the grid directory, we have to choose the dimension on which to split and decide on the interval of the chosen dimension to split. This is called the *splitting policy*. In most implementations of the grid file, the dimension is selected in a cyclic order and the domain

Figure 2.2: A Grid Directory Split By Using a New Boundary

Figure 2.3: A Grid Directory Split By Using an Existing Boundary

of each dimension is divided into equal intervals. In fact, there are several policies which are compatible with the grid file. Different splitting policies result in different refinements of the grid partition and performances of the grid file.

The counterpart of splitting is merging. There are two possible levels where merging is appropriate: *page merging* and *directory merging*. Page merging arises when a data page is empty or nearly empty. Page merging policy is influenced by two factors. First, we must decide which page pairs are candidates for merging. This decision can be based on either a buddy system or a neighbor finding system. In a *buddy system*, each page, say X , can be merged with exactly one page in any of the d dimensions, provided the resulting page region retains the shape of a multidimensional rectangular box. Ideally, the chosen page, say B_j , should have the property that at some earlier point it was split to yield pages X and B_j . This buddy is called the *true buddy*. In a *neighbor system*, each page can be merged with either of its two adjacent neighbors in any of the d dimensions, provided the resulting page region again has the shape of a multidimensional rectangular box. Either neighbor system or buddy system may lead to the so-called *spatial deadlock*, i.e., pages are generated which prevent each other from merging since their resulting page region would not be rectangular (see Figure 2.4). The details can be found in [9, 22]. In order to prevent such spatial deadlock, the additional checks have to be performed and for this reason, performance is decreased. A method simpler than the neighbor or buddy system can be derived from a given static splitting strategy as in the multidimensional extendible hashing method in [25]. In order to support such a static splitting strategy, merging must be done as the inverse of the splitting operation. This method achieves a simpler algorithm at the expense of decreasing the average page occupancy.

The second factor that influences the page merging policy is the merging threshold: when should a pair of data pages actually be merged? It should be clear that the sum of the page occupancy percentages for the contents of the merged page

Figure 2.4: A Spatial Deadlock Caused by Buddy or Neighbor Merging Policy

should not be too large, otherwise it would soon have to be split. Nievergelt et al. in [22] suggest that the average page occupancy of 70% is an appropriate merging threshold for the occupancy of the resulting page.

Directory merging arises when all the page regions in two adjacent cross-sections in the grid directory are associated with the same page. Generally, directory merging is of little practical interest since even if merging is allowed to occur, it is probable that splitting will soon take place. Nevertheless there are occasions when directory merging is of use.

The idea behind the design of the grid file is conceptually very simple, but its implementation exhibits some drawbacks.

1. If the splitting policy follows a fixed scheme that does not consider the data distribution, then average data page utilization may drop to a low percent in some cases as indicated in [22].

2. Another problem associated with the grid file is the utilization of the grid directory. For each directory splitting, a $(d - 1)$ -dimensional array is added to the directory. Among these newly added entries, only one of them is updated, all the others are copied in order to maintain the correspondence between the scales and the grid directory. When data are not uniformly distributed, the directory will contain a large number of duplicate entries. A high *fan out* of the directory arises.

If the splitting policy chosen splits an overflow page at the midpoint of an interval and in a cyclic dimension order, it may cause some useless splits since it does not consider data distribution in the data page. The consequence is that the directory is expanded unnecessarily. In the worst case, the growth of the directory is an exponential function of the number of records.

3. The grid file guarantees a bound of two disk accesses for an exact match search. Nevertheless, performance may degrade severely for range queries. For example, to process a partial match query, an entire hyperplane of the directory orthogonal to the domains of the attributes specified in the query has to be examined first. Second, we may encounter a number of false drops in accessing data pages. We have a false drop whenever a directory entry selected by a query leads to a data page that does not contain any record that satisfies the query predicate. Because the directory has some duplicate elements, it is possible to increase the number of false drops if the data pages pointed to by these elements do not contain qualifying records.
4. The performance of insertion and deletion is mainly decided by the number of disk accesses when splitting or merging a grid directory. Because the directory is a multidimensional array, splitting and merging are accompanied by copying the entire directory, i.e., reorganizing the directory. The directory is stored on disk. Therefore, reorganizing the directory would cause a large number of disk accesses.

5. There are also some specific performance issues associated with the two stages of merging; merging of data pages and merging of hyperplanes of the directory. For merging data pages, since the grid directory has a plain array structure, it does not provide a convenient data structure to recognize a buddy or a neighbor that satisfies the d -dimensional rectangular box properly. Hence, algorithms for detecting a buddy or a mergerable neighbor must be executed each time a data page is merged. Frequent execution of these algorithm will be very costly, especially when the directory is stored on disk. There are two problems associated with merging hyperplanes. First, as in splitting, merging the directory necessitates copying the entire directory. Second, to merge two sides of a hyperplane, we must first determine whether they are *mergerable*. To be mergerable, each directory entry in one side of a hyperplane must point to the same pages as the corresponding entry in the other side of the hyperplane. Detecting this condition is costly because a large number of directory entries have to be searched.

2.2 Two-Level Grid File

The first known attempt to realize a practical implementation of the grid file structure was by Hinrichs [9]. In this implementation, a set of linear scales is used, one for each dimension, to define the positions of the grid partitions in the data space. There is a directory entry for each grid cell. The directory is stored on a contiguous sequence of disk blocks, but is logically organized as a linear array of pointers to the corresponding disk pages in the data file. The mapping between the set of scale values defining the coordinates of a grid cell and the corresponding directory element is from a d -dimensional to a one dimensional array. The row (or column) major order mapping is used in this implementation. Each data page in the data file contains the records lying within one page region of the data space. The implementation basically follows the idea introduced in the original grid file. To deal with the bad effect in the grid directory which we discussed in the last section, a

two-level organization was suggested. The first level of the directory consists of the root directory, which is organized like the grid directory. The components of the root directory refer to subdirectories organized also as grid directories, where a subdirectory is exactly in one page. The components of the subdirectories point to data pages. If the root directory is kept in main memory, the two-disk-access principle for an exact match search is also realized by the two-level grid file. Nevertheless, the asymptomatic behavior of the two-level version is the same as the original one-level version. When the amount of data becomes very large, the root directory may grow up to the size which can not fit in memory. The paper does not address how to handle this situation.

2.3 Interpolation Based Grid File

To reduce the size of the directory and associated high *fan out*, the interpolation based grid file [29] uses a representation of the data space in which only page regions are stored in the grid directory. As a result, the number of directory entries is linearly proportional to the number of data pages. To achieve this, a mechanism is proposed to identify each page region so that all implicit partitions can be represented by the page regions in which they are embedded. There are three major modifications in the interpolation based grid file.

1. **Partitioning methodology and storage mapping function:** In the original grid file, splitting a grid directory is done by using a $(d - 1)$ -dimensional hyperplane to cut the page region into two. However, in the interpolation based grid file, in order to enforce a systematic numbering of all grid cells in the order in which they are created, the actual splitting of a grid cell triggers the implicit and linear splitting of all grid cells in the data space. The purpose of implicitly splitting unconcerned grid cells has no other effect but to support the systematic numbering of grid cells. For example, splitting of grid cell 3 in Figure 2.5 (a), will cause the splitting of grid cells 0 and 2. The result is shown in Figure 2.5 (b). The method of systematic numbering of grid cells follows

Figure 2.5: Numbering Grid Cells in the Interpolation Based Grid File

the order-preserving dynamic hashing mapping function [29] which defines a mapping from a set of indices of a grid cell to the grid cell number. It requires that each index be represented as a binary string as shown in Figure 2.5. If the index of a grid cell is (01, 1) in a 2-dimensional data space, the grid cell number is 6 (110) by the order-preserving dynamic hashing mapping function.

It is essential in the interpolation based grid file, that the dimensions be split in a fixed cyclic order. The domain D_j of each dimension j , $0 \leq j < d$, is divided equally into intervals by binary partitioning. Let l be the number of times the whole data space has been split, and let l_j be the number of splits that occurred along the j th axis of the data space, where $0 \leq j < d$. Based on the cyclic splitting policy, l can be expressed as $L \times d + r$ implying $l_j = L + 1$ splits occurred along axes j such that $0 \leq j \leq r$ and L splits along axes j such that $r < j < d$. Now let us see the storage mapping function which maps a given record to the grid cell number where the record falls.

Given a record $k = (k_0, k_1, \dots, k_{d-1})$, the relative position of the j th component, k_j , in the ordered domain D_j is given by

$$x_j = \frac{k_j}{|D_j|}$$

Figure 2.6: Identifying Page Regions in the Interpolation Based Grid File

A set of indices $(i_0, i_1, \dots, i_{d-1})$ represents the coordinates of the grid cell in the data file where record k is stored and the index values are given simply by

$$i_j = \lfloor x_j \times 2^{l_j} \rfloor, \text{ here } 0 \leq j < d$$

Then i_j can be converted to the binary representation, which is applied to the order-preserving dynamic hashing mapping function to obtain the grid cell number.

2. **The method of identifying page regions:** A page region is identified by a unique pair of numbers (r, l) , where r is the grid cell number and l is the space partition level. In Figure 2.6. we assume that the data page capacity is 2. The number at the left upper corner of a grid cell is the grid cell number.

Initially a page region $(0, 0)$ represents the whole data space since the data space contains only 2 records. An overflow occurs when the third record is inserted. It is resolved by splitting on the first level. Two page regions identified by $(0, 1)$, $(1, 1)$ respectively are produced corresponding to two data pages. An additional insertion causes page region $(1, 1)$ to overflow thereby making another split necessary. The distinction between implicit and explicit partitions is illustrated in Figure 2.6 (c), the one on the left shows the implicit splitting whereas the one on the right represents the explicit partition. Three page regions $(0, 1)$, $(1, 2)$, $(3, 2)$ are stored in the grid directory. After a few more insertions, Figure 2.6 (d) shows that implicit grid cells 2, 4, 6 are all embedded in page region $(0, 1)$ and implicit grid cell 5 is embedded in grid cell $(1, 2)$. The grid directory contains $(0, 1)$, $(1, 2)$, $(3, 3)$ and $(7, 3)$.

Since only page regions are stored in the directory, it must be insured that there is a way to identify a page region which embeds a given grid cell. It is observed that if a page region is split, the new page region assumes a number equal to the number of the page region being split plus 2^l , where l is the splitting level. Therefore, given a grid cell number (r, l) , all page region which possibly embed the grid cell are given in the sequence of

$$(r, l), (r - 2^{l-1}, l - 1), \dots, (r - \sum_{i=l-k}^{l-1} (2^i \times b_i), l - k), \dots, (0, 0)$$

where, $b_{l-1}b_{l-2}\dots b_0$ is the binary-bit representation of the integer r . The first page region which is found in this sequence is the page region which represents the correct data page in the data file.

3. The implementation of the grid directory:

The grid directory holds a set of page regions. How this directory is structured is the key to determining the performance of the interpolation based grid file. The number of each page region can be represented by (r, l) , where $r = (r_0, r_1, \dots, r_{d-1})$ and r_i ($0 \leq i \leq d - 1$) is the binary coordinate in

Figure 2.7: Directory Organization in the Interpolation Based Grid File

each dimension. In the above example, Figure 2.6 (d) includes four page regions. The page region number r can be represented as $(00,0)$, $(10,0)$, $(10,1)$ and $(11,1)$. The presentation of the directory can now be viewed as one of storing records whose components are the coordinates of the page regions in the data file. The same technique of organizing the records in the data space can be used in organizing the directory. Figure 2.7 represents the directory organization for the data space in Figure 2.6 (d). The advantage of having an identical structure for the directory and the data space is that the direct access mechanism, for both the directory and the data file is the same and any optimization technique devised for the data file applies to the directory and vice versa.

The main advantage claimed for the interpolation based grid file over the original grid file is that its directory is relatively well behaved with non-uniform data distribution. However, when data in a page region overflows, there is no choice in the subsequent splitting sequence, which may create empty pages. For example, the partitions would be as shown in Figure 2.8 even if all the data were confined to region $(23, 5)$. All the other regions would be empty or nearly empty. If each page region is assigned to a page, the storage utilization would still be very low, and the directory would be filled with entries for sparsely populated pages.

Figure 2.8: Partitions of a Non-Uniform Data Distribution

Note also that the strategy for merging page regions is pre-determined completely. Fundamentally, both the two-level grid file and the interpolation based grid file use the same strategy: merging is restricted to buddies. The interpolation based grid file method for numbering the page regions dictates that a page region can only merge with the buddy from which it was originally split. The adaptive ability of the interpolation based grid file is therefore strictly restricted. This will tend to reduce the average utilization of data pages in the interpolation based grid file.

2.4 BANG File

The BANG file [5] was developed to improve the storage requirement of the interpolation based grid file, i.e., to resolve the problem in Figure 2.8. The interpolation based grid file splits a page region into a number of d -dimensional rectangular page regions of possibly different splitting levels and therefore of unequal size. But in the BANG file, it is no longer necessary to impose the restriction that all page regions must be rectangular boxes. This induces a new kind of page regions which are obtained from a d -dimensional rectangular area from which, possibly, some d -dimensional rectangular holes are carved out.

Like the interpolation based grid file, splitting a page region occurs whenever

Figure 2.9: Representation of Page Regions in BANG File

the addition of a record causes the corresponding data page to overflow. The splitting algorithm essentially partitions the overflow page region into two page regions according to the partitioning methodology and the splitting policy introduced in the interpolation based grid file. The difference between the BANG file and the interpolation based grid file exists in the relationship of these two newly-formed page regions. In the interpolation based grid file, these two page regions intersect whereas in the BANG file, one of these page regions completely encloses the other. Figure 2.9 shows that a page region which represents the whole space is split into two page regions $(0, 0)$ and $(1, 1)$. Page region $(1, 1)$ alone defines the subspace S_2 , but page regions $(0, 0)$ and $(1, 1)$ are necessary to define the subspace S_1 . In general, every subspace of the BANG file is defined by a set of page regions. One member of the set represents a rectangular space which encloses all the other members of the set. These enclosed page regions represent subtractions from the enclosing page region.

The procedure of splitting an overflow page region is recursive. If one of these two page regions into which the overflow page region has been divided contains more records than the other, the page region with a large population is progressively halved until the balance of the number of records in the two pages is reserved. During this process, the best balance is chosen. The balancing technique guarantees that there are never any empty data pages. Figure 2.8 shows the creation of a sequence of partitions as the records are inserted. In the interpolation based grid file,

six page regions are needed to represent the whole space. However, in the BANG file, only two page regions $(0, 0)$ and $(23, 5)$ are needed. This shows that the objective of improving storage utilization of the interpolation based grid file in the BANG file is largely attained.

Another advantage associated with the BANG file is that the merge algorithm is more flexible in responding to changes in data distribution. If the population of a page region sinks below some predefined minimum, an attempt is made to merge it with one of the page regions which it immediately encloses, starting with the smallest. If it fails, an attempt is made to merge the page regions with its buddy. If this also fails, a final attempt is made to merge it with its immediately enclosing region which must always exist. It is not clear from paper [5] how the directory is organized and what the performance is.

2.5 Nested Interpolation Based Grid File

Ouksel in [30] proposed the nested interpolation based grid file to resolve the problem associated with the implementation of the directory structure in the interpolation based grid file. In order to apply the B-tree structure to the directory structure while preserving the spatial orders among all page regions, two types of order relations are defined first.

Denote by S the set of page regions stored in the directory at some point in time and let (r, l) be a page region in the directory. Then the number of page regions enclosing (r, l) in the directory is called the nesting depth of the page region (r, l) . If a page region is not enclosed by any page regions and therefore is only enclosed by itself, its nesting depth is 1. Let S_i denote the set of page regions, all of which have the nesting depth i , where $1 \leq i \leq h$, h is the maximum nesting depth of all page regions in S . Obviously, for each i , $1 \leq i \leq h$, the page regions in S_i are mutually disjoint. Clearly, the sets S_i , $1 \leq i \leq h$, form a set partition Φ of S . This

set partition Φ may be organized as a totally ordered list of subsets of S where the order relation $X \leq_p Y$ means that for each $x \in X$ there exists a $y \in Y$ such that y encloses x . Thus $S_h \leq_p S_{h-1} \leq_p \dots \leq_p S_1$. In other words, for each page region x in S_i , $1 < i \leq h$, there exists a y in S_{i-1} and y encloses x .

The other order relation among page regions comes from the following fact. Since each subset S_i , $1 \leq i \leq h$, consists of mutually disjoint page regions, the page region identifiers in S_i must be different. This is proven by contradiction. If there are two page regions (r, l) and (r, m) which have the same identifiers and different levels, then one page must enclose the other one, which contradicts the condition that all page regions in S_i are mutually disjoint. The identifiers of page regions are integers. Thus the page regions in S_i can be ordered according to the identifiers of the page regions.

So far these two types of orders in S have been defined. Given two page regions (r, l) and (r', l') in S , either one is a proper subset of the other, in which case the two page regions belong to two different subset S_i and S_j of Φ comparable under $\leq_p$, or they are disjoint, in which case they are comparable under the identifiers of the page regions. This observation allows us to organize the directory in two levels, as an ordered list of subsets $S_1, \dots, S_h$ where each subset S_i , $1 \leq i \leq h$, is itself organized as an order list of page region identifiers. If the outer list is represented by a binary tree, then each node of this binary tree may point to the root of the representation of an inner list. This representation can be a binary tree or a B-tree if secondary storage is required because of the size of each list. If the outer list, too, is organized as a B-tree, each entry is the root of another B-tree representing an inner list. The new directory structure leads to a B-tree behavior in processing and performance.

2.6 Multilevel Grid File

Krishnamurthy et al. in [15] proposed the multilevel grid file. It uses the same partitioning methodology and splitting policy as in the interpolation based grid file. The scales are represented by binary bits, the grid directory only contains the page regions which correspond to data pages. The major deviation from the interpolation based grid file is the implementation of the grid directory.

The content of the directory of the multilevel grid file can be summarized by the following set notation for the directory.

$$D_1 = \{(x, y) \mid x \text{ represents a page region, and } y \text{ is a pointer to the data page}\}$$

Page region x is a d -dimensional rectangular box that can be represented by d indices in the binary representation. Figure 2.10 shows a data space partition and its set of the directory, where the physical pages are shown by dashed boundary rectangles. There are twelve page regions, accordingly there are twelve entries in directory D_1 .

As already indicated in the interpolation based grid file, two problems are involved to realize the directory.

1. Given a record $k = \{k_0, k_1, \dots, k_{d-1}\}$, how does one identify an entry in the directory which points to the data page where the record is stored?
2. Since the grid directory only contains the page regions, how does one efficiently find the page region which encloses the grid cell that the record lies?

The first problem can be solved in the following way. Given a record $k = \{k_0, k_1, \dots, k_{d-1}\}$, a set of indices $(i_0, i_1, \dots, i_{d-1})$, where i_j , $0 \leq j < d$, is a binary representation of each index, can be calculated by the same method as in the interpolation based grid file. Identifying an entry in the directory can be done by

Figure 2.10: Representation of Page Regions in Multilevel Grid File

Figure 2.11: Directory Organization in Multilevel Grid File

looking for the record (x, y) in the directory set such that x_j is a prefix of the binary representation of i_j , where $x = (x_0, x_1, \dots, x_{d-1})$ and $0 \leq j < d$.

In order to address the second problem, the authors observed that D_1 is itself a file. Therefore, it needs to be organized in some fashion. Thus, the problem of finding a record in D_1 is quite similar to that of looking for a record in the original data file (say D_0). Based on this observation, they proposed to organize set D_1 as a directory D_2 for file D_1 . This organization was called the multilevel grid file. Figure 2.11 depicts the multilevel grid file representation for the Figure 2.10. The

index page capacity is five. Even though there are only twelve entries in D_1 there are four physical pages associated with it. The directory of D_1 has four entries, all contained in one physical page of D_2 . An entry (10, 0) in D_2 represents that all records whose values of the first key start with 10, and whose values of the second key start with 0. Note that the page in D_1 , which entry (10, 0) in D_2 points to, has the page region (i.e., (10, 0)) subdivided into the following page regions: (100, 00), (100, 01), and (101, 0).

The multilevel grid file organization offers the following advantages over the original grid file:

1. As achieved in the interpolation based grid file, only those directory entries that represent nonempty regions are kept in the multilevel grid file. As a result, the number of directory entries is always less than or equal to the number of records regardless of the record distribution and correlations among keys. Therefore, the asymptotic growth of the directory is linearly dependent on the growth of data.
2. Splitting and merging a directory are easier in the multilevel grid file than in the original grid file because they occur only "locally". For example, when a data page overflows, only the directory entry pointing to this page is affected and is propagated recursively to the directory entry at the next higher level, if the directory page also overflows.
3. Since the number of directory entries is less than that of the original grid file, searching for directory entries for a range query is less costly. Another characteristics of the multilevel grid file which improves the performance of range queries is its hierarchical architecture of the directory, which reduce the search space in reading the data pages.

However, the conventional splitting policy used in this method still produces a large amount of empty pages when the data is not uniformly distributed. This kind of representation of page regions takes much space in each directory page. As a result

the capacity of a directory page is decreased. Furthermore, in order to identify an entry, each record in the directory page has to be searched. Splitting directory page is processed in the same manner as splitting data page, thus the utilization of directory page decreases. Some empty directory pages may even be created.

2.7 Splitting Policies of the Grid File

Chun et al. in *A Partitioning Method for Grid File Directories* [4] provided a new method to split the data space for constructing a grid directory. It degrades the fan out of the grid directory. As indicated in [4], the directory expansion appears to have a quadratic or cubic growth rate when the data distribution is non-uniform.

The proposed method is called the midpoint record partitioning method. It splits a grid cell at the midpoint of a data record list in a selected dimension rather than splitting at the midpoint of an interval for a grid file. By using the linear scales which have the interval values for representing ranges of the grid directory, it is possible to split the directory in an interval which is not necessarily the middle of a range. Therefore, when splitting a grid, the position to split is selected according to data distribution in the overflow page. How to select the splitting dimension is not mentioned in this paper. It is also not very clear from the paper whether a selected splitting position evenly distributes the data into two data pages.

Mahnic in *Using Grid File For a Relational Database Management System* [19] proposed another splitting policy that partitions only useful boundaries to assure a successful split operation, i.e., two nonempty data pages can be generated. During a search for splitting boundary, the binary radix interval with the smallest level is chosen first and if the boundary lying at the mid-point of this interval enables a successful split the searching procedure is terminated. Otherwise the binary radix interval with the next smallest level is chosen and the boundary at the midpoint of this interval is tested for a successful split. This process is performed repeatedly

until a boundary that splits the data page into two nonempty ones is found.

By using the above methods, it is guaranteed that any insertion which causes an overflow needs only one directory split if necessary. It may still produce two pages, one with almost full occupancy and one with almost empty occupancy. Chun et al. in [4] shows results of the new partition method compared to the original grid file by performing a simulation. The results suggest that the method decreases the number of partitions. The number of directory expansions appears to be a quadratic or cubic function of the data records for highly non-uniform data distribution.

2.8 Summary and Motivation

The above improvement methods of the original grid file can be divided into three categories:

1. **Modifying the structure of the grid directory:** The major problem that influences the performance of the grid file is its d -dimensional grid directory. Most of the research has focused on modifying the structure of the grid directory. The interpolation based grid file, its two variations – the BANG file and the nested interpolation based grid file, and the multilevel grid file resolve the problem of the d -dimensional directory by using a representation of the data space in which page regions are represented explicitly so that there is always only one directory entry per data page. As a result, the index storage utilization is increased and the *fan out* of the directory is reduced. However a new problem arises in response to the modification of the grid directory. The most striking advantage of the grid file – two-disk access principle disappears and range query processing becomes inefficient. In order to obtain the address of a data page, the grid directory has to be queried for a specific page region. How the directory is structured determine the performance of the grid file.
2. **Modifying the Splitting Policy:** In previous implementations, the splitting policy follows the principle of the midpoint of an interval and a cyclic order

of dimensions. Chun et al. [4] and Mahnic [19] show an approach to split a data space according to its distribution and the simulation results in [4] prove that the change in the splitting policy dramatically reduces the size of the directory.

3. **Modifying the One Level Grid File Structure:** Many problems brought about the grid directory are due to its large size and its representation. The large size requires that the directory be stored on disk, and hence, the access to each entry of the directory needs a disk access. During the process of a record insertion, deletion and a range query, one often needs to access a large amount of entries, especially in a large sized data base system. This causes performance degradation. If the grid directory is limited to the size which can fit in memory, the number of disk accesses is decreased.

By using a one level directory to represent the whole data space, splitting an overflow grid cell may simultaneously split many empty grid cells. This increases the *fan out* of the directory. The multilevel grid file was proposed to resolve these two problems.

The analysis of the previous work on the grid file motivates us to improve the grid file mainly in the following four directions:

1. **New Mapping Function:** Compared with these proposed solutions, the original one level grid file guarantees two-disk access to locate a record and is efficient in processing range queries. It also exhibits a good utilization of index page and data page in uniform data distribution. But the cost of inserting and deleting becomes very high when splitting or merging of the directory occurs. This is because the use of the conventional array mapping function for the directory requires reorganizing the directory at each expansion or contraction step. The directory is stored on disk, therefore the reorganization is expensive. In order to resolve the problem, we use a multidimensional dynamic hashing scheme to expand or shrink the directory each time a split or merge of the

grid directory occurs. This design keeps the advantage of the original grid file and improves the performance of insertions and deletions of the grid file.

2. **Splitting Policy:** It is not necessary to split a page region at the midpoint of intervals and in a cyclic order of dimensions to simplify the algorithm. A new splitting policy is provided to partition a page region by selecting the partitioning point according to the data distribution in the corresponding data page. It is guaranteed that one partition is sufficient for any kind of distribution.
3. **Multilevel Grid File:** Non-uniform data distribution often forms local data clustering. If a splitting hyperplane cuts the whole space, many empty entries are duplicated unnecessarily. Clustered data represent a subspace of the whole space. The partition in the subspace only induces a $(d - 1)$ -hyperplane in the subspace rather than the whole space. In order to satisfy the following two requirements:
 - (a) the partition adapts to the density of data in different parts of a data space,
 - (b) an attempt is made to reduce the disk accesses for inserting and deleting records,

the multilevel grid file is proposed by restricting the size of the grid directory. The multilevel grid file reduces the size of the grid directory and degrades the cost of insertion and deletion of the grid file.

4. **Merging Policy:** The merging policy of the multidimensional buddy system and neighbor system leads to deadlocked empty regions of the data space. This can decrease data page utilization drastically. In order to prevent such spatial deadlock, additional checks have to be performed. Checking for deadlocks needs access to the directory which is stored on disk. Therefore, the performance is degraded. The merging policy of reversing the splitting operation can be used to avoid the deadlock problem and achieve a simple algorithm.

Chapter 3

Grid File Designs: One Level and Multilevel

The various implementations of the grid file organize the scales and the grid directory differently. This chapter first defines the basic data structures, the splitting and merging policies used in the following two designs. It then presents two schemes that resolve many drawbacks of the grid file caused by its multidimensional directory.

3.1 Basic Data Structures, Splitting and Merging Policies

3.1.1 Scales

The grid file scales define a partition of the domain of each dimension. They enable access to the appropriate grid cells by aiding the computation of their addresses based on the values of the relevant dimensions. In the original grid file, d one-dimensional arrays are used to implement the scale structures, called the linear scales. Though the scales are kept in memory, inserting, deleting and querying operations become inefficient as the size of database increases and non-uniform data distribution occurs. The linear scales can not represent the history of the partitions which is important to some operations such as merging two data pages. Thus, d ($d \geq 1$) binary trees are used instead of d linear scales. These are called the

Figure 3.1: Binary Tree Structured Scales in the Grid File

Binary Tree Structured scales (BTS-scales). Each internal node of such a binary tree stores a partitioning point representing a $(d - 1)$ -dimensional hyperplane that cuts the space into two rectangular shaped regions. Each leaf node is associated with a d -dimensional slice $S(i_j, j)$ of the space which is bounded by two neighboring partitioning hyperplanes, $0 \leq i_j \leq m_j$, $0 \leq j < d$, where m_j is the number of slices in dimension j . Such a slice $S(i_j, j)$ is addressed by the index i_j which is stored in the corresponding leaf. When a record $k = (k_0, k_1, \dots, k_{d-1})$ is inserted, an index i_j for each dimension j , $0 \leq j < d$, is obtained by searching the corresponding BTS-scale for the given component key k_j . The desired data page where the record lies is computed by $G(i_0, i_1, \dots, i_{d-1})$, where G is the address function, i_j is the index of dimension j . For example, if G is a row-major mapping function, then, G is defined as

$$G(i_0, i_1, \dots, i_{d-1}) = \sum_{m=0}^{d-1} \left(\prod_{n=0}^{m-1} m_n \times i_m \right)$$

In a 2-dimensional case, the BTS-scales could be shown as in Figure 3.1.

When the directory is split, the splitting dimension j and location i_j are computed, i.e., a slice $S(i_j, j)$ is determined by the splitting policy which is discussed later. Then a new partitioning point is inserted in the leaf node i_j of the BTS-scale for dimension j . This is done by replacing the leaf node i_j by a new internal node representing the partitioning point and then attaching two leaf nodes to the internal node. Additional modifications may be needed according to the different mapping functions and different implementations of the grid file. For example, in the row-major storage mapping, the indices *below* i_j retain their indices, while the indices above the point i_j are shifted by $+1$.

In the reverse action of splitting, when the directory needs to be merged, an internal node which corresponds to the merged hyperplane is located according to the merging policy. This internal node is deleted and one of its leaf nodes is moved up.

The relationship between a newly-created internal node and its two leaf nodes can be viewed differently. The two leaf nodes of an internal node represent two sides of a hyperplane. The left leaf node is used to specify the left side of the hyperplane. It retains all properties of the original slice. The right leaf node of the newly-created internal node is created with some properties determined by different storage mapping functions. The advantages of using BTS-scales over the linear scales are as follows:

1. Generally speaking, searching for a key in the tree structure is more efficient than searching the linear structure, especially when the scales become very large. Inserting and deleting a partitioning point are also simpler in the BTS-scales.
2. When merging a data page, it is necessary to find the corresponding page region in the data space. In the linear scales, a set of lower bounds and upper bounds have to be stored in the data page to represent a d -dimensional rectangular page region. Since the grid file should handle various types of record

keys, there can not be a fixed data type for defining these lower bounds and upper bounds. It is more expensive to process the different types of boundaries. Another drawback of the linear scales is that it takes a considerable amount of space to store the boundaries in a data page when the record key is represented by many bytes.

In BTS-scales, we can associate each node with a level. The level starts at 0 for the root of the binary tree. A page region corresponding to a data page can be represented by a set of levels $(l_0, l_1, \dots, l_{d-1})$. With a given component key k_i and a level l_i ($0 \leq i < d$), unique node n_i can be located by searching for the corresponding BTS-scale. The leftmost leaf nodes and the rightmost leaf nodes under the subtrees rooted in this set of nodes n_i ($0 \leq i < d$) correspond to the lower bounds and upper bounds of the page region, respectively. Therefore, an integer array of levels is used to store boundaries of the page region in the data file. Furthermore, it is easy to identify an existing boundary in the page region. An already existing boundary is enclosed in a page region only when there is a non-leaf node under the set of nodes which correspond to the levels of the page region.

3. We can keep track of buddies by representing the splitting process in d BTS-scales, thereby maintaining the buddy relationship.

Definition: A buddy of a data page B in dimension j ($0 \leq j < d$) is a data page R such that the corresponding page regions of these two data pages, B and R , are obtained by splitting a page region on dimension j in the course of partitioning the data space.

According to the above discussion, each page region can be represented by d levels and these d levels correspond to d nodes in the BTS-scales. Data pages B and R have the buddy relation in dimension j if their correspond-

Figure 3.2: Buddy Identification in BTS-Scales

ing nodes in dimension j have the same parent in the BTS-scale of dimension j .

For example, in Figure 3.2, three data pages correspond to three page regions b , b_1 and b_2 respectively, which are grid cells in the directory. Grid cell b is represented by nodes (n_1, n_3) , b_1 is represented by nodes (n_0, n_3) , and b_2 is represented by nodes (n_1, n_2) . The data pages corresponding to page regions b_1 and b have the buddy relation in dimension k_0 since their corresponding nodes n_0 and n_1 in dimension k_0 have the same parent. Similarly, the data pages corresponding to page regions b and b_2 have the buddy relation in dimension k_1 .

For a given data page, d nodes in the BTS-scales are located according to the levels of the corresponding page region. The d possible buddies can be found by looking for the siblings which have the same parents as the d nodes in the d BTS-scales. If some sibling is an internal node, the buddy in the dimension does not exist. Therefore, by using the BTS-scales, it is easy to find the buddies of a data page.

3.1.2 Directory Organization

The grid directory is used to maintain a dynamic correspondence between page regions in the data space and physical data pages stored. It is a d -dimensional array, but must be mapped onto linear storage on disk. The most natural mapping from the d -dimensional array to one dimensional space is the row major order. There are two major problems associated with this structure.

1. When splitting or merging in any dimension occurs, the array must be reorganized. As the directory becomes very large, the disk access needed for the reorganization is expensive;
2. If data is not uniformly distributed, many entries of the directory remain empty or the same. ;

Unlike the previously proposed improvements, our design does the following:

1. In order to avoid the reorganization, an approach is required to have a function for mapping the d -tuple addresses onto a linear consecutive addresses space [26]. With this idea, the problem of reorganizing a directory is transformed into the problem of expanding or contracting the directory by a $(d - 1)$ -dimensional hyperplane when splitting or merging the directory, thus the number of disk accesses is decreased considerably. The problem reduces essentially to that of defining an allocation function for the grid file. This is the basis of the one level extendible directory grid file.
2. From another viewpoint, the cost of directory reorganization can be reduced by the use of a multilevel grid file. Its motivation comes from an obvious observation: if reorganization occurs in memory, it is assumed to be inexpensive. To achieve the goal, the directory is structured into pages. Therefore, a hierarchical directory structure is obtained. The hierarchical directory extends a hyperplane over a subspace, but not the whole data space. A subspace usually represents uniform data distribution. Therefore, the multilevel grid file can effectively decrease the number of entries which point to empty data pages.

3.1.3 Data Pages

The data structure used to organize records within a page is of minor importance to the grid file system. Often the simplest possible structure, sequential allocation of records within a page, is suitable. Here we focus on the data structure which is used to represent the page regions of the data space.

The insertion and deletion of a record eventually operates on a data page. In addition to storing a set of records, a data page needs to contain a small amount of status information of the page, such as occupancy number. Another important information maintained in a data page is the correspondence between the data page and the page region. In the section on the scales, we discussed how to represent page regions according to the levels in the BTS-scales. Therefore, a one-dimensional integer array of levels can be stored in a data page to specify the correspondence of the data page to the page region. This information is used to select a splitting dimension as well as the location when an overflow page occurs. This also selects a candidate dimension to merge when a page underflows.

In most practical situations, data is not uniformly distributed over the data space. Therefore, the splitting policy of the grid file is likely to produce empty regions. Under this condition, an assignment of the page size to each page will probably result in poor storage utilization. To improve the storage utilization, it is essential to separate the empty pages from the general pages. For empty pages, we still need to store some information on the page regions because inserting of a record or merging of a data page may occur on empty pages.

3.1.4 Splitting and Merging Policies

Our splitting policy guarantees that our scheme performs well for both non-uniform and uniform data distributions.

The splitting of a data space is triggered by the overflow of a data page. In some cases all the records lie in a grid cell when an overflow occurs. Most often the records are distributed over several grid cells, and can be handled by a mere page split without any change to already existing partition boundaries. If data is not distributed uniformly, the splitting policy of the midpoint of an interval at the dimension chosen in a cyclic order makes the grid directory split more often. This simple splitting policy causes some useless partitions since it does not consider data distribution in the page region. The consequence is that the directory is expanded unnecessarily. However, another well-known probabilistic effect [22], the birthday paradox, is likely to happen: even if the number of records (people) is much smaller than the number of page regions (days in a year) in the data space, the probability of two or more points colliding in some page region (having a common birthday) is high. For this reason, the expected number of the directory elements grows faster than linearly in the number of points in the data square. The point at which a superlinear growth rate begins to be noticeable. Thus, it requires that the splitting policy favor some attributes by splitting the corresponding dimensions more often than others. It is also possible to split the directory and the scales in an interval which is not necessarily the middle of a range.

We propose a new method which splits the directory at the midpoint of a record list in the selected dimension determined by the data distribution. The method requires exactly one split. For example, suppose a data page capacity of 2 is used for the 2-dimensional grid file. If a record represented as m is inserted in the structure as shown in Figure 3.3 (a), the split of the directory can be accomplished at the midpoint of these three input records in dimension y . However, if we use the original grid file splitting policy, then the splitting is accomplished at the middle of an interval. This is four times the number necessary as is shown in Figure 3.3 (b). By using the proposed method, the grid partition of the data space successfully adapts to the actual data distribution.

Figure 3.3: Comparison of Two Splitting Policies

The new method is implemented with a queue structure. For each dimension i , $0 \leq i < d - 1$, we have a corresponding queue q_i with ordered nodes from the lower value to the higher value. Each node of the queue q_i represents a distinct value of i th component of records in the overflowing data page. The number of nodes in q_i is the number of distinct values of i th component of records. After all records in the data page are processed, the dimension whose corresponding queue has the maximum number of nodes is selected as the splitting dimension. The location is then determined at the midpoint of the set of nodes in the queue which corresponds to the splitting dimension. If the number of distinct values among all dimensions is the same, the cyclic splitting policy is carried out.

When splitting a page region, an existing boundary is selected first. If there is more than one existing boundary in the page region, we select a boundary at an earlier created point. If there is no existing boundary, the above method is used to select the splitting dimension and location.

There are three different page merging policies – the neighbor system, the multi-dimensional buddy system and the inverse of the splitting operation. The advantages and disadvantages of these three policies have been discussed in section 2.1. To avoid the deadlock problem, we use the policy of reversing the splitting operation in our design. Our design is flexible to adapt other merge policies. The policy of reversing

the splitting operation gives priority to the buddy which has the property that at some earlier point in time the partition was created to resolve the overflow of a data page. In order to keep the earlier point, a variable *last_splitting_dimension* is stored in the data page for determining the merge dimension when merging a data page is to be done. According to the method discussed in section 3.1.1, the candidate data page to be merged can be located by using the merge dimension and the levels in the data page.

After merging a data page, the variable *last_splitting_dimension* is updated to the next merge dimension. In order to obtain the next merge dimension, it is required that the order of creating the partitions in a grid file be maintained. Since the splitting policy in our design does not follow a fixed scheme, the order of the created partitions can not be maintained automatically as in the cyclic splitting policy. Therefore, when a partition is created, its order is stored in the internal node which represents the partitioning point. In this way, each partition corresponds to an order. The variable *last_splitting_dimension* is set to the dimension with the maximum order among d internal nodes which correspond to the d levels of the data page to be merged.

Each time a data page is merged, merging the directory is checked to see whether all the elements lying in one side of the merge hyperplane are the same as the elements lying in the other side. Since the grid directory is stored on disk, comparing these elements needs a large amount of disk accesses. This problem is resolved by simply attaching a field *splitno* to each internal node. The *splitno* indicates how many splittings of data pages on the hyperplane in the internal node. When a hyperplane is created, the *splitno* is initially set to 1. It is increased by 1 whenever a split occurs on the existing hyperplane. In the reverse, it is decreased by 1 whenever merging of data pages occurs on the hyperplane. Merging the directory occurs only when the corresponding *splitno* becomes 0. It also triggers the modification of the corresponding BTS-scale.

3.2 Design of the One Level Extendible Directory Grid File

Instead of reorganizing entire directory whenever a directory expansion occurs, a simple mechanism to expand the directory by adding a $(d - 1)$ -dimensional hyperplane to the end of the directory is proposed. In the reverse, shrinking of the directory involves deleting the hyperplane that was last inserted in the corresponding dimension. The implementation of this scheme is presented in section 3.2.2. To achieve the dynamic behavior of the directory, a storage mapping function proposed by Otoo [26] is introduced first.

3.2.1 The Directory Mapping Function

The grid directory can be conceived as a d -dimensional address space which is partitioned linearly into $m_0 \times m_1 \times \dots \times m_{d-1}$ cells, where m_j ($0 \leq j < d$) is the number of indices. Each cell corresponds to an entry addressed by a d -tuple integer coordinate address $\langle i_0, i_1, \dots, i_{d-1} \rangle$. The content of a cell is an address of a data page which includes all points lying in the cell. If n denotes the number of cells addressed, then $n = \prod_{j=0}^{d-1} m_j$. In order to keep this consistent with the previous definition, the set of cells having a common index of any dimension is called a slice. A slice represents a side of a hyperplane. We desire to have a scheme which expands or shrinks the directory by adding or deleting a sequence of entries each time a split or a merge of the grid directory occurs.

Consider each key K as composed of d values, i.e., $K = (k_0, k_1, \dots, k_{d-1})$. By searching the corresponding BTS-scale for each component key k_j , ($0 \leq j < d$), the d coordinates of a data point are converted into internal indices $\langle i_0, i_1, \dots, i_{d-1} \rangle$. These are derived from the information stored in the leaf nodes of the BTS-scales as described in the previous section. With the d -tuple address $\langle i_0, i_1, \dots, i_{d-1} \rangle$ of a grid cell, a mapping function G that maps the integer coordinates $\langle i_0, i_1, \dots, i_{d-1} \rangle$

one to one onto the set of consecutive addresses $\{0, 1, 2, \dots, n-1\}$ of the linear address space is needed. An extension of the index range of dimension i from m_i to m_i+1 induces the extension of the address space by $\prod_{j=0, j \neq i}^{d-1} m_j$ cells.

Consider a d -dimensional address space $U^d = (0 : m_0, 0 : m_1, \dots, 0 : m_{d-1})$ and suppose a split occurs in slice $S(i_j, j)$, where i_j is the index in dimension j , $0 \leq i_j < m_j$, $0 \leq j < d$. This implies that a $(d-1)$ -dimensional hyperplane splits the original slice $S(i_j, j)$ into two. In order to represent the additional slice in the d -dimensional address space, the index range of dimension j is extended from m_j to $m_j + 1$ and the new slice is assigned the index $m_j + 1$. If we consider that the cells within this new slice are allocated in row major order, then we can address every element of the slice once we know the starting address of the first cell of this slice, i.e., the address of the $(d-1)$ tuple cell $\langle 0, \dots, m_j + 1, \dots, 0 \rangle$. We know the last address of a grid directory in a file. By indicating the address of the first cell in the slice as the last address of a grid directory, slice $S(i_{m_j+1}, j)$ introduced by the splitting dimension j and the index $m_j + 1$ is adjoined to the grid directory at each expansion step. Using the fact that every index is created at some expansion step during the process of partitioning data space, a starting address should be related to each index in order to address the elements in the directory. S_{i_j} is denoted the starting address of the index i_j , where $0 \leq i_j < m_j$, $0 \leq j < d$. The elements in slice $S(i_j, j)$ are stored in a row major order, thus their addresses are determined by the following function.

$$G(i_0, i_1, \dots, i_{j-1}, i_{j+1}, \dots, i_{d-1}) = \sum_{n=0, n \neq j}^{d-1} (c_n \times i_n)$$

where

$$c_n = \prod_{r=0, r \neq j}^{n-1} m_r$$

The value of m_r , $0 \leq r < d$, is determined when creating the slice. We call c_r the coefficient of dimension r in the slice. Like the starting address, the c_r is related to each index. The $(d-1)$ coefficients of a slice should be also stored together

with each index. With the above structure and definition, each coordinate in index $(i_0, i_1, \dots, i_{d-1})$ is mapped one to one on $\{0, 1, 2, \dots, n - 1\}$, where $n = \prod_{j=0}^{d-1} m_j$, by the mapping function R defined as

$$R(i_0, i_1, \dots, i_{d-1}) = S_{i_z} + \sum_{j=0, j \neq z}^{d-1} (c_j \times i_j)$$

where $z = \text{highest subscript such that } i_z = \max(i_j)$, S_{i_z} is a starting address of i_z and c_j is the coefficients of dimension j in slice $S(i_z, z)$, $j \neq z$.

In order to give an intuitive understanding of function R , we depict the schematic storage layout for a 2-dimensional case in Figure 3.4 (a) which is produced by the partitions in Figure 3.4 (b). The meaning of the leaf nodes will be explained in the next section. Note that this schematic storage layout describes the directory growth that results from the partitioning process. Therefore, different space partitions correspond to different schematic storage layouts. The most important property of the mapping function R is that it allows a slice to be adjoined to the directory without reorganizing the directory.

3.2.2 Expansion and Contraction of the Directory

The dynamic behavior of the grid file is best explained by an example. Consider a file storing 2-dimensional records with $m_0 = 2$, $m_1 = 2$, where m_0 and m_1 indicate the current number of indices of the 2-dimensional data space. The initial situation is depicted in Figure 3.5, where each leaf node is represented by a 3-tuple vector corresponding to $\langle \text{index}, \text{starting_address}, \text{coefficient} \rangle$. Note that the number of coefficients is $(d - 1)$ in a d -dimensional space. Assuming a split is required at point x , a slice is adjoined to the end of directory with the starting address being $m_0 \times m_1 = 4$, the index being $m_0 = 2$ and the coefficient being $c_1 = 2$. The result is shown in Figure 3.6. Now m_0 is added to 3. If another split occurs at y in Figure 3.6, the same procedure is repeated and Figure 3.4 is obtained. Therefore, at each expansion, a $(d - 1)$ -dimensional slice is adjoined to the directory. The values of these entries in the newly adjoined slice are copied from the split slice except that

Figure 3.4: Schematic Storage Layout

Figure 3.5: Initial Situation of the Grid File

Figure 3.6: An Example of Directory Expansion

one entry will point to a newly created data page.

To contract the directory, we reverse the directory expansion process. It is assumed that a directory merge occurs on the hyperplane h in an internal node N . If the slice that corresponds the right leaf node of the internal node N is last adjoined to the directory, merging the directory is simply done by contracting the directory with the slice. For example, in Figure 3.4, merging the directory occurs on the hyperplane h in the internal node N . The directory is contracted by changing the last address of the directory as 6. But in some cases, merging the directory is required at some earlier point, say on the hyperplane h in the internal node N in Figure 3.7(a). The slice that starts at the address 4 is merged with the slice that starts at the address 0. In order to avoid reorganization of the grid directory, another slice could be used to replace this slice to be merged (call this the merged-slice). For each dimension, we can always obtain the last adjoined slice from the number of indices $(m_0, m_1, \dots, m_{d-1})$. By copying values in the slice (call this the copied-slice) which corresponds the index m_j (j is the dimension on which a merge occurs) to the values in the merged-slice, the problem of merging the directory results in moving the elements after the copied-slice toward the copied-slice and then contracting the grid directory. In the above example, the elements in entries 6, 7, 11 are first copied into entries 4, 5, 10 respectively. Then the elements in entries 8, 9, 10 are moved to entries 6, 7, 8. The directory is contracted by changing the last address of the directory as 9. The result is shown in Figure 3.7(b).

Compared with the split, the merge needs more disk accesses. This is because the entries in the merged-slice and in the copied-slice are not totally consecutive. Reading and writing these elements require a number of disk accesses. In fact, this operation is a major factor in the performance of deletion. In order to contract the directory, all the elements after the copied-slice are moved forward in the linear array. Since these elements lie in a consecutive linear array, moving operation can be accomplished in several disk accesses depending on the amount of entries. However,

Figure 3.7: An Example of Directory Contraction

compared with the reorganization of the whole directory, this method is still less expensive in time.

3.2.3 Query Processing Strategy

The idea of processing queries in the original grid file can be used in our scheme with two modifications:

1. The first modification is due to the change on the structure of the scales. For an exact match search, searching in the d linear scales now corresponds to binary tree traversal to locate d leaf nodes which store d indices.

A range query of the grid file is defined by a set of intervals. Each interval is represented by a lower bound and upper bound. By searching the BTS-scales for lower bounds and upper bounds in the intervals, a set of leaf nodes is located. These leaf nodes have the property that in each dimension the leaf node corresponding to the upper bound of the query range must lie at the right of or in the same leaf node corresponding to the low bound of the query range. All leaf nodes of the binary tree can be linked from left to right. Therefore, a range query is converted to d linked leaf node chains. Each chain starts at the leaf node corresponding to the low bound of the range query and ends at the leaf node corresponding to the upper bound of the range query in a dimension. A group of entries which intersect the range query are traversed according to a row major order function and the d linked leaf node chains. Each page pointed to by the entries is read into memory and searched for the records which satisfy the range query.

2. The second modification is due to the different mapping functions. In the original grid file, the row major order mapping function maps a record to a grid cell according to the indices of the grid cell. The mapping function in our design requires that more information be stored. It works in the following way. The coordinates of a given record are converted to d indices stored in d leaf

nodes. For each node, there are three variables, *index*, *starting_address* and *coefficient* associated with it. The *starting_address* in the *d* leaf nodes is compared first. The one with the largest value is selected as the slice that includes the grid cell pointing to the expected data page. Then the address of the grid cell is computed using the function defined in section 3.2.1, the *index* and the *coefficient* in the leaf node which contains the largest *starting_address*.

3.2.4 Summary

Splitting and merging operations do not cause an entire directory reorganization in our design, thus the performance for insertions and deletions is improved. In most applications insertions and deletions of the directory occur rarely as compared to direct access and range query. In these kinds of applications, the directory structure described is sufficient. This directory organization can be called the one level extendible directory. It keeps the principle of two disk accesses for processing exact match search. Range queries can be processed efficiently as well. The splitting policy in this design reduces the expansion of the grid directory, thus the *fan out* of the directory is decreased. Moreover, memory utilization is optimal, a larger fraction of the directory may be kept in memory. These advantages of one level grid file outweigh the penalty incurred for uniform data distribution.

3.3 Design of the Multilevel Grid File

The burden of reorganizing the directory is largely alleviated in the one level extendible directory grid file. In the following two cases, the cost of disk access is still not tolerable.

1. One data page may be pointed by more than one entry in the grid directory. When an existing boundary is used in splitting a data page, some entries of the directory have to be modified to point to a newly created data page. The addresses of these entries are likely to be non-consecutive on disk. Modifying

these entries may require several disk accesses even if part of the grid directory can be read into memory in one disk access.

2. When a directory split occurs, one must expand a $(d - 1)$ -dimensional hyperplane to the end of the directory, copy n pointers and finally update one of them, where n is the number of entries in the hyperplane. The *copying* operation may require arbitrarily large number of disk accesses because these entries are likely to be stored in non-consecutive address space. Writing the $(d - 1)$ -dimensional hyperplane into disk can be accomplished by several disk accesses depending on the size of the hyperplane.

These difficulties are caused by the fact that the size of the directory is so large that it can not be kept entirely in memory. The proposed splitting policy involves less times to split the grid directory compared with the original one. However, the high *fan out* in the directory is still a drawback of the one level grid file. The reason for the high *fan out* is that the fine partitions, required to be localized in the data space where a cluster occurs, extend over the whole data space, i.e., the partition does not adapt to the density of the data in different parts of the data space; this leads to a directory that is not proportional to the size of the underlying amount of records. In order to overcome this drawback, it is proposed that the partitions should focus on the local data distribution. Therefore, a hierarchical directory structure is recommended. In this section, we present a scheme called the multilevel grid file that builds a hierarchical directory with controlled directory expansion. The directory is a multi-way tree with fixed size nodes that grow from top to bottom.

3.3.1 Multilevel Grid File Structure

The multilevel grid file consists of a collection of pages and variable *root.ID* that gives the address of the root page. There are two types of pages in a multilevel grid file.

1. **Directory pages** – A directory page contains a d -dimensional array and d binary trees. The d -dimensional array represents the grid directory. Its element

is a pointer to a page which is either a data page or a directory page. The row major mapping function is used to map an entry of a d -dimensional array to a linear array, which is stored at the start of the directory page. The d binary trees represent the BTS-scales which define the grid in the directory page. In order to systematically organize the directory page, the BTS-scales are stored at the end of the directory page.

2. **Data pages** -- Data pages are organized in a manner similar to the one level extendible directory grid file.

The following set of properties defines the multilevel grid file structure. The algorithms for insertions and deletions are designed so as to preserve these properties, and the algorithm for queries depends only on these properties.

1. Considering each page as a node and each element in a directory page as a page pointer ID , the resulting graph structure is a multi-way tree with the root defined by $root_ID$. Figure 3.8 illustrates the structure of a 2-dimensional multilevel grid file. All directory pages are internal nodes of the tree and all data pages are leaf nodes of the tree.
2. In each directory page, the page regions in it are disjoint, and their union is a d -dimensional rectangular-shaped region. This property is consistent with the requirement of partitioning the data space in the original grid file.
3. If the root page is a directory page (or if this is the only page in the tree, it will be a data page), the union of its page regions span the whole data space.
4. If a page pointer ID corresponds to a page region R , and the page Q , referred to by the page pointer ID , is a directory page, then the union of the page regions of the page Q defines the page region R .
5. If a page pointer ID corresponds to a page region R , and if the child is a data page, then all the records in the data page must be in page region R .

Figure 3.8: Structure of a 2-dimensional Multilevel Grid File

Generally the size of each directory page is defined as the size of a disk block which is a unit that an operating system reads or writes data from or to a file. Therefore, a directory page can be read into memory in one disk access. A directory page contains the grid directory and the scales in a subspace. When splitting of the grid directory is required, it can be easily done by reorganizing the grid directory in memory, as long as the resulting grid directory can fit into the directory page. When merging of the grid directory is required, it is also done by reorganizing the grid directory as long as the resulting grid directory contains more than one element. We discuss how to handle an overflow directory page in the next section.

3.3.2 Splitting and Merging Operations

When an insertion of a record causes the corresponding data page to overflow, the data in the data page will be distributed according to the splitting policy presented in section 3.1.3. When the corresponding region, which represents the data page with overflow, covers only one grid cell, splitting of the data page triggers a split of the corresponding grid cell in the data space, i.e., a new slice needs to be added to the directory. Since the size of the directory page is limited, adding the slice to the directory may cause the directory page to overflow. A method is proposed to solve the problem of an overflow directory page in the multilevel grid file.

When a directory page becomes full, a new directory page is created. This new directory page represents the cell that corresponds to the overflow data page in the original directory page. For example, Figure 3.8 shows directory page P has overflowed when a split occurs in cell R of P . A new directory page named P' is created as the child of directory page P . There is a page pointer in P pointing to the directory page P' . By creating a new directory page to resolve the overflow directory page problem, the multilevel grid file performs like a tree that grows from top to bottom.

The maximum number of children of a directory page is bounded by the number

of grid cells allowed in a directory page. Since some grid cells can share the same data page in the original grid file, it is possible that some page pointers in a directory page point to the same child which must be a data page. For the directory page which is not the root, there is only one page pointer from its parent pointing to it. That means a directory page is generated only when its corresponding page region in the parent page contains only one grid cell. This results in a straightforward merging algorithm. Merging of data pages is still done according to the policy of reversing the splitting operation. Merging of the grid directory is done by the reorganization of the grid directory. When only one data page is assigned to a directory page, the directory page can be replaced with the data page, thus deleting the directory page.

According to the method of processing an overflow directory page, inserting a record needs at most four disk accesses, two for distributing the data into two data pages and two for operating on two directory pages — the overflowed page and the newly created page. Therefore, the multilevel grid file clearly improves the performance of insertion and deletion in the original grid file. For non-uniform data distribution, the hierarchical directory structure of the multilevel grid file decreases the number of partitions by adapting data distribution in different parts of the space. The data area with high density needs more partitions which only influences the subspace corresponding to the directory page. Furthermore, the *fan out* of the directory is decreased since the data clustered in a local area often displays a uniform distribution.

3.3.3 Query Processing Strategy

The exact match search in the multilevel grid file is processed as an exact match search in the one level grid file preceded by a tree search. The exact match search in the one level grid file is executed exactly the same as the original grid file, except that all operations take place in memory. The result of the query is a page pointer which points to either a data page or a directory page. Like the general tree search algorithm, the search continues until a leaf node, i.e., one data page, is located.

Therefore, the performance depends on the depth of the tree.

The range query has the same basic idea as the exact match search. Instead of locating a single page as in the exact match search, a set of elements in a directory page is identified for further processing. For each element, the page referred to by it is read into memory. Depending on the type of the page, either the processing terminates or the processing continues recursively. Unlike the one level grid file, where the number of disk accesses for a range query comes from reading only data pages and the directory elements, the multilevel grid file incurs the disk accesses in reading directory pages of the tree structured directory.

3.3.4 Two Techniques for Improving Directory Page Utilization

An under utilized directory page arises whenever a new directory page is created. At the time of creating a new child of a directory page, there are only two grid cells in the directory page. If one entry of the grid directory is represented by a four-byte integer, then one node of binary trees needs 20 bytes for two pointers to children and a value of the partitioning point. Some additional information, such as the number of indices in each dimension, also needs to be stored in a directory page. Therefore, if a page which is generally 2048 bytes is assigned to a directory page at the creation time of the directory page, the utilization of directory pages drops considerably.

Two techniques are proposed to alleviate the problem, one is called elastic pages and the other is called deferred splitting directory pages.

1. **Elastic Pages** - A changeable size for directory pages is defined. When such a directory page is created, the minimum size is assigned to it. As records are inserted, the size is expanded until the maximum size of directory pages is met. At this point, the directory page has overflowed. A new child needs to be created. In order to systematically manage the variable size pages, each size

is defined as a factor of the maximum size, such as the minimum is 512 if the maximum is 2048. Therefore, whenever a page is created, a minimum size of 512 is allocated. As the directory is split, the successive page size is assigned to replace the previous one until the directory page with the maximum size has overflowed and then a new page is created. This is one method to solve the problem of low index page utilization.

2. **Deferred Splitting Directory Pages** - Whenever a new page P is created at level l in the multi-way tree, the new page comes about by distributing $c + 1$ records into two data pages where c is the capacity of data pages. To search for one record in these two data pages, $l + 2$ disk accesses are needed, l for searching the directory pages above level l , 1 for searching the newly created directory page and 1 for searching the data page. In order to solve the problem of low utilization of pages, an overflow page is created to resolve the problem. The retrieval performance remains the same, but the low page utilization problem is alleviated. Therefore, a newly created page contains at least four entries.

3.3.5 Summary

The advantages of the multilevel grid file over the one level grid file are:

1. **Easy Splitting and Merging** — Splitting and merging directory are local operations affecting only a few pages. The discussion on these issues has been given in section 3.3.2. The number of disk accesses needed to perform splitting and merging operations on the grid directory is minimized by storing the grid directory in different directory pages independent of each other.
2. **Fan Out of the Directory** — The multilevel grid file decreases the *fan out* of the directory in the following two ways: 1) By extending a hyperplane over a subspace but not a whole data space, the total number of entries in all directory pages is drastically reduced. Many empty entries that would

otherwise be stored in the one level grid file are deleted in the multilevel grid file. 2) For each directory page, usually representing a clustered data subspace, the low *fan out* can be achieved for such uniform data distribution. Therefore, the problem of the high *fan out* in the one level grid file is resolved.

3. Range Query -- The multilevel grid file is efficient in processing range queries. Range queries are potentially very costly for two reasons:

- a large portion of the directory, i.e., a hyperplane must be searched;
- a large number of false drops results in accessing the data pages.

The former problem is greatly alleviated in our scheme because the directory is stored in memory. Reading an element does not take any disk access. The hierarchical structure can represent a large amount of data, therefore the depth of the tree generally is acceptable. The grid file allows several grid cells to share a data page. In order to avoid searching for records in the same page, the page should be checked whether it has already been searched before actually reading the page into memory. In the one level grid file, this operation still needs disk accesses. In non-uniform data distribution, a lot of entries in the directory point to the same pages or empty pages. Therefore, much more disk accesses are required for the check procedure which does not produce the real result of a query. The multilevel grid file completely solves the problem by operating in memory.

Another important enhancement comes from the multilevel architecture of the directory. A directory entry representing a region points to a physical block that contains lower level directory entries subdividing the region. These lower level directory entries, which are close to one another in the domain space, tend to be in the same physical block. Therefore, searching a hyperplane must be less costly because more than one directory entry can be searched in one page access. This contrasts with the case of the one level grid file, in which one page access is needed per one directory entry.

Figure 3.9: Limitation in the Balanced Multilevel Grid File

Like all unbalanced tree structures, the multilevel grid file suffers from the drawback of the worst case $O(P)$ access performance when searching a record, where P is the total number of pages. Although the splitting policy decreases the level of trees properly, we still risk to find a record depending on the depth of the tree.

To solve this problem, the balanced multilevel grid file is proposed, where the tree is always totally balanced in the sense that the number of nodes accessed on a path from the root node to a leaf node is the same for all leaf nodes. In order to implement a balanced structure, the scales and the grid directory in each directory page should be split or merged like the original grid scales and grid directory respectively. The multi-way tree is expected to grow from bottom to top for supporting the balanced tree structure and improving the directory page utilization.

The major obstacle preventing the design of a balanced multilevel grid file is how to select the splitting dimension and splitting location. For example, a directory page could be shown in Figure 3.9. Each grid cell refers to a lower level directory page or a data page. The splitting policy which adapts to the actual distribution could form a non-systematic partitions so that only historically first splitting line is guaranteed to extend over the whole data space, e.g., the thicker line in Figure 3.9. If the splitting policy for the directory page requires that the partition must happen at the splitting line which extends over the whole data space, a directory page split would not cause

splits to cascade over lower levels, but it causes the directory page utilization to fall below a given parameter. In the above example, splitting along the thicker line implies that one of the new directory page will be nearly empty. In the other hand, moving the thicker split line to the left would result in a better directory page utilization, but it poses the problem of what to do about lower level nodes which lie on both sides of this split line. It means that the splitting will cascade over lower levels and may create lower level directory pages and data pages which are nearly empty. Therefore, when time comes to split a directory page into two, there is no convenient way to make the split. K-d-B-tree [33] met the same problem, but it gave no explicit internal representation and explanation. The question of efficient implementation of the balanced multilevel grid file is still left open.

Chapter 4

Implementation and Software Modules

4.1 The One Level Extendible Directory Grid File

4.1.1 Data Structure

The scales are organized as independent d binary tree scales in a d -dimensional space. There are two types of nodes: an internal node represents a hyperplanes in the data space and a leaf node represents a slice bounded by two neighboring hyperplanes. The data structure for internal nodes contains two pointers to children and other additional information. Three additional information are associated with each internal node. These are: *splitno*, *order* and *key*. The variable *splitno* defines the number of splitting times for data pages using the hyperplane in an internal node. It is initially set to 1. When the value is 0, it means that all the elements in two adjacent slices of the hyperplane in the grid directory are the same. Merging the two adjacent slices of the directory is then required. The variable *order* gives the order of creating hyperplanes. It is for implementing the merging policy efficiently. The variable *key* defines a splitting point which represents a $(d - 1)$ -dimensional hyperplane in the d -dimensional data space. Leaf nodes include all necessary information defined in section 3.1.1. Its structure is described as follows:

$\langle index, description[1..d] \rangle$

with the following interpretation:

index : the index of the corresponding slice.

description[1..*d*] : For a leaf node in dimension *j*, the *description*[*j*] gives the starting address of the slice. Other elements of the *description*[*i*] ($i \neq j, 0 \leq i < d$) define the coefficients in the slice.

The grid directory is stored on disk. Each element is an integer representing the page number in a file. If the element is smaller than 0, the corresponding data page is an empty page which is stored in a different place from non-empty data pages. We use a separate file to maintain all information on empty pages. We keep a directory page in memory, which includes a set of elements of consecutive entries in the grid directory. The address of the first entry and the size of the directory page are also kept in memory. Whenever an entry is required, it is first looked up in the directory page. If the required entry is found in the directory page, its element can be accessed without a disk read. Otherwise, a set of entries which start from the required entry is read into memory to replace the elements in the directory page.

Records are stored in data pages one after another in no specific order. Each data page has a *header* which includes the information used for managing the data pages and the correspondence between the page region and the data page. This information includes *freelink*, *pageno* and *number*. The variable *freelink* is a link pointer for maintaining all empty space in the data file. The variable *pageno* is the order of pages in the data file, starting from 0. The variable *number* indicates the current number of records in the data page. A page region is represented by *d* integers called *levels* in the BTS-scales. The *last_splitting_dimension* is also stored in the *header* for conveniently accomplishing the merging policy. In order to improve storage utilization, we separate the empty pages from the actual pages. Empty pages are created and deleted in the same way as actual pages. They only contain

the header information as described above.

In our implementation of the grid file, the scales are loaded into memory as d binary trees at the beginning of a session and kept in memory until the session ends. Some information on the grid file, such as the number of indices in each dimension, is also read from disk at the beginning of a session.

4.1.2 Insertion and Deletion Algorithms

The algorithms of insertion and deletion must follow an exact match search. Generally the exact match search returns several information for subsequent operations, such as the data page B where the record to be inserted or deleted lies and a set of indices $\langle i_0, i_1, \dots, i_{d-1} \rangle$ of the grid cell whose element points to data page B . In our implementation, we do not retain duplicate records. Before presenting the algorithms, we define some symbols that we use in the following algorithms.

$K = (k_0, k_1, \dots, k_{d-1})$: key of a record;

$I = (i_0, i_1, \dots, i_{d-1})$: index of a grid cell in the d -dimensional grid directory;

$root = (root_0, root_1, \dots, root_{d-1})$: roots of d binary trees;

$M = (m_0, m_1, \dots, m_{d-1})$: number of indices in each dimension.

The components of the above four variables will be represented in the form of array elements in our algorithms, for example, k_j is represented as $k[j]$, $0 \leq j < d - 1$.

$B, B1, B_{empty}$: data pages which can have some attributes defined in the proceeding section. Attributes are represented in the form of $B \rightarrow attribute$, e.g., $B \rightarrow number$;

$Address_B$: address of data page B ;

$Region_B$: the page region corresponding to data page B ;

$Entry$: an entry in the grid directory;

- G* : the mapping function;
- Node* : a node of binary trees;
- File* : a grid file.

We assume the existence of the following functions or procedures:

- ContractDirectory(M, Root, Merge_dimension):** Contracts a directory according to the method in section 3.2.2 and modifies the scales;
- CreatePage()** : Creates a data page in the data file and returns the data page;
- CreateEmptyPage()** : Creates an empty page in the data file;
- DeleteEmptyPage(B_empty)** : Deletes empty page *B_empty* from the data file;
- DeleteRecord(K, B)** : Deletes record *K* from data page *B*;
- ExtendDirectory(M, Root, B1, Splitting_dimension, Splitting_location):** Extends the grid directory by the slice defined in the arguments according to the method in section 3.2.2;
- FindInPage(B, K)** : Looks for record *K* in data page *B*;
- GetPageRegion(B)** : Returns the page region of data page *B*;
- GetBuddy(B, root[j])** : Searches for a buddy of data page *B* in the tree *root[j]* and returns the buddy;
- GetLevel(B, root[j])** : Returns a *Node* in binary tree *root[j]* which corresponds to the level of data page *B* in dimension *j*;
- GetPartition(Region_B, Splitting_dimension, Splitting_location):** Returns *Splitting_dimension* and *Splitting_location* according to the splitting policy in section 3.1.4;

- InsertRecord(K, B)** : Inserts record K into data page B ;
- MaximumOrder($Root, B$)** : Searches d binary trees for the node with the maximum order;
- MergePage($B, B1$)** : Combines the records in two data pages B and $B1$ into data page B ;
- ModifyRegion($Region_B, Value$)** : Modifies the elements lying in region $Region_B$ to point to $Value$;
- ReadPage($Address_B$)** : Takes the input as an address of a data page and returns the page in the data file;
- ReadElement($Entry$)** : Reads the element which is stored in $Entry$ of the grid directory, i.e., returns the corresponding address of a data page;
- SearchTree($root[j], k[j]$)** : Converts component $k[j]$ into index $i[j]$ by searching the BTS-scales defined by $root[j]$;
- SplitPage($B1, B2, Splitting_dimension, Splitting_location$)** : Distributes the records in data page $B1$ into two data pages $B1$ and $B2$ according to the splitting dimension and splitting location.

The following is an algorithm for processing exact match search in the one level extendible directory grid file. The algorithm takes as input the specified grid file *File* and a record K . If the grid file contains only one data page B , we search the page for record K . Otherwise, the d indices are obtained by searching d binary trees for d components of record K . The mapping function maps the d indices to the address of a grid cell in the grid directory. The element in the grid cell is first read to memory, then its corresponding data page is read to memory. We search the data page for the Record K . The data page and the d indices are returned.


```

ALGORITHM Exact_Match( File, K, B, I )
BEGIN
  IF ( File == Empty )
    THEN return( FALSE )
  ELSE IF ( File contains only one page )
    THEN BEGIN
      B = 0;
      return( FindInPage( B, K ) );
    END
  ELSE BEGIN
    FOR ( each dimension j )
      I[j] = SearchTree( root[j], K[j] );
    Entry = G( I );
    Address_B = ReadElement( Entry );
    B = ReadPage( Address_B );
    return( FindInPage( B, K ) );
  END
END

```

In the above algorithm, only *ReadElement* and *ReadPage* may access disk, therefore the principle of two-disk access is preserved.

The following is the insertion algorithm of the one level extendible directory grid file. The algorithm takes as input the specified grid file *File* and a record *K*. If the grid file is empty, a data page is created and record *K* is simply inserted into the first data page of the grid file. Otherwise, the exact match search algorithm is invoked. This returns the corresponding data page *B* and index *I* in the grid directory. If data page *B* is empty, then an actual data page is created, the record is inserted into it and the empty data page is deleted from the data file. The information on the page region is copied from the empty page into the created actual data page. The elements in the page region are changed to point to the address of the actual data page. If data page *B* is not empty and not full, the record is inserted into data page *B*. Otherwise, an additional data page is created, the splitting dimension and location are determined by the splitting policy. The records in data page *B* and record being inserted are distributed into two data pages. If the split of data page *B* triggers the splitting of the grid directory, the grid directory is expanded and the program returns. Otherwise, an existing boundary is used. If using the boundary to

split data page B creates an empty data page, the splitting procedure will continue until a successful split occurs.

```

ALGORITHM Insert( File, K )
BEGIN
  IF ( File == Empty )
  THEN BEGIN
    B = CreatePage();
    InsertRecord( K, B );
    return( TRUE );
  END;

  IF ( Exact_Match(File, K, B, I) == TRUE )
  THEN BEGIN
    print "the record is in the Grid File";
    return( FALSE );
  END;

  IF ( B is an empty page )
  THEN BEGIN
    B1 = CreatePage();
    B1 = B;          /* Copy the header information from B to B1 */
    InsertRecord( K, B1 );
    DeleteEmptyPage( B );
    Region_B1 = GetPageRegion( B1 );
    ModifyRegion( Region_B1, B1->pageno );
    return( TRUE );
  END;

  IF ( B->number < PAGE_CAPACITY )
  THEN BEGIN
    InsertRecord( K, B );
    return( TRUE );
  END;

  /* process an overflow of a data page */
  B1 = CreatePage();
  need_split = TRUE;
  WHILE ( need_split )
  BEGIN
    Region_B = GetPageRegion( B );
    GetPartition( Region_B, B, Splitting_dimension, Splitting_location );
    IF ( the partition is an existing boundary )
    THEN BEGIN
      SplitPage( B, B1, Splitting_dimension, Splitting_location );
      IF ( B->number == 0 OR B1->number == 0 )

```

```

        THEN BEGIN
            /* the split creates an empty data page */
            B_empty = CreateEmptyPage();
            Region_empty = GetPageRegion( B_empty );
            ModifyRegion( Region_empty, B_empty->page_no );
            need_split = TRUE;
        END
    ELSE BEGIN
        /* the split success */
        Region_B1 = GetPageRegion( B1 );
        ModifyRegion( Region_B1, B1->pageno );
        need_split = FALSE;
    END
END
ELSE BEGIN
    SplitPage( B, B1, Splitting_dimension, Splitting_location );
    ExtendDirectory( M, Root, B1, Splitting_dimension, Splitting_location );
    need_split = FALSE;
END
END
END
END

```

The following is the deletion algorithm of the one level extendible directory grid file. The algorithm takes as input the specified grid file *File* and a record *K*. If the grid file is empty or the record is not in the grid file, the value *FALSE* is returned. Otherwise record *K* is deleted from the data page where the record lies. The algorithm checks to see if page merges can take place each time a deletion is done. The buddy of the data page to be merged is obtained by searching the binary tree which corresponds to the variable *last_splitting_dimension*. If the sum of the number of the records in two pages (buddy and the data page from which a deletion occurred) is less than a predefined threshold, merging of data pages occurs. After merging data pages, a check is made to see if the directory can also be merged. If the variable *split_no* is 0, the grid directory is contracted and the grid scales are modified. The algorithm ends when no more data pages need being merged.

```

ALGORITHM Delete( File, K )
BEGIN
  IF ( File == Empty )
  THEN
    return( FALSE );

  IF ( Exact_Match( File, K, B, I ) == FALSE )
  THEN BEGIN
    print "the record is not in the Grid File",
    return( FALSE );
  END;

  DeleteRecord( K, B );
  need_merge = TRUE;
  WHILE ( need_merge )
  BEGIN
    Merge_dimension = B->Last_splitting_dimension;
    B1 = GetBuddy( B, root[Merge_dimension] );
    IF ( ( B->number + B1->number ) > PAGE_THRESHOLD )
    THEN
      need_merge = FALSE
    ELSE BEGIN
      MergePage( B, B1 );
      Region_B1 = JetPageRegion( B1 );
      ModifyRegion( Region_B1, B->pageno );
      B->last_splitting_dimension = MaximumOrder( Root, B );
      Node = GetLevel( B, root[Merge_dimension] );
      Node->splitno = Node->splitno - 1;
      IF ( Node->page_no == 0 )
      THEN
        ContractDirectory( M, Root, Merge_dimension );
      END
    END
  END
END

```

4.1.3 Query Algorithm

We consider the algorithm for processing partial match and range searches in this section. The algorithm to process an exact match search is equivalent to one specified in the previous section. A query is defined by a set of intervals

$$Q = ((l_0, u_0), (l_1, u_1), \dots, (l_{d-1}, u_{d-1})).$$

By searching the BTS-scales one after the other, Q is converted into a multidimensional rectangular region R in the directory. In our implementation, R is represented by d pairs of leaf nodes in the BTS-scales.

$$R = ((n_0^l, n_0^u), (n_1^l, n_1^u), \dots, (n_{d-1}^l, n_{d-1}^u))$$

Each pair of leaf nodes specifies the low and upper values of the region R in each dimension. The entries lying in this region R which are computed according to the mapping function G and coordinate indices $\langle i_0, i_1, \dots, i_{d-1} \rangle$ stored in the nodes $\langle n_0, n_1, \dots, n_{d-1} \rangle$, where $n_j^l \leq n_j \leq n_j^u$, $0 \leq j < d$, are processed by traversing the region R in column major order starting at the entry corresponding to d -tuple indices stored in the nodes $\langle n_0^l, n_1^l, \dots, n_{d-1}^l \rangle$. The elements of these entries are first read from disk. Data pages which are pointed to by the elements of the entries are then read from disk and searched for records that satisfy the range query. In order to guarantee that each of these data pages is loaded from disk exactly once, we check if the data page has already been processed before reading a data page from disk.

In our implementation, a query is initialized by the procedure and the records found in a query are passed to the user by subsequent calls of transfer procedures. Subsequent calls include *GetFirst*, *GetLast*, *GetNext* and *GetPrevious*.

Some new symbols and functions are:

- Q : as defined above, a set of query intervals;
- $R[j][m]$: the multidimensional rectangular region corresponding to Q . It is a 2-dimensional array, where $0 \leq j < d - 1$ and $m = l$ or $m = u$;
- $\text{CheckProcess}(\text{Address_B})$: Checks whether the page referred by *Address_B* has been processed;

GetNextEntry(I) : traverses all elements in the multidimensional rectangular region defined by $R[j][m]$ in the row major order;

SearchBound(root[j], Q, R[j][1], R[j][u]): Searches for a pair of leaf nodes in the BTS-scales defined by $root[j]$ according to Q ;

SearchInPage(B, Q) : Searches the records that satisfy Q in data page B .

```

ALGORITHM Range_Search( File, Q )
BEGIN
  FOR ( each dimension j )
    SearchBound( root[j], Q, R[j][1], R[j][u] );

  FOR ( each dimension j )
    i[j] = R[j][1];

  REPEAT
    Entry = G( I );
    Address_B = ReadElement( Entry );
    if ( CheckProcess( Address_B ) == FALSE )
      THEN BEGIN
        /* The page pointed to by Address_B has not been processed. */
        B = ReadPage( Address_B );
        SearchInPage( B, Q );
      END;
    GetNextEntry( I );
  UNTIL ( All entries have been traversed );
END

```

4.2 The Multilevel Grid File

4.2.1 Data Structure

There are two types of pages in the multilevel grid file. The data structure of data pages is exactly the same as one described in the one level extendible directory grid file, except that there is an additional *type* information indicating whether a page is a directory page or a data page. There are three parts of information in a directory page. The first is the header information of the directory page. It includes the number of indices and the root pointer to a BTS scale in each dimension. The

second part is the grid directory. Each element is a pointer to a page. The third part is the BTS-scales with their roots defined in the header information of the directory page. Each node of the BTS-scales consists of the variables *lchild*, *rchild* and *key*. The variable *key* defines a splitting point which represents a $(d - 1)$ -dimensional hyperplane in the d -dimensional data space. The values of the variables *lchild* and *rchild* in the internal nodes of the BTS-scales are greater than 0. These represent pointers to children of the internal nodes. In the leaf nodes, these values are equal to or less than 0, and represent indices of the grid cells.

4.2.2 Insertion and Deletion Algorithms

As in the one level extendible directory grid file, an insertion and deletion operation first identifies the affected data page by an exact match search. The address of the root page is indicated by the variable *Root_ID*. Each page has a *page_ID* stored in its parent node as an element in the grid directory. The multilevel grid file uses *ReadPage* for reading a data page or a directory page. The algorithm for processing an exact match search in the one level directory is quoted in this algorithm. We denote a directory page by *D*.

The following is an algorithm for processing an exact match search in the multilevel grid file. It takes as input the specified grid file *File* and a record *K*. If the grid file only contains one data page, we search the data page for the record. Otherwise, we traverse the tree. Starting at the root page of the grid directory, each time a page is read into memory, we determine whether the page is a directory page or data page. If the page is a data page, we search the page for the record *K*. Otherwise the page is a directory page. The exact match algorithm of the one level grid file is invoked and returns the address of the page where the record may lie. The algorithm is recursively repeated down the tree until a data base is located.

```

ALGORITHM Exact_Match( File, Root_ID, K, B, D, I )
BEGIN
  IF ( File == Empty )
  THEN
    return( FALSE )
  ELSE BEGIN
    IF ( File contains only one page )
    THEN BEGIN
      B = 0;
      return( FindInPage( B, k ) );
    END;
  END;

  D = Root_ID;
  WHILE ( TRUE )
  BEGIN
    One_Level_Exact_Match_Search( D, K, Page_ID, I );
    B = ReadPage( Page_ID );
    IF ( B->type == Data_Page )
    THEN
      return( FindInPage( B, k ) )
    ELSE
      D = B;
    END
  END
END

```

The above exact match algorithm provides three kinds of information. These are: the data page B where the record lies, the directory page D which is the parent of data page B and a set of indices I corresponding to the address of the data page B in the directory page D . Insertion and deletion modify the directory page D when a split or a merge occurs. Therefore all the operations on the grid directory defined in the previous section apply to the directory page in memory, instead of the whole directory on disk. We continue to use the previous function with one extra parameter D (directory page) which explicitly refers to the grid directory and the scales of the page modified. Four new functions are now introduced:

CheckDirectoryFull(D , Splitting_dimension, Splitting_location): Checks whether an overflow will happen in directory page D because of the required split;

CheckMergeDirectory(D, merge_dimension): Checks whether a merge of the directory occurs;

GetParent(Root_ID, D) : Finds the parent of directory page *D* in the multi-way tree rooted by *Root_ID* of the grid file;

Reorganize(D) : Reorganizes directory page *D* because of splitting or merging the grid directory.

The following is the insertion algorithm of the multilevel grid file. It takes as input the specified grid file *File* and a record *K*. If the grid file is empty, the first data page is created and record *K* is inserted into it. Otherwise, an exact match algorithm is invoked. This returns the data page *B* where record *K* should lie and the directory page *D* that is the parent of the data page *B* in the multi-way tree of the grid file. If the data page *B* is empty, then an actual data page is created and record *K* is inserted into it. The elements in the page region represented by the data page *B* are modified to point to the newly created data page. The empty page is deleted from the data file. If the data page *B* is not empty or full, record *B* is simply inserted into it and the algorithm returns. In the case that the data page *B* is full, a new data page is created and the page *B* is split according to the splitting policy. If the split requires adding a new boundary, directory page *D* is organized in memory. When directory page *D* is full, a new directory page is created. If the split uses the existing boundary, the elements lying in one side of the boundary are changed to point to the newly created data page. Using the existing boundary to split a data page may create an empty data page, which requires the splitting operation to continue until a successful split occurs.

```

ALGORITHM Insert( File, K )
BEGIN
  IF ( File == Empty )
  THEN BEGIN
    B = CreatePage();
    InsertRecord( K, B );
    return ( TRUE );
  END.

```

```

IF ( Exact_Match( File, Root_ID, K, B, D, I ) == TRUE )
THEN BEGIN
    print "the record is in the Grid File".
    return( FALSE );
END;

IF ( B is an empty page )
THEN BEGIN
    B1 = CreatePage();
    B1 = B; /* Copy the header information from B to B1 */
    InsertRecord( K, B1 );
    DeleteEmptyPage( B );
    Region_B1 = GetPageRegion( D, B1 );
    ModifyRegion( D, Region_B1, B1->pageno );
    return( TRUE );
END;

IF ( B->number <= PAGE_CAPACITY )
THEN BEGIN
    InsertRecord( K, B );
    return( TRUE );
END;

/* process an overflow of a data page */
B1 = CreatePage();
need_split = TRUE;
WHILE ( need_split )
BEGIN
    Region_B = GetPageRegion( D, B );
    GetPartition( D, Region_B, Splitting_dimension, Splitting_location );
    IF ( the partition is an existing boundary )
    THEN BEGIN
        SplitPage( B, B1, Splitting_dimension, Splitting_location );
        IF ( B->number == 0 OR B1->number == 0 )
        THEN BEGIN
            B_empty = CreateEmptyPage();
            Region_empty = GetPageRegion( D, B_empty );
            ModifyRegion( D, Region_empty, B_empty->page_no );
            need_split = TRUE;
        END
    ELSE BEGIN
        Region_B = GetPageRegion( D, B1 );
        ModifyRegion( D, Region_B, B1->pageno );
        need_split = FALSE;
    END
END
END

```

```

ELSE BEGIN
    SplitPage( B, B1, Splitting_dimension, Splitting_location );
    IF ( CheckDirectoryFull( D, Splitting_dimension, Splitting_location ) == TRUE )
    THEN BEGIN
        D1 = CreatePage();
        Entry = G( I );
        D->Entry = ( D1->pageno );
        Reorganize( D1 );
    END
    ELSE
        Reorganize( D );
    need_split = FALSE;
END
END
END

```

The following is the deletion algorithm of the multilevel grid file. It takes as input the specified grid file *File* and a record *K*. If the grid file is empty or record *K* is not in the grid file, the algorithm returns the value *FALSE*. Otherwise the record is deleted from the data page where the record lies. The algorithm checks to see if merging data pages can take place each time a deletion is done. The buddy of the data page to be merged is obtained from searching the BTS-scale which corresponds to variable *last_splitting_dimension*. If the sum of the number of the records in two pages (the buddy and the data page from which a deletion occurs) is less than a pre-defined threshold, merging of data pages occurs. After merging a data page, the directory merge is checked in memory. If the directory merging occurs, the directory is merged by reorganization. When the directory page only contains one data page, it is combined into its parent.

```

ALGORITHM Delete( File, K )
BEGIN
    IF ( File == Empty )
    THEN return( FALSE );

    IF ( Exact_Match( File, Root_ID, K, B, D, I ) == FALSE )
    THEN BEGIN
        print "the record is not in the Grid File";
        return( FALSE );
    END;

```

```

DeleteRecord( K, B );
need_merge = TRUE;
WHILE ( need_merge )
BEGIN
    merge_dimension = ( B->Last_splitting_dimension );
    B1 = GetBuddy( D, root[merge_dimension] );
    IF ( ( B->number + B1->number ) > PAGE_THRESHOLD )
    THEN need_merge = FALSE
    ELSE BEGIN
        MergePage( B, B1 );
        Region_B1 = GetPageRegion( D, B1 );
        ModifyRegion( D, Region_B1, B->pageno );
        B->last_splitting_dimension = MaximumOrder( D, B );
        IF ( CheckIfMergeDirectory( D, merge_dimension ) == TRUE )
        THEN BEGIN
            Reorganize( D );
            IF ( D->Root == NULL )
            THEN BEGIN
                D1 = GetParent( Root_ID, D );
                CombineDirectoryPage( D1, D );
            END
        END
    END
END
END
END

```

4.2.3 Query Algorithm

This section considers the range query algorithm in the multilevel grid file. As defined in section 4.1.3, a query can be expressed by the Cartesian product Q of intervals $[l_j, u_j]$ (corresponding to dimension j of a d -dimensional grid file, $0 \leq j < d$). We assume that the range query algorithm of the one level grid file is known. It returns a multidimensional rectangular region in the directory page. This region is represented by the two dimensional array $R[i][j]$ as defined in section 4.1.3.

The following is a range query algorithm of the multilevel grid file. The algorithm takes the input as specified grid file *File*, the address of the root page *Root_ID* and a query Q . It recursively traverses the directory pages in the depth-first search order to retrieve the records in the data pages whose corresponding page regions are

intersected with the query region defined by Q . Starting at the root page of the tree, each time a page is read into memory, we determine whether the page is a directory page or data page. If it is a directory page, the query Q is first converted into a rectangular region R in the grid directory of this page by searching its BTS-scales. It is done by invoking the range query algorithm of the one level grid file. Then the pages which are pointed to by the elements lying in this region R are recursively processed by the traversing R . If a data page has been read from disk, the records stored in it are processed. If the corresponding page region is completely contained in Q , then all records in the data page are qualified. Otherwise, the corresponding page region is not completely contained in Q . Each record stored in the page must be checked to see whether it is in Q .

During the query all directory pages and all data pages whose regions intersect Q are read from disk exactly once. Before loading a page from disk, it is checked whether this page has already been processed. Since the elements in the query region R of a directory page are traversed in the row major order, this check can be done by detecting the neighbor grid cells that have been processed to see if the same page pointer has been met.

```

ALGORITHM Range_Search( File, Root_ID, Q )
BEGIN
  IF ( Root_ID == NULL )
    THEN return;

  Page_ID = Root_ID;
  B = ReadPage( Page_ID );
  IF ( B->type == Data_Page )
    THEN BEGIN
      SearchInPage( B, Q );
      return;
    END
  ELSE
    R = One_level_Range_Query( B, Q, R[1][1], R[1][u] );

  FOR ( each dimension j )
    I[j] = R[j][1];

```

```

REPEAT
    Entry = G( I );
    IF ( CheckIfProcess( B, Entry ) != TRUE )
    THEN BEGIN
        Page_ID = GetElement( B, Entry );
        Range_Search( File, Page_ID, Q );
    END;
    GetNextEntry( I );
UNTIL ( All entries have been traversed )
END

```

4.3 Cache and Opened Grid File List

In order to improve the efficiency of operations on the grid file, two data structures are used in the implementation of the grid file.

1. **Cache** — An attempt to make optimal use of available memory leads to the scheme of managing the grid directory and data pages on disk with a cache in memory. It is worthwhile to use more auxiliary information in memory so as to save disk accesses. In our implementation, we create a cache in memory for buffering both the directory and data pages in the multilevel grid file and data pages in the one level extendible directory grid file.

The cache consists of a set of slots. The number of slots is called the size of a cache. Each slot contains the information for managing the cache and a buffer in memory. This information includes *occupiedbit*, *pinbit*, *changebit*, *reference*, *fileid*, *pageno* and *buffer*. They have the following meanings:

occupiedbit : It identifies if the slot is available;

pinbit : It identifies if the data in the buffer of the slot can be taken out from memory at the time of searching for an available slot;

changebit : It identifies if the data in the buffer of the slot has been modified;

Figure 4.1: A Cache Structure

reference : It identifies if the data in the buffer of the slot is referred recently;

fileid : It indicates the file that the data in the buffer of the slot comes from;

pageno : It indicates the location in the file where the data in the buffer of the slot is stored;

buffer : It is a pointer to a buffer where data resides.

The size of the buffer and the size of the cache can be defined at the time of creating the cache. The structure of a cache is shown in Figure 4.1. The operations on the cache are:

SetupCache(cache_size, buffer_size, cache) : It creates a cache with specified cache size and buffer size. All memory required by the cache are allocated;

ClearCache(cache) : It writes the modified data in the buffers of the cache into files and frees all memory that the cache

takes;

FindAvailableSlot(cache, slot) : It looks for an available slot in the cache.

If there is no empty slot, we apply the *leastrecentlyused* policy to find a slot and write the data in the buffer of the slot to the file if the data has been modified;

CheckPage(cache, pageno, fileid) : It checks if the page identified by the *pageno* and *fileid* is in the cache;

PushPin(cache, slot, status) : It changes the value of *pinbit* in the *slot* as *status*;

ModifyChangebit(cache, slot, status) : It changes the value of *modifybit* in the *slot* as *status*;

PageIn(cache, slot, pageno, fileid) : It reads data from the file specified by *fileid* and *pageno* to the buffer in the *slot*;

PageOut(cache, slot) : It writes the data in the buffer of the *slot* into the file.

In addition to improving the efficiency of the file operations, the cache structure also facilitates the portability of the grid file system since the module which implements the cache separates the disk-dependent parts from the grid file system itself.

- 2. Opened Grid File List** — We use this data structure to improve the efficiency of the system as well. This is due to the fact that some grid file information, such as the BTS-scales of the one level extendible directory, is kept in memory during a session. With this structure, a user is expected to open a grid file before he operates on it. From the point of view of the grid file system, the *open* operation reads the necessary information of the grid file into memory and opens the corresponding UNIX files. From the point of the view of users, the *open* operation returns a handler in memory to the grid file. This handler is an integer number. The subsequential operations on the grid

file use this handler. The opened grid file list also allows one to operate on several grid files at the same time. The operations on the opened grid file list are *OpenGridFile*, *CloseGridFile*, *AttachCache* and *InquireGFDesc*.

4.4 Software Modules and Dependencies

We have implemented our design concepts in the C language. It consists of about 9000 lines of code and runs on the SUN4/SPARC workstation. The grid file system is installed under the following four directories:

1. **directory source:** This includes all source code about the grid file.
2. **directory lib:** This contains a grid file library named *libgridfile.a* which includes all object code of the grid file.
3. **directory include:** This includes two grid file header files, one for the definition of data structures, the other for error messages.
4. **directory database:** This contains all UNIX files for maintaining the grid file, such as files storing data pages, the grid directory, etc.

In order to facilitate the portability of the grid file system and the ease of future development and maintenance, the grid file system is modulized into the following modules.

- GFhost** : The module contains the machine-dependent part of the grid file system. In fact, it includes all the functions about **Cache**.
- GFUtil** : The module includes a set of common functions between these two implementations of the grid file.
- GFManager** : The module contains the procedures for creating, deleting, opening, closing a grid file and attaching a cache to a grid file.
- GFInsert** : The module is related to the insertion procedure of the one level extendible directory grid file.

Figure 4.2: Module Dependencies in the Grid File Implementation

GFdelete : The module contains the deletion procedure of the one level extendible directory grid file and its relative functions.

GFfind : The module contains a set of query procedures for the one level extendible directory grid file.

MGFinser : It is like file *GFinsert* except that its all functions are regarded to the implementation of the multilevel grid file.

MGDelete : It is similar to file *GFdelete* except using the multilevel grid file technique.

MGFind : This includes a set of query procedures of the multilevel grid file.

Figure 4.2 illustrates the structure of these modules.

2 OF/DE 2

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

Chapter 5

Simulation Experiments

The performance of a file system is determined by two criteria: response time to access requests and storage utilization. Response time consists of the processing time in memory and disk access time. Since disk access time is much greater than the processing time in memory in most applications, the major component of this response time is the time spent in accessing disk. The maximal amount of data transferred in one access is fixed (a disk block or page). We will therefore measure the request efficiency in terms of the number of disk accesses per request. The storage utilization consists of the utilization of the data file associated with data pages and the utilization of the index file associated with the grid directory in our context.

Regnier in [32] presented the analysis of the grid file algorithms, but he made assumptions on the partition methodology and on the splitting policy of the grid file, which is not general to all the grid file design. A theoretical analysis of the grid file behavior appears to be difficult for two reasons [22] : many of the techniques developed for analyzing single-key data structures do not directly generalize to their multidimensional counterparts, and the grid file has parameters that are complicated to capture in a mathematical model (e.g., the different splitting and merging policies). For these reasons we resort to simulation to study the behavior of our implementation.

5.1 Objectives of the Simulation

The one level extendible directory grid file and the multilevel grid file have been implemented for experimentation. In our experimental set-up we consider only the case of the growing file, i.e. we insert records into the file, but do not delete them. Furthermore, we assume that each record is accessed with equal probability. When creating the file from an initial empty file, the following values are observed:

- S_b : the average storage utilization of a data page.
- S_i : the average storage utilization of an index page.
- F_{out} : the fan out of the grid directory.
- U : the average number of disk accesses required for a record insertion. We consider a disk access as either a disk read or write.

The average storage utilization of a data page is defined by $S_b = N_r / (c \times N_b)$, where N_r is the total number of records inserted into the grid file, c is the capacity of a data page and N_b is the number of current data pages.

The average storage utilization of an index page is measured according to the two different implementations defined above. In the one level extendible directory grid file, the index file consists of a one-dimensional array on disk, whose elements are pointers to data pages. The average number of the grid directory entries per page is a good measure of the efficiency of the grid directory. Therefore we indicate the storage utilization of index pages by its fan out, $F_{out} = N_e / N_b$, where N_e is the number of entries in the directory and N_b is the number of data page. The higher the fan out, the lower storage utilization of index pages. In the multilevel grid file, we consider the fan out of the directory as well as the average utilization of an index page. From the description of the method, each index page contains a header, a

one-dimensional array and a set of scales. The average utilization of an index page is measured by

$$S_i = \frac{U_b}{\sum_{i=1}^n (B_i * P_i)}$$

where, U_b is the total number of used bytes in directory pages, n is the number of different directory page sizes. For each directory page size i , B_i is the number of bytes in the directory page and P_i is the number of directory pages.

The number of disk accesses for a record insertion consists of two parts; one for searching for the data page that the record should lie in, the other for inserting the record into the data page. The former part is equivalent to an exact match search. It takes two disk accesses in the one level extendible directory grid file. In the multilevel grid file, the number of disk accesses to locate a record depends on the depth of the multi-way tree.

In the following cases, each case requires a difference number of disk accesses to insert a record into a given data page.

1. In the case of no page overflow, inserting a record into a given page, in both the one level extendible directory grid file and the multilevel grid file, is only a *write* operation. If paging is used, no actual physical write to disk is performed.
2. In the case that an overflow of a data page occurs and an existing boundary is selected to split the data page, some directory elements may have to be modified to point to a newly created data page. The multilevel grid file modifies these elements in one disk access since these elements lie in one directory page. But in the one level extendible directory grid file, the modification may cause more than one disk access depending on the number of elements being modified and the positions of these elements in the directory.

3. In the case that an overflow of a data page occurs and a new boundary is created during splitting of the overflow data page, one has to modify the grid directory. The multilevel grid file has an advantage over the one level extendible directory grid file, since four disk accesses are required if the directory page simultaneously overflows. In the one level extendible directory grid file, the grid directory is extended by a new slice. The elements in the slice are acquired by copying n elements and modifying one of them, where n is the number of elements in the slice. The number of disk accesses for the *copying* operation depends on n and on the positions of the n elements in the directory.

In general, the one level extendible directory grid file is better than the multilevel grid file for a record retrieval but worse than the multilevel grid file for a record insertion.

The simulation runs also have the following objectives — exact match and range queries. Range queries include partial match and partial range queries. For the exact match query, at most one record can be found in our design since duplicate records are not stored in our design of the grid file. The number of disk accesses for an exact match query is measured in our experiment.

For range queries, we assume that R_Q is the number of records found by a range query, R_T is the total number of records in the grid file, P_Q is the number of pages accessed by the range query and P_T is the total number of pages in the grid file. A range query that retrieves R_Q records out of R_T records accesses a fraction R_Q/R_T of the grid file; if the grid file is stored as P_T pages, $(R_Q/R_T) \times P_T$ should be the ideal number of page accesses. Therefore the efficiency of a range query is defined to be

$$\frac{(R_Q/R_T) \times P_T}{P_Q}$$

The detailed information can be referred to [33].

5.2 Simulation Model

These simulations are aimed at studying the effect of such parameters as the number of dimensions and the data distribution on the performance characteristic in a paged memory environment.

We retain a 20 slot cache in memory. A page fault occurs if a referenced page is not resident in memory and is fetched from secondary storage. We apply the *leastrecentlyused* policy for the page replacement algorithm.

To demonstrate the performance of the one level extendible directory grid file and the multilevel grid file, we generated 6 files ($F1$) – ($F6$), each of which consisted of 10,000 records. Our aim is to test the performance of the algorithms on different distributions and on different dimensions. In the following, c is the capacity of a data page and d is the number of dimensions.

- (F1) $c = 12, d = 2$: with keys uniformly distributed in the 2-dimensional data space;
- (F2) $c = 12, d = 2$: with keys non-uniformly distributed in the 2-dimensional data space, where all points lie in a diagonal region of the attribute space;
- (F3) $c = 12, d = 2$: with keys non-uniformly distributed in the 2-dimensional data space, where all points lie in a circle region of the attribute space;
- (F4) $c = 12, d = 3$: with keys uniformly distributed in the 3-dimensional data space;
- (F5) $c = 12, d = 3$: with keys non-uniformly distributed in the 3-dimensional data space, where all points lie in a diagonal region of the attribute space;

(F6) $c = 12, d = 3$: with keys non-uniformly distributed in the 3-dimensional data space, where all points lie in a circle region of the attribute space .

Query experiments are also performed on the six of the grid files generated above. The exact match, partial match and range queries are separately tested. The exact match query is measured by randomly producing 2000 records in an indicated data space. We set up the partial match query according to random selected dimensions and random values in the dimensions. The range query is performed by specifying various ranges, and then randomly generating x queries with the given ranges, where x is defined in the following test cases. For each of the given range sizes, the results are averaged over x randomly generated queries. The range size is given as a percentage of the size of the entire data space. We have generated the following six query files for demonstrating the query performance:

- EQ1 : 2000 exact match query;
- PMQ1 : 200 partial match queries;
- RQ1 : 100 square/cubic range queries with volume 0.01;
- RQ2 : 20 square/cubic range queries with volume 0.25;
- RQ3 : 20 square/cubic range queries with volume 0.49;
- RQ4 : 20 square/cubic range queries with volume 0.72.

5.3 Simulation Results

5.3.1 Visualization of the Geometry of Partitions

First, we give a visualization of the distributions of points in the data space and the geometry of partitions. This helps to understand how our design works. We show how the grid file adapts its shape to different data distributions. Figure 5.1 shows

(a)

(b)

Figure 5.1: Visualization of Partitions of a Uniform Data Distribution

(a)

(b)

Figure 5.2: Visualization of Partitions of a Diagonal Data Distribution

(a)

(b)

Figure 5.3: Visualization of Partitions of a Circular Data Distribution

	<i>One Level Extendible Directory Grid File</i>	<i>Multilevel Grid File</i>
F1	0.65	0.67
F2	0.65	0.69
F3	0.65	0.68
F4	0.65	0.65
F5	0.65	0.67
F6	0.65	0.67

Table 5.1: Utilization of Data Pages

the page regions obtained after inserting 2000 records from a uniform data distribution. Figure 5.2 and Figure 5.3 show how the grid file "absorbs" non-uniformity where the data distribution is in the forms defined in $F(2)$ and $F(3)$. Figures 5.1(a), 5.2(a) and 5.3(a) show the results of data partitions processed by the one level extendible directory grid file. Figures 5.1(b), 5.2(b) and 5.3(b) show the partitions of the multilevel grid file. From these diagrams, the multilevel grid file obviously partitions the data space in a more robust manner. The regions with more data are split the most.

5.3.2 Utilization of Data Pages

Average page utilization need not be closed to 100%, but it must be prevented from becoming arbitrarily small under any circumstance. The splitting policy which partitions each overflow data page by the midpoint of the input record list, guarantees at least 50% utilization. The results of the experiments shown in Table 5.1 confirms this. The constant average page utilization observed above is achieved by our splitting policy. The utilization of the multilevel grid file is slightly higher than the one level extendible directory grid file. This is due to the fact that finer partitions occur in the subspaces which generally have uniform data distribution.

Figure 5.4: The Directory Fan-out in the 2-Dimensional Tests

5.3.3 Utilization of Index Pages

As indicated in section 5.1, the utilization of index pages is measured by the fan out of the grid directory. In the original grid file, correlated attributes which often cause a non-uniform distribution of data, are very likely to increase the directory size substantially. In the worst case, the growth of the directory is exponential in the number of data pages. In our design, the splitting policy guarantees that one partition is sufficient for any kind of non-uniform data distribution. Therefore, the directory size can be reduced dramatically. The Figure 5.4 shows the growth of the directory in the number of data pages during the insertion of 10,000 records from three 2-dimensional test files $F(1) - F(3)$ into data pages of capacity $c = 12$ in the one level extendible directory grid file and the multilevel grid file. Figure 5.5 shows the same measurements on the test files $F(4) - F(6)$. These graphs show the following three results:

1. The number of directory entries in the multilevel grid file is much less than that of the one level extendible directory grid file, especially in the case of

Figure 5.5: The Directory Fan-out in the 3-Dimensional Tests

non-uniform data distribution.

2. In the multilevel grid file, different data distributions for different dimensions do not influence the growth of the directory very drastically. The one level extendible grid file is very sensitive to data distributions and the number of dimensions.
3. From the diagrams, the fan out of the directory in these two schemes appears to be a linear function of the number of records. The exponential growth rate of the original grid file in the case of non-uniform data distribution disappears due to the new splitting policy.

The multilevel grid file exhibits a striking advantage in the directory growth over that of the one level extendible directory grid file.

Figure 5.6 and 5.7 depict 6 curves of the average directory page utilization with record insertions in the 6 test cases. The "rapid decreasing phenomenon" can be

Figure 5.6: The Directory Page Utilization in a 2-Dimensional Space

Figure 5.7: The Directory Page Utilization in a 3-Dimensional Space

	<i>One Level Extendible Directory Grid File</i>			<i>Multilevel Grid File</i>		
	<i>Read</i>	<i>Write</i>	<i>Read+Write</i>	<i>Read</i>	<i>Write</i>	<i>Read+Write</i>
F1	2.1	1.4	3.5	1.7	1.2	2.9
F2	2.1	1.6	3.7	1.7	1.2	2.9
F3	2.4	1.7	4.1	1.7	1.2	2.9
F4	2.3	1.6	3.9	1.7	1.2	2.9
F5	3.3	3.3	6.6	1.7	1.2	2.9
F6	2.5	2.1	4.6	1.7	1.2	2.9

Table 5.2: Average Disk Access for Insertion of a Record

explained as follows. In the multilevel grid file, each directory page generally corresponds to a uniform data space. There are moments when almost every grid cell in a directory page has its own page and the directory page is full. With further insertion these data pages overflow. The resulting partition refinements require that we create new children, leading to a rapid increase in the number of directory pages i.e. rapid decrease in the directory page utilization. From the graph, we also observe the directory page utilization in the non-uniform data distribution is better than the directory page utilization in the uniform data distribution. This results from the tree growing from top to bottom.

5.3.4 Disk Accesses for Insertion

Table 5.2 shows the average number of actual disk accesses required per key insertion for the one level extendible directory grid file and the multilevel grid file. The table shows the following results:

1. In the multilevel grid file, the *read* and *write* times for inserting a record is constant among all cases. It is very easy to understand the constant for the *write* operation since there is an upper bound on the number of disk accesses during inserting a record into a given data page. For the *read* operation, the

result shows the multi-way tree created in our experimentation has almost the same depth for all the branches. This is due to our splitting policy. When the amount of records processed becomes very large and data distribution is heavily non-uniform, the read time for inserting a record will be increased. Furthermore, the dimensionality of keys does not significantly affect the performance of the insertion.

2. In the one level extendible directory grid file, the more non-uniform data distribution is, the more disk accesses are required. And the number of dimensions significantly affects the performance of the insertion. The higher dimension needs more disk accesses than the lower dimension because it maintains more entries in the directory.
3. In the case of a limited depth of the multi-way trees, the performance of insertion in the multilevel grid file is much better than the performance of insertion in the one level extendible directory grid file.

5.3.5 Query Performance

The results of our experimentation on query processing are given in Table 5.3. Note that the exact match query is an average of 1.98 disk access in the one level extendible directory grid file, and an average of 1.91 disk access in the multilevel grid file. The reason of less than 2 disk accesses in the one level extendible directory grid file is due to a paged memory for the grid directory. The good performance achieved in the multilevel grid file is because the root page of the multilevel grid file is kept in memory during a session and the depth of the multi-level tree is 3 in our experimentation. From the above results, we can not conclude the performance of the exact match query for both schemes is the same. The exact match query is guaranteed to be at most two disk accesses in the one level structure, but in the multilevel structure it depends on the depth of the multi-way tree.

		<i>EQ1</i>	<i>PM1</i>	<i>RQ1</i>	<i>RQ2</i>	<i>RQ3</i>	<i>RQ4</i>
<i>One Level Extendible Directory Grid File</i>	<i>F1</i>	1.97	0.003	0.22	0.33	0.31	0.35
	<i>F2</i>	1.98	0.003	0.19	0.26	0.27	0.26
	<i>F3</i>	1.98	0.003	0.21	0.21	0.24	0.26
	<i>F4</i>	1.98	0.010	0.17	0.28	0.26	0.36
	<i>F5</i>	1.98	0.004	0.08	0.13	0.11	0.10
	<i>F6</i>	1.98	0.008	0.08	0.08	0.16	0.17
<i>Multi-Level Grid File</i>	<i>F1</i>	1.96	0.005	0.48	0.77	0.81	0.82
	<i>F2</i>	1.92	0.006	0.57	0.85	0.87	0.89
	<i>F3</i>	1.89	0.006	0.55	0.82	0.84	0.89
	<i>F4</i>	1.99	0.013	0.29	0.54	0.62	0.70
	<i>F5</i>	1.80	0.011	0.41	0.78	0.83	0.82
	<i>F6</i>	1.92	0.012	0.37	0.61	0.71	0.76

Table 5.3: Summary of Query Performance

For range queries, the performance of the multilevel grid file is much better than that of the one level grid file in all cases. Furthermore, the performance in the one level grid file scheme is sensitive to data distribution and the number of dimensions. The multilevel scheme gives better performance in non-uniform data distribution. It should be noted that the performance of the multilevel grid file may be degraded when the amount of data processed is very large, i.e., the depth of the multi-tree is very high. The experimental results also indicate that the partial match query has the worse performance compared with a range query.

Chapter 6

Further Discussion

In this chapter we discuss two issues. The first is how the grid file system can be used for storing spatial objects. The second is how the grid file supports the implementation of relational join operation.

6.1 Methods of Object Storage Using the Grid File System

The grid file has been designed primarily for storing points in a multidimensional space. In geographic and CAD applications as well as in VLSI design, often large sets of geometric objects which may be lines, regions and solids, have to be organized in a file system. The goal of the file organization for spatial objects is to support efficient range queries such as intersection and containment queries. In addition, it is desirable that the storage scheme reflects as many topological and geometric properties of spatial objects as conveniently possible, such as the type of an object, its location in space or bounds on its spatial extension [9]. To achieve the goals, the file structures for storing multidimensional point data which partition the underlying data space into cells are adopted for storing spatial objects. To this end, the data space is divided into subspaces and a data page is associated with each subspace. All objects in a subspace are stored in the corresponding data page.

Figure 6.1: Approximation: A Technique to Represent Spatial Objects

Before introducing the techniques for storing spatial objects, it is necessary to define spatial objects in our context. In most practical applications, the appearances of spatial objects can be very complicated. There is one known technique used to approximate complex spatial objects into simpler ones. An irregularly shaped spatial object can be approximated by enclosing it in a container chosen from a class of simple shapes [9]. Figure 6.1 shows an approximation of polygons by rectangles. These simple primitives, such as boxes, spheres and cylinders can be determined by a fixed number of parameters such as a set of intervals. Therefore, the techniques for storing spatial objects only need to handle these simple objects. There are three major methods [37] used for storing spatial objects in the grid file system. We summarize them in the following sections.

6.1.1 Reducing Object Storage to Point Storage

Any simple object, defined by d parameters, can be interpreted as a point in a d -dimensional space in order to use point data structures in a standard way. For example, an aligned rectangle (i.e., a rectangle with sides parallel to the axes) can be described by its 4 parameters. These are the left bottom point (l_x, l_y) and right top point (r_x, r_y). This rectangle can be considered to be a point in 4-dimensional space. Objects, which are defined by different number of parameters, are then stored in different grid files. Inserting or deleting an object corresponds to inserting into or

Figure 6.2: A Partition of Spatial Objects in the Grid File

deleting the corresponding point from a grid file. Objects are found by performing a range query on each of the grid files. The advantage of this kind of the representation of spatial objects is that no additional algorithms are needed to handle spatial objects. However, this approach suffers from the inherent disadvantage that d -dimensional points may be arbitrarily far apart, even if they correspond to objects that are (at least partially) close neighbors. That is, the spatial neighborhood of objects is not appropriately preserved under this interpretation. As a consequence, a range query on intervals may become an extremely inefficient operation on the corresponding points. This method has been realized with the grid file by Hinrichs in [9].

6.1.2 Clipping Method

Objects are stored according to their original meaning, for example, a polygon by the set of its intervals. The data space is partitioned into pairwise disjoint subspaces as the grid file processes points. Figure 6.2 illustrates a corresponding partition of the data space containing rectangles. Problems are caused by rectangle r , shown with dashed boundary, because it cannot be associated uniquely with one subspace. In this method, object r is clipped into some pieces and stored in all data pages whose

associated subspaces intersect the intervals. This causes the redundancy of information on object r . If the redundant information is the entire description of objects, the space requirements increase substantially, i.e., more data pages are needed. Since data pages correspond to subspaces, an increasing number of data pages induces a finer partition of the data space. The finer the partition, however, the more the objects have to be clipped; more clipping, on the other hand, leads to a finer partition, etc. The resulting high degree of redundancy reduces the efficiency of operations. The insertion, deletion, or update of a simple object may affect many data pages. Also, range queries become substantially less efficient, because objects usually will be reported more than once. In order to solve the problem of redundant information, it is proposed that objects be stored via one representative point, such as its center of gravity. The information that a given point in space is the representative point of a stored object conveys no further information (such as the type of object it is, or its extent) without accessing the full description of the object which we assume is stored on disk. In this case, the range query is poorly supported because not all data pages associated with subspaces intersecting object r , information on r is stored. Then r cannot be found (without extra data page accesses) in any range query intersecting only subspaces whose associated data pages do not contain information on r .

To alleviate the severity of these problems, it seems promising to copy only parts of object description, instead of entire description. For instance, one can copy the geometric key of a clipper object, and store the rest of the description elsewhere.

6.1.3 Overlapping Method

The third method is different from the proceeding one in processing object r which falls into splitting lines of the data space. Suppose that the set of rectangles shown in Figure 6.2 is stored in grid file G_1 . The capacity of data pages of this grid file is three rectangles. When object r falls into the splitting lines in grid file G_1 , it cannot be stored in G_1 without clipping. Therefore, r is stored in an additional grid file G_2 . Note that now there are two grid files, G_1 and G_2 , each of which stores a

Figure 6.3: Overlapping Method of Storing Spatial Objects in the Grid File

subset of rectangles in such a way that no rectangle is clipped. Some rectangles are stored in G_1 if they fit entirely into a subspace of G_1 , others intersect page region boundaries in G_1 and therefore are stored in G_2 . Of course, in G_2 clipping is also avoided. Nevertheless, it is possible that a rectangle may fall onto a splitting line in G_1 and one in G_2 . Then it has to be stored in the third grid file G_3 . Figure 6.3 shows an example of three partitions of a data space by three grid files. There is a subdivision of the data space into overlapping page regions by overlaying three grid files, the bottom most layer being G_1 , and the top most G_3 . The data space in each layer is still partitioned into pairwise disjoint subspaces.

In general, the overlapping method creates a new grid file with higher layer G_i when an object can not fit any page region in all existing grid files G_j ($1 \leq j < i$). Therefore the data space is divided into overlapping subspaces. Object clipping is avoided. Each object is entirely contained in a subspace. This method is proposed in [37]. Its drawback is that a range query becomes an inefficient operation. This is because a query is answered for each layer. All data pages corresponding to the subspace containing the query range in each layer may have to be inspected.

6.2 An Application for Storing Maps in the Grid File System

In this section, we present an application of the grid file system. When the idea of using computers to store maps was proposed, a traditional map sheet was generally transformed into a base map and a set of layers, each of which holds a specific subject information in the same area. For example, a map can consist of a base map and two layers with the subjects *soil* and *water* for each layer respectively. The advantage of this kind of representation is that a variety of layers can be combined to form different maps for specific users.

Both base map data and layered data are represented by a mixture of point, line and volume types of geographic objects. Each object consists of two descriptions. One is a geometric description which defines the topologic information on the object; the other is a non-geometric description such as feature codes of objects.

The base map includes the basic information such as hydrology, administrative boundaries, transportation features, etc. The characteristics of base map data are: 1) the information contents of large area base maps are very high and the resulting data files are extremely voluminous; 2) the base map data does not need to be modified very often and 3) the data is usually non-uniformly distributed. Therefore, the data base for storing base maps can be thought of as a large sized static data base. Dynamic map modification is of no concern in base maps. Objects can be subdivided along grid borders. Our aim is to provide a structure for fast retrieval of base map elements.

In contrast to the large sized static base map data, layers represent another kind of information in the geographic database. Its characteristics are: 1) each layer contains information on a small area of a map; 2) the amount of data varies with different applications and 3) frequent data modifications are needed. In order to

support frequent data updates, objects in layered data such as lines and areas are expected to be stored within one single data page.

According to the characteristics discussed above, we present two storage schemes for base map data and layered data respectively.

6.2.1 Storing Base Map Data

Elements of base map data are separated at grid cell borders. A reconstruction of object integrity and topology is not required. Information pertaining to adjacent objects should be stored whenever possible in one data page. The purpose is to design the storage strategy for the base map data so that a fast search of base map elements is achieved and disk storage space is minimized.

The first problem associated with the storage strategy is to select a method to store spatial objects. As discussed previously, three major methods are used for storing spatial objects in the grid file system. The method of reducing object storage to point storage does not appropriately preserve the spatial neighborhood of objects. The overlapping method results in high query costs since each object is placed in a data page. The clipping method is best suitable for base map data. In this method, each type of objects are represented by a set of characteristic parts – a point by a coordinates (x, y) , a line by a set of segments, (each of which described by its left end point (l_x, l_y) and right end point (r_x, r_y)), and a polygon by a set of lines. Therefore, data pages contain base map elements in the form of points or a chain of points. The geometric description of an object, together with a reference key, is stored in all data pages whose associated subspaces intersect the object. The non-geometric part of the object is stored somewhere else. The inefficiency of insertion, deletion and update of the clipping method is diminished due to the static nature of base map data,

Figure 6.4: A Multilevel Grid File Model for Base Map Data

The second problem associated with the storage strategy is to select a type of the grid file. According to the requirement of efficient searching for base map elements, both the one level extendible directory grid file and the multilevel grid file appear to be optimal. The class of the grid files based on the interpolation idea changes the structure of the grid directory. As a result, range query processing becomes less efficient. The fact that base map data used for mapping or spatial analysis is highly clustered suggests that the multilevel grid file is the most efficient for this task. It minimizes the storage space by physically clustering adjacent data. This concept is illustrated in Figure 6.4.

6.2.2 Storing Layered Data

The objects in layered data have to be handled as integral entities. Similar to the discussion of storing base map data, the storage strategy for layered data is in two fields.

The first field is to select a method to store the spatial objects. Both the method of reducing object storage to point storage and the overlapping method appear to be optional since they all guarantee each object to entirely fit in a data page. However, there is a severe problem associated with the representation of lines and areas in the method of reducing object storage to point storage. For example, a line consists of a set of segments. When mapping a line into a point by using this method, the number of dimensions corresponding to the point storage should be determined by the number of segments in the line. In geographic applications, it is less possible to have the lines with the same length, thus a large amount of grid files with different dimensions are created for storing lines with different length. Therefore, each grid file may contain a few records with a large number of dimensions. These problems make it difficult to store geographic objects with the method of reducing object storage to point storage. The overlapping method represents objects according to their original meaning. However, in the original overlapping method, the number of layers of the grid file is unlimited. This is not practical. We pursue a symbiotic combination of both, where clipping as well as overlapping is allowed to a limited extent. The overlapping method uses $k + 1$ grid files from layer L_1 to L_{k+1} . The grid file at layer L_i is created when an object can not fit any page region in any existing grid file G_j , where $1 \leq j < i$. Objects that do not find a place in any of the $k + 1$ layers are treated as a set S . The clipping method can then be used to organize the objects in S .

The second field of the storage strategy is to select a type of the grid file. According to the characteristics of layered data, no particular grid file scheme is suggested. Final decision on the scheme of the grid file depends on the specifics of the layered

data to be recorded and the circumstance of their use. A major decision will depend on the amount as well as the distribution of layered data.

Compared with other multidimensional file organizations, the grid file structure is well suitable for map processing due to its symmetric attribute structure and practical performance.

6.3 Grid File Join

Most contemporary relational database management systems use the index sequential file organization to provide several single-key access paths to the stored data. Consequently, a variety of different access paths and algorithms must be analyzed to find the optimal way for query evaluation. The grid file data structure replaces a collection of individual access paths by a set of grid scales and a single grid directory. Using these two structures, it is possible to limit very precisely the extent of a relation scan by simultaneously considering the constraints on all the attributes specified by the query. The research in this field can be seen in [9, 34, 19]. In this section, we first define the grid file join model and present two major join methods. Then a join algorithm and its implementation in the one level extendible directory grid file and the multilevel grid file are presented.

6.3.1 The Grid File Join Model And Two Major Join Methods

Following the grid file join model taken by Rotem [34], we define a grid file join model as follows. Given two files F_1 and F_2 , an abstract join operation can be viewed as creating a bipartite graph $G(V_1, V_2, E)$ where V_i corresponds to a set of elements from F_i , for $i = 1, 2$, and each edge in E corresponds to two elements one from each F_i which be joined based on some join predicate. In the concept of the grid file system, file F_i corresponds to a grid file and the elements are data pages. An edge between two data pages B_1 and B_2 , where $B_i \in F_i$, is defined if at least

one record from B_1 matches a record in B_2 based on the join predicate.

The grid file in our discussion is thought of as a data organization scheme for the multidimensional point data. The join predicate can be defined as the distance between points. This type of join operation is useful in applications which require finding points in one file lying within a given distance ε from the points in another file.

Given two rectangles, we say that they have an ε -overlap if there exist two points one in each rectangle whose distance from each other is less than or equal to ε . When $\varepsilon = 0$ this definition coincides with the usual definition of overlap. As already discussed about the data structures of the grid file, each data page corresponds to a page region in the data space. Also each page region corresponds to a rectangle region of directory cells all pointing to that data page. Let F_1 and F_2 be two grid files. Two data pages, one from F_1 and the other from F_2 , will be connected by an edge of the bipartite graph if the corresponding page regions which they represent have an ε -overlay for some ε defined by the join operation. Two join methods have been proposed:

1. **The Page Oriented Join Method** - Using the method, a join query is performed as follows. All data pages of the grid file F_1 are processed by traversing its grid directory in the row (or column) major order. Let n_1 be the number of pages in F_1 and n_2 be the number of pages in F_2 . For each distinct data page B_i^1 , a search takes place for pairs of pages (B_i^1, B_j^2) satisfying ε -overlap, where the superscript represents the grid file which the data page comes from, $1 \leq i \leq n_1$ and $1 \leq j \leq n_2$. If there exist such pairs of data pages, these two pages (B_i^1, B_j^2) will be connected by an edge of the bipartite graph. Note that F_1 and F_2 are not handled symmetrically in this implementation. During the join query, each page belonging to the grid file F_1 is accessed from disk at most once, whereas data pages belonging to F_2 may be accessed more than once.

It is easy to design an algorithm which generates the bipartite graph for two grid files and runs in time $O(n_1 \times n_2)$. Simply, for each page region of F_1 , we can check which page regions of F_2 have an ε -overlap with it and insert the corresponding edges into the bipartite graph. The lower bound for this algorithm is clearly proportional to the total number of page region overlaps or $O(E)$ where E is the number of edges in the join graph [34]. The naive algorithm is therefore optimal if the join graph is dense. However, in most cases we expect the graph to be sparse and therefore a faster algorithm is desirable. As indicated in [9], if there is a big enough memory to buffer some pairs of data pages, the performance of a join query varies depending on the sequence of the page pair processing. The problem of minimizing the number of disk accesses in a join query can be represented as a special case of the Hamiltonian path problem and thus it is shown to be *NP*-complete. Heuristic procedures are given in [21] for scheduling the processing of the pairs in order to obtain near optimum solutions. It is an open problem to find an optimal schedule for the page pair processing if more than two pages of the two grid files involved can be buffered,

2. **The Slice Oriented Join Method** - This method is based on the fact that each grid file can be divided into slices, which are defined by the grid scales. To compute the join of two grid files the slices are processed pairwise: one slice of grid file F_1 and one slice of grid file F_2 . In each pair of ε -overlapping slices, the ε -overlapping regions are detected. In this way the computation of the join is transformed into a sequence of join operations of pairs of slices which are much smaller than the original grid files. Only those pairs must be processed that could contain points with an ε -overlap.

6.3.2 An Algorithm for Grid File Join

We now describe an algorithm to generate a bipartite graph from two given grid files and then present its implementation of the algorithm in the one level extendible directory grid file and in the multilevel grid file. The basic idea of the algorithm is based on [34]. We make modifications on identifying the sweeping line, candidate regions and overlapping regions.

Any geometrical relationship between the rectangular areas covered by the two grid files is possible. For simplicity, we will describe the algorithm only for $\varepsilon = 0$ and two dimensional grid files.

This algorithm is developed from the slice oriented join method. Let us assume that one grid file is super-imposed on top of the other one to obtain a single grid graph which includes all grid cells (see Figure 6.5). The idea is to use a technique in computational geometry in which an imaginary line called a sweeping line, parallel to the y axis, scans the super-imposed grid graph from left to right. At each point in time, all page regions cut by the line are called candidate regions. A region stops being candidate when all its points are to the left of the sweeping line. It can be observed that if two regions have an overlap they must be simultaneously candidate at some point during the execution of the algorithm as the sweeping line must cut their common intersect area.

For that reason, we need to maintain a list of the current candidate regions for each grid file F_i , and add or delete page regions from it as the sweeping line enters or leaves them respectively. Edges are added to the bipartite graph whenever it is detected that a region r from F_i , which just became candidate, has an overlap with any of the other regions on the candidate list for F_j where $i \neq j$.

To implement the algorithm, the following information should be able to be provided by the grid file data structure.

$$h_1^1 \ h_1^2 \quad h_2^1 \ h_2^2 \quad h_3^2$$

Figure 6.5: Superimposed Method for the Grid File Join

1. **Sweeping Line** - There are two problems associated with the sweeping line. One is the moving direction of the sweeping line, i.e., along x axis or y axis in a 2-dimensional grid file. It is obvious that the decision is taken with respect to the dimensions in join attributes. In most cases, there is more than one join attribute; the sweeping line moves along one of the dimensions in the join attributes. Another problem is the selection of detection points. In the grid file, the boundaries of page regions are defined by a set of hyperplanes, which are stored in the scales. Note that the boundaries of whole data space are also thought of as hyperplanes. Analog to the technique in computation geometry where detection points are selected at the start and end points of lines, a set D of detection points consists of all hyperplanes on dimension s in both grid files, where s is the dimension which the sweeping line moves along. In order to scan systematically the set of detection points, we sort the hyperplanes in D to obtain the sorted set $\tilde{D}$.

$$\tilde{D} = \langle h_1, h_2, \dots, h_m \rangle$$

h_i is a hyperplane on dimension s in F_1 or F_2 , $1 \leq i \leq m$, and m is the sum of the number of hyperplanes on dimension s in F_1 and F_2 .

For example, in Figure 6.5, if a sweeping line moves along dimension x , the set $\tilde{D}$ is as follows

$$\tilde{D} = \{h_1^1, h_1^2, h_2^1, h_2^2, h_3^2\}$$

where the superscribe represents the grid file where the hyperplane comes from, and the subscribe represents the order of hyperplanes in one grid file. h_1^1 and h_1^2 represent the left boundaries of the data spaces of F_1 and F_2 respectively. They are added to the set of the detection points since it is defined that a page region becomes a candidate when its lower boundary is the sweeping line. The detailed discussion of candidate regions follows. The right boundaries of these

two data spaces do not need to be considered since the boundaries do not induce any new candidate regions.

2. **Candidate Regions** – The current candidate region list R_i define all page regions in grid file F_i . These should be considered to possibly intersect the page regions in the current candidate region list R_j in terms of the current sweeping line, where $i \neq j$. The algorithm simulates the sweeping line motion by checking at each successive h_i which region has become a candidate or stopped being a candidate. According to the activity of the sweeping line discussed above, the current candidate regions R_i in F_i includes all page regions whose lower boundaries are the sweeping line l_{h_i} where h_i is a detection point in $\tilde{D}$ and h_i comes from F_i , i.e., h_i is a hyperplane on dimension s in F_i . For example, in Figure 6.5, $R_2 = \{a, b\}$ when the sweeping line is at h_1^2 . The remaining problem is how to determine the current candidate region R_j ($j \neq i$). We observe that h_i must either cut a slice $S(i, s)$, where s is the dimension in which the sweeping line moves along and i is an index interval on dimension s in F_j , or be also a hyperplane in F_j . In the former case, the current candidate regions R_j consists of all page regions which intersect the grid cells in the slice of F_j . For example, $R_1 = \{0, 2\}$ when the sweeping line is at h_1^2 . In the latter case, the current candidate regions R_j consists of all page regions whose left boundaries are h_i in F_j .
3. **Overlapping Regions** – If two regions have an overlap, they must simultaneously appear in the corresponding current candidate regions. At each detection point, the candidate regions for each grid file are updated first. We then need to call a detection routine to find out whether new overlaps are introduced. According to the property of the grid file -- attributes are treated symmetrically, the detection of overlap regions can be processed in the same way as creating candidate regions. The process of creating candidate regions can be thought of as looking for the overlap slices in a different dimension. Therefore to detect the overlap regions, another imaginary sweeping line moves on an

other dimension s' of join attributes. The detection points also consist of all hyperplanes on dimension s' of both grid files. In the process of creating candidate regions, a routine is invoked to produce two candidate regions. In the process of detecting overlap regions, a routine is invoked to examine whether two regions have an overlap.

There is a problem associated with how to implement the above algorithm in the one level extendible directory grid file and in the multilevel grid file since page regions are not explicitly represented in the grid file. Given a slice, we can only know a set of grid cells in it. By comparing the values of these cells, it is known that some cells lie in a page region. In the following two sections, we describe how the algorithm is efficiently implemented in the one level extendible directory grid file system and in the multilevel grid file system.

6.3.3 Join of Two One Level Extendible Directory Grid Files

In the one level extendible directory grid file system, the grid scales are defined by binary trees which are kept in memory. Without loss of generality, it is assumed that the sweeping line moves along dimension k_0 . Set $\tilde{D}$ can be generated by a tree pre-order search algorithm. During the pre-order search, we maintain two pointers, pr_0 and pr_1 , pointing to the current internal nodes respectively in the two binary trees which correspond to dimension k_0 in the two grid files. The sweeping line starts at the left boundary of the data space which has a smaller value between these two grid files. In order to obtain the successive detection point, the *keys* in pr_0 and pr_1 are compared. The hyperplane with the smaller value of these two *keys* is selected as the successive detection point and the corresponding pointer pr moves to the next node according to the pre-order search algorithm. In each detection point h_i , the following two steps are taken:

1. **Creating candidate regions** – All grid cells whose left boundaries are h_i in grid file F_i are read from disk and their values are inserted into the set R_i .

where h_i comes from F_i . The duplicate values of grid cells are inserted once. By searching for h_i in the binary tree of dimension k_0 in F_j ($j \neq i$), a slice is located. The values of all grid cells in the slice are inserted into the candidate regions R_j . If h_i is less than the lower boundary of the data space of F_j , the sweeping line moves to the successive detection point in F_i . If h_i is larger than the upper boundary of the data space of F_j , the algorithm terminates.

2. **Detecting overlap pages** - The directory in the one level extendible directory grid file is stored on disk. If the method discussed above is used to detect overlap pages, a large number of disk accesses are required. To reduce the disk accesses, one auxiliary structure is constructed. This auxiliary structure is defined as

$$\langle \text{value}, k_1^b, k_1^t \rangle$$

with the following interpretation:

value : a pointer to the data page.

k_1^b : the bottom coordinate of the page region in dimension k_1 .

k_1^t : the top coordinate of the page region in dimension k_1 .

The data structure defines the size of page regions in dimension k_1 . When each grid cell is added into the corresponding candidate region R , it is checked if the grid cell has already been in R . If it is in R , k_1^b and k_1^t will be modified to represent a larger coverage of the grid cell in dimension k_1 . Otherwise, the k_1^b and k_1^t are set to the bottom and top coordinate of the grid cell in dimension k_1 and inserted to R together with the value of the grid cell. By using the data structure, the process of detecting overlap pages is done by comparing the k_1 coordinates in both page regions. No extra disk access is required.

6.3.4 Join of Two Multilevel Grid Files

In the multilevel grid file, the scales and the grid directory are stored in directory pages. When looking for candidate regions and detecting overlap pages, disk accesses are not needed. What needed to be considered is how to handle the hierarchical structure of the multilevel grid file. In each directory page, the same procedure as the above is performed, i.e., selecting the successive detection point, creating candidate regions and detecting overlay pages. Both joined grid files started simultaneously from the root pages in the multi-way trees. Once a pair of pages (either a directory page or data page) is detected to have an overlap, a routine is invoked to process the following cases.

1. CASE 1 – If both pages are data pages, then an overlap is reported.
2. CASE 2 – If one of them is a data page, say *B*, and the other one is a directory page, say *P*, the join operation is transformed to a range query in grid file *F* which *P* comes from. It is because the data page *B* defines a query region *R* by its boundary. All data pages, which lie under directory page *P* in the multi-way tree of grid file *F* and intersect with region *R*, will be thought of as having an overlap with the data page *B*.
3. CASE 3 – If both pages are directory pages, the join operation can be recursively processed as both of them are root pages in two sub-multi-way trees.

The process of the relational join operation requires the identification of the smallest possible portion of physically stored relations such that all the records that satisfy the join predicate can be found in this portion. The grid file data structure offers the possibility to restrict the search region of each relation by simultaneously considering the constraints on all the attributes specified by the join query.

Chapter 7

Conclusions

In this thesis, we have presented two generalizations of the grid file. The one level extendible directory grid file has all the inherent advantages of the original grid file. It has a guaranteed two-disk access to locate a record. Compared with other multidimensional file organizations, it is efficient in processing range queries with respect to all the attributes. In addition, our design overcomes a major performance penalty that the grid directory needs to be reorganized whenever the directory is split or merged. Our simulation shows that the one level extendible directory grid file exhibits a reasonable utilization of data pages and a linearly increasing directory growth with data size. Especially in the uniform data distribution, a low *fan out* of the directory is achieved. The number of disk accesses for inserting a record is decreased since no reorganization of the directory is required.

Non-uniform data distribution is most likely to be encountered in practice and especially in engineering databases. By refining the grid directory in memory and partitioning the data space according to the region of high data density, the multilevel grid file requires less number of disk accesses for insertion and deletion than the one level extendible directory grid file. It also reduces the directory size considerably. Furthermore, the multilevel grid file has very good performance on processing range queries since the hierarchical structure combined with the robust splitting policy supports a large amount of data stored in the multilevel grid file with a reasonable

depth. Therefore, most applications are expected to achieve good performance by using the multilevel grid file. Two techniques are used in our implementation to alleviate possibly poor utilization of directory pages.

Our new splitting policy partitions a subspace entirely according to the data distribution. As a result, when the data distribution is non-uniform, the directory grows linearly with the number of records inserted. Our splitting policy guarantees a lower bound on the utilization of data pages. Simulation experiments with one level extendible directory grid file and multilevel grid file conform these. By using a merging policy that is an exact inverse procedure of the splitting process, the scheme avoids the spatial deadlock problem. This leads to more efficient merging operation.

The grid file can be used as a storage model for spatial data. The object storage has been considered under this model. By applying the grid file for cartographic map storage and spatial objects, we have demonstrated how versatile the grid file is in its adoption for different applications. The method we proposed is a meaningful attempt to apply the idea of the grid file to the geographic information system and map processing. Based on some previous work, we have demonstrated how the grid file is used in the implementation of traditional relational join operation.

We believe that the contribution of this thesis is a step further towards the design of multidimensional file organizations implementable in practical systems where data is more likely to be non-uniformly distributed. We remark that the balanced multilevel grid file is desirable as the first step towards achieving a good utilization of directory pages and maintaining a balanced multilevel tree structure. To fully comprehend the significance of the balanced multilevel grid file, further investigation is necessary. Other future work is in parallel operations on the multilevel grid file system.

References

- [1] Bentley, J.L.: *Multidimensional Search Tree Used for Associative Searching*, Communication ACM, 18, 9 (1975), 509-517
- [2] Bentley, J.L.: *Multidimensional Binary Search Trees in Database Applications*, IEEE Trans. Software Engineering, SE-5, 4 (1979), 333-340
- [3] Burkhard, W.A.: *Interpolation-based Index Maintenance*, Proc. 2nd Symp. on Principles of Database Systems (1983), 76-89
- [4] Chun, Sei H., Hedrick, G. E., Lu, Huizhu and Fisher, D.D.: *A Partitioning Method for Grid File Directories*, IEEE International Conference on Data Engineering (1991), 271-277
- [5] Freeston, M.: *The BANG File: a New Kind of Grid File*, Proc. ACM SIGMOD (1987), 260-268
- [6] Fagin, R., Nievergelt, J. and etc.: *Extendible Hashing - a Fast Access Method for Dynamic Files*, ACM Trans. on Database Systems, Vol. 4, No. 3, Sept. 1979, 315-344
- [7] Finkel, R.A., and Bentley, J.L.: *Quad Tree - A Data Structure for Retrieval on Composite Keys*, Acta Informatica; 4, 1 (1974), 1-9
- [8] Guting, H., Kriegel, H.P.: *Multidimensional B-tree: An Efficient Dynamic File Structure for Exact Match Queries*, Springer, Berlin-Heidelberg-New York (1980) Proc. 10th GI Annual Conference, Informatik-Fachberichte 33, 375-388

- [9] Hinrichs, K.H: *The Grid File System: Implementation and Case Studies of Application*, Ph.D. Dissertation, Swiss Federal Institute of Technology, Zurich, 1985
- [10] Hutflesz, A., Six, H.-W. and Widmayer, P.: *Globally Order Preserving Multidimensional Linear Hashing*, Proc. of the 4th IEEE International Conference on Data Engineering, Los Angeles, (1988), 572-579
- [11] Kriegel, H.P.: *Performance Comparison of Index Structures for Multi-key Retrieval*, Proc. ACM SIGMOD International Conference on Management of Data (1984), 186-196
- [12] Kriegel, H.P., Seeger, B.: *Multidimensional Order Preserving Linear Hashing with Partial Expansions*, Proc. International Conference on Database Theory, Sept 8-10 (1986), 203-220
- [13] Kriegel, H.P., Seeger, B.: *PLOP-Hashing: A Grid File without Directory*, Proc. of the 4th IEEE International Conference on Data Engineering, Feb. 1988, 369-376
- [14] Kriegel, H.P., Seeger, B.: *Multidimensional Dynamic Quantile Hashing Is Very Efficient for Non-uniform Record Distributions*, Proc. International Conference on Data Engineering (1987), 10-17
- [15] Krishnamurthy, R., Whang, K.-Y.: *Multilevel Grid Files*, Research Report, IBM Research Lab., Yorktown Heights(1985)
- [16] Larson, P.-A: *Linear Hashing with Partial Expansions*, Proc. 6th International Conference on Very Large Database (1980), 224-232
- [17] Larson, P.A.: *Dynamic Hashing*, BIT 18 (1978), 184-201
- [18] Litwin, W.: *Linear hashing : a New Tool for Files and Table Addressing*, Proc. 6th International Conference on Very Large Database, 1980, 212-223

- [19] Mahnic, Viljan: *Using Grid File for a Relational Database Management System*, Microprocessing and Microprogramming, 31 (1991), 13-18
- [20] Merrett, T. H. and Otoo, J. E.: *Dynamic Multipaging: a Storage Structure for Large Shared Data Banks*, Proc. 2th International Conference on Databases. Improving Usability and Responsiveness. Academic Press (1982), 237-256
- [21] Merrett T. H., Kambayashi Y. and Yasuura H.: *Scheduling of Page-fetches in Join Operations*, Proc. 7th International Conference on Very Large Data Bases, (1981), 488-497
- [22] Nievergelt, J., Hintenberger, H., Sevick, K.C.: *The Grid File: an Adaptable, Symmetric Multikey File Structure*, ACM Trans. Database System, Vol.9, No 1, Mar, 1984, 38-71
- [23] Orenstein, Jack A.: *Multidimensional TRIEs Used for Associative Searching*, Information Processing Letters, Vol. 14, 4 (1982), 150-157
- [24] Orenstein, Jack A.: *A Class of Data Structures for Associative Searching*, Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, (1984), 181-190
- [25] Otoo, E.J.: *A Mapping Function for the Directory of a Multidimensional Extendible Hashing*, Proc. 10th international Conference on Very Large Data Bases, Singapore(1984), 493-506
- [26] Otoo, E.J. : *A Multidimensional Digital Hashing Scheme for Files with Composite Keys*, Proc. ACM SIGMOD International Conference on Management of Data, (1985) 214-229
- [27] Otoo, E.J.: *Balanced Multidimensional Extendible Hash Tree*, Proc. 5th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems (1986), 100-113
- [28] Otoo, E.J.: *Linearizing the Directory Growth in Order Preserving Extendible Hashing*, Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems (1986), 580-588

- [29] Ouksel, M.: *The Interpolation Based Grid File*, Proc. 4th ACM SIGACT-SIGMOD Symposium on Principles of Data Base Systems, 1985, 20-27
- [30] Ouksel, M and Mayer Otto: *The Nested Interpolation Based Grid File*, Research Report, Universitat Kaiserslautern. (1990)
- [31] Ouksel, M. and Scheuermann, P.: *Storage Mapping for Multidimensional Linear Dynamic Hashing*, Proc. 2th ACM SIGACT-SIGMOD Symposium on Principles of Data Base Systems, 1983, 90-105
- [32] Regnier, M.: *Analysis of Grid File Algorithms*, Bit 25 (1985), 335-357
- [33] Robinson, J.T.: *The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes*, Proc. International Conference on Management of Data, ACM SIGMOD New York. 1981, 10-18
- [34] Rotem, D.: *Spatial Join Indices*, Proc. of the 4th IEEE International Conference on Data Engineering, 500-509 IEEE 1991
- [35] Ruchte, W.D.: *Linearizing Hashing with Priority Splitting: A Method for Improving the Retrieval Performance of Linear Hashing*, Proc. International Conference on Data Engineering (1987), 2-9
- [36] Scheuermann, P., Ouksel, M.: *Multidimensional B-trees for Associative Searching in Database Systems*, Information Systems Vol. 7, No. 2(1982) 123-137
- [37] Six, H. W. and Widmayer, P.: *Spatial Searching in Geometric Database*, Proc. of IEEE 4th International Conference on Data Engineering, (1988), 496- 503
- [38] Tamminen, M.: *Order Preserving Extendible Hashing and Bucket Tries*, BIT 21 (1981), 419-435
- [39] Tamminen, M.: *The Extendible Cell Method for Closest Point Problems*, BIT 22 (1982), 27-41

END

14-12-93

FIN