

H24/3170

ATA BASE

MONASH UNIVERSITY  
THESIS ACCEPTED IN SATISFACTION OF THE  
REQUIREMENTS FOR THE DEGREE OF  
DOCTOR OF PHILOSOPHY

ON..... 3 May 2002 .....

Sec. Research Graduate School Committee

Under the copyright Act 1968, this thesis must be used only under the normal conditions of scholarly fair dealing for the purposes of research, criticism or review. In particular no results or conclusions should be extracted from it, nor should it be copied or closely paraphrased in whole or in part without the written consent of the author. Proper written acknowledgement should be made for any assistance obtained from this thesis.

## ERRATA

- P2 Last sentence: "accesses" for "access"
- P8 para 1, line 11: "\$100,000" for "\$100000"
- P10 para 1, line 3: "don't" for "doesn't"
- P12 para 1, line 1: "in one way" for "in a one way"
- P15 para 1, line 4: "underlying" for "underline"
- P16 last para, line 7: "Comer[20] and Knott[69]" for "[20,69]"
- P17 point (2): "method" for "methods"
- P21 para 2, line 2: "are" for "is"
- P21 para 2, line 3: "known" for "know"
- P26 para 2, line 10: " $2^{d-1} - 1$ " for " $2^{d-1}$ "
- P39 para 1, line 3: "a way" for "away"
- P40 para 1, line 2: "Section 2.6" for "section 2.6"
- P42 para 3, line 5: "function" for "functions"
- P44 para 1, line 1: "include" for "includes"
- P48 para 4, line 2: "discuss" for "discusses"
- P50 para 1, line 5: "of" for "of of"
- P50 para 2, line 1: "Section 2.1" for "2.1"
- P53 para 1, line 2: "2-dimensional" for "an 2-dimensional"
- P53 para 1, line 3: "levels" for "level"
- P53 para 1, line 11: "partitions" for "partition"
- P53 para 4, line 2: "Figure 3.3" for "3.3"
- P62 para 1, line 3: "occur" for "occurs"
- P62 para 2, line 2: "by Freeston" for "Freeston"
- P89 para 1, line 1: " $b_{i,j,1}$ " for " $b_{i,j,0}$ "
- P89 para 1, line 2: " $b_{i,j,k} - 1$ " for " $b_{i,j,k}$ "
- P91 para 3, line 9: "and so" for "and an so"
- P115 para 3, line 2: "Section 5.3" for "section 5.3"
- P115 para 3, line 4: "Section" for "section"
- P116 para 3, line 1: "satisfy" for "satisfies"
- P120 Last para, line 1: "let  $l_h$  be the" for "let  $l_h$  the"
- P121 para 2, line 10: "Equation 5.7" for "Equation 5.7 is"
- P130 para 1, line 4: "smaller page sizes" for "smaller pages"
- P142 para 2, line 1: "compute" for "computes"
- P152 para 3, line 3: "Smaller buffers" for "Smaller buffer"
- P170 para 1, line 5: "the cost of other" for "the cost the other"
- P185 last para, line 2: "Student ID containing" for "Student ID. containing"
- P189 para 1, line 3: "have increased" for "have increased"
- P189 para 1, line 8: "these devices" for "this devices"
- P190 para 1, line 2: "this thesis" for "these thesis"
- P190 para 3, line 6: "don't" for "doesn't"
- P192 para 2, line 1: "Many" for "A lot"
- P192 para 2, line 3: "many" for "a lot"
- P192 para 4, line 1: "we showed how" for "we showed that how"

**Optimal  
Multidimensional Storage  
Organisation  
for Efficient Query Processing  
in Databases**

*Salahadin Mohammed*

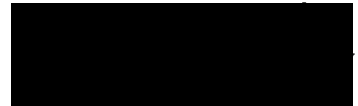
**School of Computer Science and Software Engineering  
Monash University  
Australia**

**Thesis submitted in fulfillment of the requirement  
for the degree of Doctor of Philosophy**

**September 2001**

## DECLARATION

This thesis contains no material which has been accepted for the award of any other degree in any other university. To the best of my knowledge, this thesis contains no material previously published or written by another person, except when due reference is made in the text of the thesis.

A solid black rectangular box used to redact the signature of the author.

Salahadin Mohammed

September 15, 2001



## ACKNOWLEDGEMENTS

I am grateful to my supervisors, Prof. Bala Srinivasan, Prof. Rao Kotagiri and Dr Evan Harris for their able guidance, valuable suggestions, and support in all respects throughout my degree.

I extend my sincere thanks to all the staff and postgraduate students in both Monash and Melbourne Universities for their timely help, patience and coordination. Particularly, Dr Pei Li Joe Zhou, Dr Maria Indrawan, Dr Campbell Wilson, and Dr Phu Dung Le for their understanding, friendship, and patience for the long period of time we shared the same office. I also direct my thanks to Robert Redpath, Duke Fonias, See, Dr Rosanne Price, Arie, Alamin, Mariam, and Khadija for their friendly support.

I dedicate this thesis to my parents, my brothers and sisters, my daughter, and my dearest wife Samira for their love and support throughout my studies.

## ABSTRACT

In a database management system, the performance of query processing is significantly affected by the way the underlying data is organised and accessed. The organisation of uni-dimensional data has been studied extensively, but little work has been done in optimising the organisation of multi-dimension data. The lack of order that preserves spatial proximity of records in uni-dimensional access methods makes them much easier to design than multidimensional access methods. This is because there is no total ordering of objects in two or higher dimensional space that completely preserves spatial proximity. Optimally organising multidimensional data is NP-hard. One way to circumvent the problem is to find heuristic solutions, that is, to look for total orders that preserve spatial proximity at least to a great extent. The goal of all heuristic solutions is that objects located close to each other in the original space should likely be stored close together on the disk. This could contribute substantially in minimizing the number of disk accesses per query.

The little work that has been done in optimising multidimensional data was limited to uniform data distribution and rarely considered the probability of use of each query. And those who did consider the probability of use of each query, they were limited to either partial match query or range query. This is the first ever work which shows that by combining heuristics and combinatorial algorithms, near-optimal solutions can be found which organise multidimensional data (uniform or skewed) on which all the relational queries are efficiently performed. The heuristic algorithms reduce the problem to a

manageable size and the combinatorial algorithms determine near-optimal solutions.

The proposed optimal multidimensional storage organisation is done by selecting a number of bits from each significant attribute of a relation, and then arranging these bits in an optimal way to form a vector (called the *choice vector*). The choice vector is then used to store and retrieve the records of the relation.

We also propose algorithms which use choice vectors for range queries, join queries and other relational queries, and describe the cost of these algorithms. Then, using these algorithms we compare the performance of the proposed storage design with the existing multidimensional storage arrangements.

Combining algorithms which utilise memory efficiently and those which organise storage optimally is expensive. However, we show that when the queries are processed using our algorithms, which use choice vectors and determine good memory utilisation, the cost of the average query can be dramatically reduced.

The experimental results of the proposed algorithms show that performance gains of up to 3617% are achieved, when compared with standard schemes. Moreover, the proposed algorithms are not very sensitive to the change in the query distribution. The result show that if the query probabilities change by upto 80% of their original values, the original storage organisations remain near optimal. Hence, frequent reorganisation of the storage arrangement is also not required.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Optimising the performance of a database management system	2
1.2	Objective of this thesis . . . . .	4
1.3	Related previous work . . . . .	7
1.4	Contributions of this thesis . . . . .	8
1.5	Research methodology . . . . .	10
1.6	Thesis layout . . . . .	11
<b>2</b>	<b>Background</b>	<b>14</b>
2.1	Introduction . . . . .	14
2.2	Multidimensional access methods . . . . .	18
2.3	Multidimensional hashing based PAMs . . . . .	21
2.3.1	Partial-match retrieval and PAMs . . . . .	22
2.3.2	Multidimensional order preserving linear hashing with partial expansion . . . . .	25
2.3.3	The Grid file . . . . .	29
2.4	PAMs based on hierarchical access methods . . . . .	31

2.4.1	Multilevel grid file . . . . .	32
2.4.2	The hB-tree . . . . .	34
2.5	Current methods of optimising PAM design . . . . .	37
2.6	Combinatorial optimisation techniques . . . . .	40
2.6.1	The optimisation problem . . . . .	40
2.6.2	Minimal Marginal Increase (MMI) . . . . .	41
2.6.3	Simulated annealing . . . . .	42
2.6.4	Combining MMI and simulated annealing . . . . .	43
2.6.5	Other optimisation solutions . . . . .	43
2.7	Summary . . . . .	45
<b>3</b>	<b>Reorganising multidimensional data</b>	<b>47</b>
3.1	Introduction . . . . .	47
3.2	The BANG file . . . . .	50
3.3	The BANG file structure . . . . .	51
3.4	Data partitions . . . . .	55
3.5	Partition identifier . . . . .	57
3.6	BANG directory . . . . .	59
3.7	Searching, insertion, deletion and merging . . . . .	65
3.8	Binary division . . . . .	66
3.9	Choice vectors . . . . .	67
3.9.1	Cyclic choice vector . . . . .	69
3.9.2	Optimized choice vector . . . . .	70
3.9.3	Choice vector size . . . . .	71

3.10	Effect of choice vectors on the load factor . . . . .	72
3.10.1	Experimental results and analysis . . . . .	73
3.11	Conclusion . . . . .	82
<b>4</b>	<b>Optimising Partial-match Queries</b>	<b>85</b>
4.1	Introduction . . . . .	85
4.2	Partial-match retrieval . . . . .	87
4.3	Partial-match retrieval Algorithm using The BANG file . . . .	88
4.4	Optimising partial-match retrieval . . . . .	90
4.5	Cost functions . . . . .	91
4.6	Performance Evaluation . . . . .	94
4.6.1	Environment . . . . .	95
4.6.2	Results . . . . .	95
4.7	Conclusion . . . . .	112
<b>5</b>	<b>Optimizing Range Query Retrieval</b>	<b>114</b>
5.1	Introduction . . . . .	114
5.2	Range Queries . . . . .	115
5.3	Minimising range query costs . . . . .	118
5.4	Cost functions . . . . .	119
5.5	Experimental Results . . . . .	121
5.5.1	Environment . . . . .	122
5.5.2	Effect of data and query distributions . . . . .	125
5.5.3	Number of attributes . . . . .	127

5.5.4	File size . . . . .	128
5.5.5	Page size, elapsed time and CPU time . . . . .	129
5.5.6	Stability . . . . .	131
5.5.7	Query-space size . . . . .	134
5.6	Conclusion . . . . .	136
<b>6</b>	<b>Join query processing for skewed data distributions</b>	<b>138</b>
6.1	Introduction . . . . .	138
6.2	The proposed join algorithms . . . . .	140
6.2.1	The selection-module . . . . .	141
6.2.2	The matching-module . . . . .	150
6.3	Optimizing join query processing . . . . .	151
6.3.1	Buffer size vs wav. size . . . . .	152
6.3.2	Wave size and choice vector . . . . .	153
6.3.3	Heuristic algorithms and Cost functions . . . . .	155
6.4	Results and analysis . . . . .	157
6.4.1	Environment . . . . .	158
6.4.2	Effect of data and query distributions . . . . .	158
6.4.3	Number of attributes . . . . .	160
6.4.4	File size . . . . .	161
6.4.5	Page size . . . . .	162
6.4.6	Buffer size . . . . .	163
6.4.7	Stability . . . . .	164
6.5	Conclusion . . . . .	168

<b>7</b>	<b>Optimizing other relational operations</b>	<b>170</b>
7.1	Introduction . . . . .	170
7.2	Selection . . . . .	171
7.3	Projection . . . . .	174
7.4	Join . . . . .	174
7.4.1	Nested loop . . . . .	175
7.4.2	Sort-merge . . . . .	177
7.4.3	Hash joins . . . . .	178
7.4.4	The proposed join algorithm . . . . .	180
7.5	Intersection . . . . .	182
7.6	Union . . . . .	183
7.7	Difference . . . . .	184
7.8	Division . . . . .	185
7.9	Duplicate removal and aggregation . . . . .	187
7.10	Conclusion . . . . .	187
<b>8</b>	<b>Conclusions and Future work</b>	<b>189</b>
8.1	Conclusions . . . . .	189
8.2	Future work . . . . .	191



# List of Tables

4.1	Query distributions. . . . .	96
4.2	Average query costs for a uniform data distribution. . . . .	97
4.3	Average query costs for a clustered data distribution. . . . .	97
4.4	Average query costs for a sinusoidal data distribution. . . . .	98
4.5	Average query costs for a linear data distribution. . . . .	98
4.6	Effect of the number of attributes on the average query cost. .	101
5.1	Query distribution $\Theta_1$ . . . . .	123
5.2	Query distribution $\Theta_2$ . . . . .	124
5.3	Query distribution $\Theta_3$ . . . . .	124
5.4	Query distribution $\Theta_4$ . . . . .	125
5.5	Average query cost for a uniform data distribution. . . . .	126
5.6	Average query cost for a clustered data distribution. . . . .	126
5.7	Average query cost for a sinusoidal data distribution. . . . .	126
5.8	Average query cost for a linear data distribution. . . . .	126
5.9	Effect of the number of attributes on the average query cost. .	128
6.1	Average query cost for a uniform data distribution. . . . .	159

6.2	Average query cost for a clustered data distribution. . . . .	159
6.3	Average query cost for a sinusoidal data distribution. . . . .	159
6.4	Average query cost for a linear data distribution. . . . .	160
6.5	Effect of the number of attributes on the average query cost. .	161
6.6	Effect of buffer size on query cost. . . . .	165
7.1	SUBJECTS table . . . . .	185
7.2	STUDENTS-SUBJECTS table . . . . .	185
7.3	Answer . . . . .	186

## List of Figures

2.1	Multi-level grid file . . . . .	35
3.1	A BANG file of 12 records and 5 data pages. . . . .	52
3.2	The structure of a BANG file. . . . .	54
3.3	$P_{0,1}$ encloses $P_{0,2}$ and $P_{0,3}$ , but it directly encloses $P_{0,2}$ and not $P_{0,3}$ . . . . .	55
3.4	Peer-split of $P$ into $P_1$ and $P_2$ . . . . .	56
3.5	Enclosure-split of $P$ into $P_3$ and $P_4$ . . . . .	56
3.6	Assigning partition-numbers to sub-spaces. . . . .	58
3.7	Splitting a directory. . . . .	61
3.8	Splitting a directory partition. . . . .	63
3.9	The four data distributions used in generating the results. . .	75
3.10	Effect of the choice vector on the load-factor. Number of attributes = 4, page size = 1024 bytes, data distribution = uniform. . . . .	76
3.11	Effect of the choice vector on the load-factor. Number of attributes = 4, page size = 1024 bytes, data distribution = clustered. . . . .	76

3.12	Effect of the choice vector on the load-factor. Number of attributes = 4, page size = 1024 bytes, data distribution = sinusoidal. . . . .	77
3.13	Effect of the choice vector on the load-factor. Number of attributes = 4, page size = 1024 bytes, data distribution = linear. . . . .	77
3.14	Effect of the choice vector on the load-factor. Number of attributes = 2, page size = 1024 bytes, data distribution = uniform. . . . .	78
3.15	Effect of the choice vector on the load-factor. Number of attributes = 3, page size = 1024 bytes, data distribution = uniform. . . . .	79
3.16	Effect of the choice vector on the load-factor. Number of attributes = 4, page size = 1024 bytes, data distribution = linear. . . . .	79
3.17	Effect of the choice vector on the load-factor. Number of attributes = 8, page size = 1024 bytes, data distribution = uniform. . . . .	80
3.18	Effect of the choice vector on the load-factor. Number of attributes = 4, page size = 512 bytes, data distribution = uniform. . . . .	81
3.19	Effect of the choice vector on the load factor. Number of attributes = 4, page size = 2048 bytes, data distribution = uniform. . . . .	82
4.1	relation $R_{0,0}$ . . . . .	88
4.2	$A_{0,0}$ is specified as 40. $R_{0,0}$ has more intervals based on $A_{0,0}$ . . . . .	90
4.3	Effect of the page size on performance. . . . .	99

4.4	Effect of page size on CPU time. . . . .	99
4.5	Effect of file size on performance. . . . .	100
4.6	Stability of optimized solution. Query distribution = $\Theta_1$ , data distribution = uniform. . . . .	104
4.7	Stability of optimized solution. Query distribution = $\Theta_2$ , data distribution = uniform. . . . .	104
4.8	Stability of optimized solution. Query distribution = $\Theta_3$ , data distribution = uniform. . . . .	105
4.9	Stability of optimized solution. Query distribution = $\Theta_4$ , data distribution = uniform. . . . .	105
4.10	Stability of optimized solution. Query distribution = $\Theta_1$ , data distribution = clustered. . . . .	106
4.11	Stability of optimized solution. Query distribution = $\Theta_2$ , data distribution = clustered. . . . .	106
4.12	Stability of optimized solution. Query distribution = $\Theta_3$ , data distribution = clustered. . . . .	107
4.13	Stability of optimized solution. Query distribution = $\Theta_4$ , data distribution = clustered. . . . .	107
4.14	Stability of optimized solution. Query distribution = $\Theta_1$ , data distribution = sinusoidal. . . . .	108
4.15	Stability of optimized solution. Query distribution = $\Theta_2$ , data distribution = sinusoidal. . . . .	108
4.16	Stability of optimized solution. Query distribution = $\Theta_3$ , data distribution = sinusoidal. . . . .	109

4.17	Stability of optimized solution. Query distribution = $\Theta_4$ , data distribution = sinusoidal. . . . .	109
4.18	Stability of optimized solution. Query distribution = $\Theta_1$ , data distribution = linear. . . . .	110
4.19	Stability of optimized solution. Query distribution = $\Theta_2$ , data distribution = linear. . . . .	110
4.20	Stability of optimized solution. Query distribution = $\Theta_3$ , data distribution = linear. . . . .	111
4.21	Stability of optimized solution. Query distribution = $\Theta_4$ , data distribution = linear. . . . .	111
5.1	Query-space intersecting four partitions. . . . .	116
5.2	$P_{2,2}$ is a $\Phi$ -partition but $P_{2,0}$ and $F_{2,1}$ are not . . . . .	117
5.3	Query-space intersecting four partitions. . . . .	118
5.4	Effect of file size on relative performance. . . . .	129
5.5	Effect of page size on performance. . . . .	130
5.6	Effect of page size on CPU time. . . . .	131
5.7	Stability of the optimised choice vector using $\Theta_1$ and the uni- form data distribution. . . . .	132
5.8	Stability of the optimised choice vector using $\Theta_1$ and the clus- tered data distribution. . . . .	133
5.9	Stability of the optimised choice vector using $\Theta_1$ and the si- nusoidal data distribution. . . . .	133
5.10	Stability of the optimised choice vector using $\Theta_1$ and the linear data distribution. . . . .	134

5.11	Effect of query-space size on relative performance. . . . .	135
6.1	$p_{0,0}$ and $P_{1,0}$ are join-compatible while $P_{0,0}$ and $P_{1,12}$ are not. .	141
6.2	For each interval of $R_2$ there are two intervals of $R_3$ . . . . .	146
6.3	$P_{4,0}$ spans $I_{4,0}$ and $I_{4,1}$ . . . . .	147
6.4	$W_{5,1}$ is embedded in $W_{5,0}$ . . . . .	148
6.5	$W_{6,1}$ is join-compatible to $W_{7,0}$ and $W_{7,1}$ . . . . .	148
6.6	Waves with number of partitions higher than the buffer size result in higher join query cost. . . . .	154
6.7	Reducing the number of partitions per wave so that they fit in the available buffer reduces cost. . . . .	155
6.8	Effect of file size on relative performance. . . . .	162
6.9	Effect of page size on performance. . . . .	163
6.10	Stability of the optimised choice vector using $\Theta_1$ and the uni- form data distribution. . . . .	166
6.11	Stability of the optimised choice vector using $\Theta_1$ and the clus- tered data distribution. . . . .	166
6.12	Stability of the optimised choice vector using $\Theta_1$ and the si- nusoidal data distribution. . . . .	167
6.13	Stability of the optimised choice vector using $\Theta_1$ and the linear data distribution. . . . .	167

# Chapter 1

## Introduction

Effective and efficient management of a large volume of data is critical in modern and future computer applications, such as business data processing, multimedia applications, computer aided design and manufacturing, library informations retrieval systems, scientific computations, real-time process control and many other systems. As the size and speed of computer systems has increased, so has the amount of data which has to be manipulated. The manipulation of data is done by using a database management system, which is a collection of programs that enable a user to create, query and maintain a *database* (a collection of related data). So the aim of this thesis is to find optimal or near optimal ways of manipulating large volumes of multidimensional data which improve the performance of a database management system.

The main criteria of evaluating the performance of a database management system is the amount of time it takes to respond to users queries. Researchers have been and still are conducting extensive research with the aim of improving the performance of database management systems. The



primary focus of the research has been on secondary storage systems, that is, using disk drivers to store the data. In recent years, the increase in the amount of physical memory in computer systems has lead to research into primary storage systems, in which the data is held in memory. But with large volumes of multimedia data, which needs to be stored on secondary or tertiary storage systems, means that techniques for managing and manipulating data on non primary storage systems are likely to be required for a number of years.

This chapter has 6 sections. Section 1.1 discusses the main research areas of optimizing the performance of a database management system. The research area chosen in this thesis is explained in section 1.2. In section 1.3, the existing approaches that try to solve similar problems of this thesis are introduced. The main contributions of this thesis are presented in section 1.4. Section 1.5 discusses the setup of the experiments done in this thesis and section 1.6 presents a summary of each chapter of this thesis.

## **1.1 Optimising the performance of a database management system**

In the last thirty years extensive research has been conducted to improve the performance of a database management system. Following is a list of areas where most of the research was focused.

- **Minimising I/O.** One of the main factors in calculating the cost of a query is the number of access done to the hard disk when executing

the query. One technique of reducing this factor is indexing [22, 46] and another technique is clustering [51-54, 100].

- **Reducing computation.** The number of tuples to be compared can be reduced if good hashing, clustering or partitioning algorithms are used [3, 83, 84, 116].
- **Hardware support.** Using dedicated database machines such as hardware hashing units, sorters and filters, processing can be speeded up. There are plenty of such machines and a comprehensive survey can be found in [132].
- **Parallel processing.** Many database operations have high degree of inherent parallelism. This can be exploited to perform them in parallel [23].
- **Query optimisation.** Query optimisation is mainly selecting the best way of executing a query. For example, determining optimal nesting of joining in a multiway join or devising optimal strategies for distributed join processing [19, 113, 133, 142].
- **Optimising buffer usage.** By optimal buffer usage we mean reducing the amount of memory needed in executing a query or making the best of the available memory in executing a query. Optimising buffer usage can reduce the number of disk blocks accesses thus reducing query cost [53, 100].

## 1.2 Objective of this thesis

As mentioned in the beginning of this chapter, one of the most important criteria for evaluating the performance of a database management system is how fast the system can access the data queried by users. This criteria depends, to a large extent, on the organisation of the underlying data. In relational database management system, data is first organised as records. A *record* consists of a list of *fields*. Each field, also called *attribute*, has a *domain*, which is a set of values from which a field value can be drawn. A collection of related records form a *file*. A file normally resides in a disk. A disk is divided into blocks. Each disk block has a capacity of a limited number of records. So each file is stored in one or more disk blocks. The number of disk blocks of a file depends on the size of the disk block, the size of each record, the number of records and the way the records are organised in the file. A disk block is also a unit of transfer between a disk and the primary memory. So a request of one record from a disk block causes all the records in the block to be transferred to the primary memory.

A *query* is an expression that describes records to be retrieved from a file or a database. An answer to a query is the retrieval of all the described records. The four main database queries are exact match, partial match, range and join [25].

- In an *exact match query* the record to be retrieved is described by specifying all the fields.
- In a *partial match query* the records to be retrieved are described by specifying a subset of the fields.

- In a *range query* records to be retrieved are described by specifying a range of values to a subset of the fields.
- In a *join query* records are retrieved from more than one file and mainly described using common fields.

Records of each disk block containing at least one required record are transferred to the primary memory. Then the primary memory is searched to retrieve the required records. Since the disk access time is considerably higher than the primary memory retrieval time, the time to respond to a query is mainly measured in terms of the number of disk access done to answer the query. So having two required records each residing in a separate disk block will cost nearly twice as much time as two or more required records residing in a single block.

One way of improving the performance of algorithms manipulating data on secondary storage is to cluster similar data [3,83,84,116]. The rationalization behind this approach is that if one item of data is required to answer a query, a similar item of data is also likely to be required to answer the query. By clustering the data items together, the amount of time taken to locate and retrieve the data will be reduced. If the amount of data is large, the increase in performance can be substantial.

The clustering of data will be of most benefit if it results in the reduction of the time taken to perform operations which are frequently required of the database management system. To achieve the optimal performance, the frequency, type and the cost of each operation must be taken into account when designing a clustering arrangement. If it is not known, statistics can

be kept on the operations performed on existing systems, and can be used to reorganise the data into better clustering arrangement.

Even if the frequency, type and cost of each operation is known, determining the optimal arrangement can be expensive. For example, Moran [94] showed that designing a particular optimal partial-match retrieval system was NP-hard. However, efficient algorithms have been found which can quickly find optimal or near optimal solutions to this problem [3, 83, 84, 90, 91, 116].

Little work has been done in determining an optimal clustering of multidimensional data for queries other than partial-match retrieval. Other clustering techniques have been proposed; however, they rarely consider the probability of an operation being asked to be performed. For example, Faloutsos and Roseman [33] proposed using fractals to cluster multidimensional data in one dimension for storage on disk. They showed that this clustering technique performed better for range queries than a number of older clustering techniques, but they did not consider varying the frequency of queries. The join is very important and expensive operation in a relational database management system [119, 136, 137]. As the join operation is so expensive, any increase in its cost can result in a significant degradation of the performance of the database management system. Also, most of the other relational database operations, such as intersection, union and difference, are very similar in implementation to the join. As a result a large amount of research has been conducted to find methods of efficiently implementing the join. Using the clustering provided by a data structure to increase the performance of the join has been considered in the past, by Ozkarahan and Ouksel [50, 111, 135]. However, none of

the authors attempted to find the optimal clustering organisation.

The primary objective of this thesis is to find optimal multidimensional data organisation which supports the efficient processing of range queries, join queries and other relational operations, and to compare the performance of this arrangement with that of the standard multidimensional data organisation arrangement.

An optimal data organisation is expected to be optimal only for a query distribution that was used to find it and, perhaps, other similar query distributions. This thesis also addresses the issue of how much must the query distribution change before current data organisation arrangement is no longer near optimal. In other words, how sensitive are our techniques of data organisation to changes in the query distribution?

### 1.3 Related previous work

As mentioned above little work has been done on attempting to determine an optimal organisation of multidimensional data for queries other than partial-match retrieval. Other clustering techniques have been proposed; however, they rarely consider the probability of an operation being asked to be performed. The few who did, assumed a uniform data distribution [51,99]. J. Lee, Y. Lee, K. Whang and I. Song [79], attempted to do the same like what is done in this thesis but their research is limited to range queries. Their approach will be explained in more detail in Section 2.5, but the following is a brief summary of their approach.

A range query can be expressed by specifying a region, called *query region*. It is convenient to think of a query region as a cross product of the specified intervals (partial domains) in the query. A *range query* is a conjunction of equality predicates with at least one range predicate. Query processing can be interpreted as an operation of accessing all the pages intersecting the query region, and then retrieving the required records from these pages. Hence, J. Lee et. al. suggested that the cost of a query can be minimised if the number of pages intersecting the query region is minimised. They define an *interval ratio* as the ratio of intervals of the attributes. For example, in the following query: "Find all employees whose ages are between 30 and 50 and whose salaries are between \$50,000 and \$100000", the interval ratio for age and salary becomes 1:2500 when the same integer domain is used. Interval ratios are used to represent the shape of a region. J. Lee et. al. suggested that the number of regions intersecting with a query region at a given arbitrary position is minimised when the interval ratio of each page region in the domain space is the same as that of the query region, regardless of the size of the page region.

## 1.4 Contributions of this thesis

In this thesis, we propose new data organisation techniques which minimise the average cost of a given set of operations whose probability distribution is known. What makes our work different from the other similar works is that, the data distribution can be skewed and attributes can be correlated. Plus our study not only covers partial match query, which is the case with the

other studies, but other relational operations like range query, join, union, difference and others. The main contributions of this thesis are:

- New techniques of optimising storage design for a set of queries with known probability distribution are proposed. Finding optimal storage design is NP-hard. We show that by combining heuristics and combinatorial algorithms, near-optimal storage designs can be achieved which organise multidimensional records on which all the relational queries are efficiently performed. The heuristic algorithms reduce the problem to a manageable size and the combinatorial algorithms determine near-optimal storage designs.
- Nearly all existing storage design algorithms are limited to optimising uniform multidimensional data. In our proposed algorithms there is no such limitation. This makes our work the first ever work which optimises the organisation of multidimensional data when the data distribution is skewed.
- New algorithms for processing exact match, partial match, range, join, union, intersection, difference and division queries are proposed. Combining algorithms which utilise memory efficiently and those which organise storage optimally is expensive. However, we show that when the queries are processed using our algorithms, which use the proposed storage design techniques and determine good memory utilisations, the cost of the average query can be dramatically minimised. Experimental results of the proposed algorithms show that performance gains of up to 3617% are achieved, when compared with standard schemes.



- New and more accurate cost models for all the relational database operations is proposed. Unlike the existing cost models, the proposed cost models doesn't ignore the cost associated with directory pages. These cost models together with the proposed heuristic algorithms are used to find the optimal way of organising multidimensional data.
- Query distribution change overtime. Storage organisation which is optimal for the current query distribution may not be optimal in the future once the query distribution changes. Rearranging storage design in order to optimise it for the current query distribution can be an expensive operation. The proposed storage design algorithms are not that sensitive to minor changes in the query distribution. Using the proposed techniques, query distributions can change up to 80% of their original values, and the original storage organisations remain near optimal. Hence, frequent reorganisation of the storage arrangement is not needed.

## 1.5 Research methodology

To carry out the experiments and to validate the performance of the algorithms in this thesis, a mini relational database management system was implemented using C++. The underlying data was organised by a multidimensional file structure called the Balanced and Nested Grid (BANG) file which was also implemented using C++. The BANG file is explained in Chapter 3. This mini database management system supported, exact match, range, and join queries. A query language was implemented using C++ to

manipulate the data in the mini database management system. All the experiments were conducted on Sun workstations. Each experimental relations were populated by randomly generating one million data records. The distribution of the multidimensional data ranged from uniform to extremely skewed.

## 1.6 Thesis layout

This thesis consists of seven chapters. Chapter 2 provides the background for this thesis. This chapter contains the discussion of the terminology, the data organisations and the tools which are fundamental to the work of this thesis. The chapter starts by the introducing the terms used in this thesis, followed by a brief discussion of data organisations, mainly multidimensional, and their access methods. This is followed by a brief introduction and analysis of the existing approaches which optimise data organisations. The chapter concludes by a discussion of some tools (heuristic algorithms) used in this thesis to optimise data organisations and their access methods.

Chapter 3 discusses techniques of organising data and an access structure known as the BANG file [37]. This is the file structure that is used in this thesis for all the experiments. Hence the chapter explains the structure of the BANG file, what the initial structure of the BANG file looks like, how its structure changes as more records are inserted, what happens as the number of records to be stored in a data block exceeds its limit, how partitions of the file are labeled, how new directory blocks are added, how records are searched, inserted and deleted, and how under populated blocks are merged

in the BANG file. In the original BANG file, data was organised in a one way. In the same chapter we discuss our version of the BANG file which allows multiple ways of organising data. Also, in the same chapter, the effect of different ways of organising data on the load factor and other characteristics of the BANG file are discussed.

In Chapter 4, we present a technique of clustering records in multidimensional structures which minimises the average cost of partial-match query. The chapter starts by discussing partial match queries in general. It then explains a partial-match retrieval algorithm using the BANG file. This is followed by the introduction of new techniques of optimising partial-match queries using heuristic algorithms and some cost functions. The chapter concludes by presenting the analysis and the experimental results of the new techniques.

Chapter 5 discusses new ways of organising data for optimising range queries. The chapter first explains range queries in general, which is followed by the discussion of range query algorithms using the BANG file. Also in the same chapter a new way of minimising range query cost is explained. At the end of the chapter the experimental results and analysis of the proposed algorithms are presented.

In Chapter 6, the optimisation of join queries with known probability distribution is discussed. In this chapter we explain how the join query processing can be optimised by optimising the organisation of data. The experimental results and analysis of the proposed algorithms are also discussed in the same chapter.

In Chapter 7, we combine and generalise the work in the previous chapters. We also discuss the implementation and the comparison of a number of other basic relational operations such as union, division and intersection, required of a database management to answer queries.

Chapter 8 presents the conclusion and possible future work in the area.

# Chapter 2

## Background

### 2.1 Introduction

The main objective of this thesis is to optimise the organisation of multidimensional data (skewedly or uniformly distributed) in order to minimise the cost of a set of queries with known probability distribution. In this chapter we cover the different types of multidimensional data organisations and the methods that are used to optimize them. The chapter starts with the introduction of common terms used in this thesis, followed by a brief discussion of multidimensional data organisations, which is again followed by a brief introduction and analysis of some existing approaches which optimise multidimensional data organisations. The chapter concludes by a discussion of some methods (heuristic algorithms) which are used in this thesis to optimise multidimensional data organisations.

One of the most important criteria for evaluating the performance of a database management system is how fast the system accesses data queried by users. This criteria depends, to a large extent, on the organisation of the underline data. In relational database management system, data is first organised as records. A *record* consists of a list of *fields*. Each field, called *attribute*, has a *domain*, which is a set of values, from which the field value can be drawn. A collection of related records form a *file*. Records of a file are organised in blocks. So a file consists of a number of blocks. In this thesis we refer to the *block* as the basic unit of storage. It is also the basic unit of transfer between disk and memory. It is also expected to be a small multiple of the disk hardware sector size. The block *address* is a number which the index scheme interprets to determine the physical location of a block within the data file.

A *query* is an expression that describes records to be retrieved from a file or a database. An answer to a query is the retrieval of all the described records. The four main database queries are exact match, partial match, range and join [25]. In an *exact match query* the record to be retrieved is described by specifying all the fields. In a *partial match query* the records to be retrieved are described by specifying some and not all of the fields. In a *range query* the records to be retrieved are described by specifying a range of values to at least one of the fields. In a *join query* records are retrieved from more than one file and mainly described using common fields.

The main objective of this thesis is to find efficient ways of organising records in files in order to speed up query processing. An ideal data organisation is an organisation which has:

- Access methods which are:
  - fast
  - efficient for all types of operations
  - able to adopt well to database growth
  - simple with few special cases
  - efficient in handling concurrent transactions
  - easy and minimum impact when integrated to existing systems
  - independent of data order and distribution.
- High storage utilization. The amount of data in each disk block on the average should be high. The index size should be small compared to that of the actual data.

To speed up query processing, each record in a file is placed using the values of one or more of its attributes. The attributes that determine the placement of records in a file are called the *organising attributes*. A file whose records are placed using one organising attribute has a uni-dimensional file organisation. File organisations like B-trees [4, 5], linear hashing [78, 81] and extendible hashing [29] are some examples of uni-dimensional file organisation. Although somewhat outdated [20, 69] present a good coverage of uni-dimensional file organisations. A file whose records are placed using

more than one organising attribute has a *multidimensional file organisation*. Some examples of multidimensional file organisation are X-tree [11], filter trees [127], BSP-Tree [39, 40], BV-tree [38], G-tree [76], GBD-Tree [105], and hB-tree [28]. Methods used to access multidimensional files are called *multidimensional access methods*. A good coverage of multidimensional access methods can be found in [41, 87].

With an increasing number of applications such as computer aided design [66] and VLSI [130], robotics, geometric or geographic systems, medical imaging [48], environmental protection, data warehouse [26], visual perception and text retrieval systems, searching using several attributes is common than using one attribute. With such applications, it is better to use multi-attribute indexing instead of several single attribute indexes for the following two reasons:

1. The number of disk blocks to be accessed can be minimised, because one index instead of multiple single attribute indexes has to be searched.
2. When new records are inserted, deleted or updated multiple updates are required for multiple single indexes, but a single update of index is needed for a multidimensional access methods.

The databases of the above mentioned applications, tend to be notoriously large, and are growing fast [18]. Despite growing primary memories, it is often impossible to hold the entire database in main memory.

The time taken to answer a query (query cost) is mainly measured by the number of disk accesses performed to retrieve the records described by the query. If the described records of a query are scattered in many blocks, the



cost of the query will be high. But if the described records are clustered in smaller number of blocks the cost of the query will be lower. Minimizing the cost of single attribute access methods has been extensively studied, hence our techniques will look at multidimensional access methods. Few researchers studied techniques of minimizing query costs when using multidimensional file structures, but nearly all of them were limited to uniform data distributions [52-54, 60, 77, 79, 133]. Some of the existing techniques will be discussed in subsequent sections. To our knowledge this is the first study which introduces techniques of optimally clustering records in a multidimensional file structure when the data distribution is skewed.

This chapter has 7 sections. Section 2.2 gives introduction to multidimensional access methods. Sections 2.3 and 2.4 explain point access methods which are based on hashing and those which are based on hierarchical or tree like access structures respectively. Existing techniques of optimizing physical database design for efficient query processing are explained in Section 2.5. Section 2.6 discusses some of the heuristic and combinatorial algorithms used in the optimisation of physical database design. Section 2.7 summarizes this chapter.

## 2.2 Multidimensional access methods

As was explained in the last section, a file whose records are placed using more than one organising attribute has a *multidimensional file organisation*. The methods used to access multidimensional files are called *multidimensional access methods*.

Multidimensional access methods are classified into two types: *point access methods* (PAM) and *spatial access methods* (SAM). Point access methods are primarily been designed to perform searches in databases that store only points. Points correspond to records (entities) that doesn't have spatial extension. Examples of such access methods are interpolation based grid files [108,110], twin grid file [58], hB-tree [85], k-d-b-tree [119] and Buddy-tree [125]. Spatial access methods however, can manage extended objects, such as lines, polygons and higher dimensional polyhydra. Examples of spatial access methods are DO [32], the different flavor of R-trees [8, 49, 64, 65, 121, 124, 126], cell tree [47], LSD-tree [55] and SKD-tree [101].

A multidimensional file with  $n$  organising attributes can be envisioned as a  $n$  dimensional domain space. We define a *domain space* as the Cartesian product of the domains of all the organising attributes. The domain space is partitioned into a number of regions. Each region corresponds to a disk block. In PAM, each record is represented as a point within a region and is stored in the disk block corresponding to the region.

In the current PAMs, each region is accessed using a multidimensional hash function or hierarchical access methods (search trees) or both. PAMs which use multidimensional hash functions include the GRID file [97], EXCELL [134], the Two-Level Grid File [56], Multidimensional Linear Hashing and its variants like multidimensional order-preserving linear hashing with partial expansions (MOLHPE) [57, 71] and PLOP-hashing [75]. PAMs which use hierarchical access methods, unlike hash based methods, perform no address computation. Example of such PAMs include Balanced Multidimensional Extendible Hash Tree [106, 107], k-d-B-Tree [119] and the LSD-

Tree [55]. Some PAMs have tree structured directory and also employ dynamic hashing scheme. Some examples of such PAMs are Interpolation Based Grid File [110], The BANG file [36], the buddy tree and the hB-tree and its variants [27, 28, 85, 86].

The lack of order that preserves spatial proximity of records in unidimensional access methods makes them much easier to design than multidimensional access methods [42, 104]. There is no total ordering of objects in two or higher dimensional space that completely preserves spatial proximity. One way to circumvent the problem is to find heuristic solutions, that is, to look for total orders that preserve spatial proximity at least to a great extent. The goal of all heuristic solutions is that objects located close to each other in the original space should likely be stored close together on the disk. This could contribute substantially in minimizing the number of disk accesses per range query. One thing that all proposed methods have in common is that they first partition the domain-space into regions. Each of the regions is labeled with a unique number that defines its position in the total order. The records (points in the domain-space) are then sorted and indexed according to their region. The way the regions are labeled determines how clustered adjacent regions are stored in the secondary memory. Some common labelling techniques used are: *row-wise enumeration of regions* [122], *Peano curve* [95], *quad codes* [34], *N-trees* [141], *location codes* [2], *z-ordering* [104] (used by Oracle in 1995), *Hilbert curves* [33], and *Gray ordering* [30, 31].

To further elaborate the design of PAMs, four PAMs are discussed in more detail in the subsequent subsections. The BANG file will be discussed in more details in the next chapter, because it is implemented and used

in the experiments of this thesis. Though the experiments in this thesis were carried out using the BANG file, they are equally applicable with other multidimensional access methods.

## 2.3 Multidimensional hashing based PAMs

In this section, example of PAMs which use multidimensional hash functions are discussed. First the common features of these PAMs is discussed and then some of the well know PAMs which use multidimensional hash functions are discussed. The use of multidimensional hashing in partial-match queries is also mentioned.

In hashing schemes, the address of a disk block where a record resides is determined by a hash key calculated for that record. If the file on which the record resides has one organising attribute, a hash function is applied to the value of that attribute. But if the file has many organising attributes, then as many hash functions are used. Each organising attribute has a hash function which maps a value into bit strings. For example, in a relation,  $R_i$ , with organising attributes  $A_{i,0}, A_{i,1}, \dots, A_{i,n-1}$ ,  $n$  hash functions,  $h_{i,0}, h_{i,1}, \dots, h_{i,n-1}$ , are employed.  $h_{i,j}$ , maps each  $A_{i,j}$  value to a bit string,  $b_{i,j,0}b_{i,j,1} \dots b_{i,j,c_{i,j}-1}$ , where  $c_{i,j}$  is the minimum number of bits needed to represent any values of  $A_{i,j}$ . For example, if  $A_{i,j}$  represents a gender of an employee in a company, then the minimum number of bits needed to represent any value of  $A_{i,j}$  is 1. Hence,  $c_{i,j} = 1$ . The hash key for a record is constructed by taking  $d_{i,j}$  bits, where  $0 \leq d_{i,j} < c_{i,j}$ , from the bit string of  $h_{i,j}$  and combining them in a specific order. This order is maintained by a structure known as a *choice*

*vector*. In short a choice vector specifies the order by which the hashed bit strings are combined to form a hash key of a record. Each element of a choice vector is a bit position and is denoted as  $b_{i,j,k}$ , where  $0 \leq k < c_{i,j}$ . For example,  $b_{i,1,0}b_{i,3,0}b_{i,3,1}b_{i,0,0}b_{i,3,2}b_{i,1,1}b_{i,2,0}$  is a valid choice vector for  $R_i$  with four attributes,  $A_{i,0}$ ,  $A_{i,1}$ ,  $A_{i,2}$  and  $A_{i,3}$ .

Let the number of elements in the choice vector be denoted as  $d_i$ , where  $d_i = \sum_{j=0}^{n-1} d_{i,j}$ . Since each element of a choice vector is assigned a value of a 0 or a 1, the maximum number of blocks in the file of  $R_i$  is  $2^{d_i}$ .

### 2.3.1 Partial-match retrieval and PAMs

As was explained in the beginning of this chapter a query is an expression that describes records to be retrieved from a file or a database. An answer to a query is the retrieval of all the described records. In a *partial match query* the records to be retrieved are described by specifying a subset of the fields.

To answer a partial-match query, when using PAMs which use multidimensional hash functions, a hash key is constructed from the query, using the same hash functions which are used to store the data. The hash functions for each attribute value specified by the query is applied to the value of the attribute, forming a bit string for the attribute. The hash key is formed using the choice vector and the bit strings for the attribute. The bits in the choice vector of attributes which were not specified in the query are not set in the hash key. All the blocks in the data file which match the hash key are retrieved and searched for matching records. If a bit is not set in the hash key, blocks with either value for that bit in their address must be retrieved.

We use a "\*" to mark the place of the each bit in the hash key which is not set.

For example, consider the following choice vector of an arbitrary relation  $R_i$  which has four attributes:  $b_{i,0,0}b_{i,1,0}b_{i,3,0}b_{i,2,0}b_{i,1,1}b_{i,1,2}b_{i,3,1}$ . Assume a query that specifies values for  $A_{i,0}$ ,  $A_{i,2}$  and  $A_{i,3}$  but not by  $A_{i,1}$ . Also, assume that the values of the attributes specified in the query results in the following bit strings:

$$h_{i,0} = 1011010010100$$

$$h_{i,2} = 0110101010110$$

$$h_{i,3} = 1010110001101$$

Combining the corresponding bit strings, as specified by the choice vector,  $b_{i,0,0}b_{i,1,0}b_{i,3,0}b_{i,2,0}b_{i,1,1}b_{i,1,2}b_{i,3,1}$ , results in the hash key  $1*10**0$ . Since  $A_{i,1}$  is not specified in the query, the three bits which correspond to  $A_{i,1}$  in the hash key,  $1*10**0$ , are assigned "\*". The 3 unknown bits in the hash key will retrieve  $2^3$ , 8, blocks. The resulting eight blocks retrieved to answer the query are:

```

1010000
1010010
1010100
1010110
1110000
1110010
1110100
1110110

```

The order of the bits in the choice vector can have an impact on the performance of the retrieval algorithm. For example, two consecutive disk blocks can be retrieved faster than two non-consecutive disk blocks because no seek is required to locate the second block once the first has been read. As mentioned in section 2.3.1, the way the regions are labeled determines how clustered adjacent regions are stored in the secondary memory. Some common labelling techniques used are: row-wise enumeration of regions, Peano curve, quad codes, N-trees and Gray ordering. Faloutsos in [30] suggested using Gray codes to map the hash keys to disk block addresses which only differ in precisely one of the last two bit positions will be located in consecutive blocks. This results in better retrieval performance.

Without additional information to aid in determining what the composition of the choice vector should be, the choice vector for a relation usually consists of an equal number of bits from each attribute arranged in a cyclic fashion. Such choice vectors are known as *cyclic*. By using additional information, such as the probability of each attribute being accessed and the cost of access, better choice vectors can be built. We will call such choice vectors as *optimised*. Our aim in the subsequent chapters is to find optimised choice vectors.

Aho and Ullman in [3] described how to determine the optimal number of bits to take from the bit string of each attribute to make up the choice vector for partial match retrieval. Their method assumes that the probabilities of each attribute appearing in a query is specified, and that the probabilities are independent of each other. Moran in [94] showed that for the general problem when the probability of an attribute appearing in a query is not

independent of the other attributes, finding the optimal bit allocation is NP-hard. Lloyd in [83, 84] presented an efficient heuristic algorithm for finding a good solution to this general problem.

### 2.3.2 Multidimensional order preserving linear hashing with partial expansion

Multidimensional order preserving linear hashing with partial expansion, MOLHPE, is a dynamic hashing scheme introduced by Kriegel and Seeger in [71]. MOLHPE used very small directory or used no directory at all. The hash key for a record, whose construction is determined by the choice vector, specifies the address of the block in the data file in which the record is stored. MOLHPE combines *standard linear hashing* [57, 81] *linear hashing with partial expansion* [78], and *order preserving linear hashing* [57, 109] schemes. So to understand MOLHPE, let us first discuss briefly, the standard linear hashing, linear hashing with partial expansion, and order preserving linear hashing.

#### Linear hashing

In Section 2.3 it was mentioned that the size of a hash file which belongs to  $R_i$  is  $2^{d_i}$  blocks, where  $d_i$  is the size of the choice vector of  $R_i$ . If this is maintained then it means that the size of a hash file is increased by doubling its current size, from  $2^{d_i}$  to  $2^{d_i+1}$ . For example, if the current size of  $R_i$  is  $2^{d_i}$  blocks, then the next size of  $R_i$  is  $2^{d_i+1}$  blocks. Doubling a file size in one step is a waste of disk space and a very expensive operation because all



existing records have to be reorganised. To avoid doubling the hash file in one step, researchers came with different methods of expanding a hash file. For example, Letwin in [81] proposed a hash file called *linear hash file* which expands by one block at a time. By choosing appropriate functions and by expanding the file size by one block at a time, only records in one block are rearranged.

In a linear hash file overflow records are stored in blocks chaining from the primary blocks. A *primary block* is a block which contains no overflow records. A block which contains overflow records is called an *overflow block*. Linear hashing increases the storage space gradually by splitting the primary blocks in an orderly fashion. Consider a file consists of  $2^{d_i}$  primary blocks which are labeled as  $0, 1, \dots, 2^{d_i} - 1$ . When the splitting of block 0 takes place, the file is extended by one block, which is block  $2^{d_i}$ , and approximately half of the records in block 0 will move to block  $2^{d_i}$ . After the split of block 0 is finished, the next block to split is block 1, then block 2 and so on until block  $2^{d_i}$ . A pointer is used to indicate the next block to be split. This pointer starts from block 0. After the split of block  $2^{d_i} - 1$  into two block  $2^{d_i} - 1$  and  $2^{d_i+1} - 1$ , the pointer is reset to block 0 again and the same splitting process is repeated again but this time with twice the original number of primary blocks which is  $2^{d_i+1}$ .

#### Linear hashing with partial expansion

An important factor in hashing techniques is that the best performance is achieved when the records are uniformly distributed among the file blocks. With linear hashing that is not the case. In linear hashing, when blocks with

no overflow blocks is split, the storage utilization of the two resulting blocks is half that of the original block. To improve the distribution of records among blocks of a linear hash file, Larson in [78] proposed expanding the hash file by more than one block in one step. He called his version of linear hash file as *linear hashing with partial expansion*. A full expansion increases the size of a file from  $B \cdot 2^{d_i}$  to  $B \cdot 2^{d_i+1}$  by splitting each of the  $2^{d_i}$  blocks into two, a block at a time, in the manner we have just described. By using partial expansions, the file size increases from  $B \cdot 2^{d_i}$  to  $(B + 1) \cdot 2^{d_i}$ , to ... to  $(2B - 1) \cdot 2^{d_i}$ , to  $B \cdot 2^{d_i+1}$ . While this still results in an even decrease in the storage utilisation, the difference is much smaller than that of the standard linear hash.

During the first partial expansion,  $B$  blocks are split into  $B + 1$  blocks by moving some records from each of the  $B$  block into block  $B + 1$ . Records are not moving between the  $B$  blocks. During the second partial expansion,  $B + 1$  blocks are split into  $B + 2$  blocks, in the same way as in the first partial expansion. This is repeated for each of the  $B$  partial expansions. During the last partial expansion,  $2B - 1$  blocks are split into  $2B$  blocks. The value of  $2^{d_i}$  is then set to  $2^{d_i+1}$ , so that there are  $B$  groups of  $2^{d_i+1}$  blocks, instead of  $2B$  groups of  $2^{d_i}$  blocks, and the process starts again. This was analysed and discussed in more details in [114].

By using linear hashing with partial expansion, the reallocation of records to new blocks is ordered and doesn't occur all at once. Therefore, the cost of increasing the size of the file is low.

### Order preserving linear hashing

Order preserving linear hashing was independently discovered by Burkhard [15], Orenstein [104], and Ouksef and Scheuerman [109]. It is implemented by using an order preserving hash function to generate the hash key for records which are then stored in a linear hash file. Instead of taking the  $d_i$  least significant bits from the hash key to index a file of size  $2^{d_i}$ , as is usually done in linear hashing, the  $d$  most significant bits must be taken. This ensures that the file can expand dynamically while still remaining ordered.

The primary problem with order preserving linear hashing occurs when the data is not uniformly distributed. It results in a large number of overflow blocks for some hash keys, and sparsely filled blocks for others. To overcome this problem, Orenstein proposed *multilevel order preserving linear hashing* [104]. The problem of long overflow chain is reduced by storing the overflow blocks of each hash key in a  $B^+$ -tree instead of in a list. The problem of sparse blocks is reduced by assigning different level (depths) to the blocks stored in order preserving linear hash files, and by eliminating sparsely filled blocks.

MOLHPE combines order preserving linear hashing and linear hashing with spatial expansions. In MOLHPE each dimension is treated equally. The key space of each dimension is mapped into a number between 0 and 1 by an order preserving hash function. As in linear hashing with partial expansion, the size is doubled by a series of partial expansions. Only one dimension is expand at a time. That is, the file size is doubled by splitting in one dimension. MOLHPE outperforms its competitors for uniformly distributed data. However, with nonuniform data distribution it fails because hash functions

don't adapt gracefully to the given distribution. To overcome this problem Kriegel and Seeger [72, 73] attempted to employ stochastic techniques ( $\alpha$ -quantiles) [16] to determine the split point. The idea is to transform the nonuniform data into uniformly distributed values for  $\alpha$ . These values are then used as input to MOLHPE algorithm for retrieval and update. Since the region boundaries are not simple binary intervals, a small directory is needed. They claimed that this method guarantees the performance of MOLHPE to be nearly the same for both uniform distributions and nonuniform data distributions. Unfortunately, this is true only if the distribution of a data in each dimension is independent. Other variant of MOLHPE was introduced by Hutflesz [57] using z-hashing [103] to guarantee that points located close to each other are also stored close together on the disk. But later it was proved to have similar limitations.

Coming back to our main point, to improve the performance of query processing in MOLHPE, optimised choice vector can be used to determine which dimension is to be split at each stage, rather than splitting each dimension in a cyclic manner.

### 2.3.3 The Grid file

The grid file of Nievegelt et al. [97] and some of its variants [7, 117] are typical representatives of a point access method (PAM) based on hashing. The grid file superimposes a  $d$ -dimensional grid on the domain space, thus dividing it into partitions known as cells. The superimposed grid may not be regular, hence the resulting cells may be of different shapes and sizes. One or more cells are associated with one disk block. The association between a disk block

and its cells is maintained by a multidimensional directory. The directory is usually too big to fit in the main memory so it is usually kept on secondary storage. To guarantee that data items are found in two disk access, the list of split points for each dimension (the grid) is kept in the main memory in a  $d$  one-dimensional array called scales. An answer to an exact match query incurs the use of the scales to locate the appropriate directory cell (which will be read from disk), and then the disk block containing the required record.

The original grid file scheme does not specify a splitting or merging policy, and how the grid directory should be implemented. Those are left to the implementor. However, Nievergelt et al. [97] recommended that the splitting policy should be such that a block is always divided into two blocks during splitting. They reasoned that splitting a block into more than two blocks results in a significantly lower average block occupancy. The choice of dimension and location within the dimension to be split are not specified. They noted that one policy is to choose the dimension according to a fixed schedule, such as cyclically. The location of the split could be the midpoint of the interval being split, but it need not be. Nievergelt, compared the buddy and neighbour system for block merging. In the buddy system, a block can only merge with one adjacent, equal-sized buddy in each dimension. In the neighbour system, a block can merge with either of its two adjacent neighbours in each dimension, providing the resulting region is convex. Every buddy is a neighbour, but not every neighbour is a buddy. Two blocks can be merged if the number of records in the two blocks can be contained within a single block. The neighbour system result in a higher storage utilisation because a neighbour is more likely to be available for merging than a buddy.

Depending on the implementation of the grid directory, merging may require a complete directory scan [56, 74]. The grid directory may be implemented in many ways, from lists of lists to a multidimensional array. Nievergelt [97] favored the multidimensional array for space efficiency. In this implementation, each time a dimension is split, the directory size doubles because the space covered by each directory entry is divided in two. However, the number of data blocks is only increased by one. The reference of many directory entries to the same data block illustrates a well known problem on the grid file, which is a super-linear growth of the directory even for a uniformly distributed data [117]. Theoretical analysis of various grid file can be found in [6, 117].

While others have discussed the grid file, it has generally been assumed that the directory is stored as multidimensional array, that dimensions are always split into two at the midpoint of the range, and dimensions are typically chosen cyclically [117]. Optimised choice vectors can be used to determine which dimension is to be split thus optimizing the query processing.

## 2.4 PAMs based on hierarchical access methods

In this section we discuss some PAMs which use hierarchical directory structures, mainly trees, and perform little or no address computation. Like hashing methods, however, they organise the data points in a number of buckets. Each bucket usually corresponds to a leaf node (also called data node) of the tree and the disk block, which contain those points located in the cor-

responding bucket region. The interior nodes of the tree (also called index nodes) are used to guide the search; each of them typically corresponds to a larger subspace of the universe that contains all bucket regions in the subtree below. A search operation is then performed by a top-down tree traversal.

In the rest of this section we will discuss example of PAMs which use hierarchical access methods like Extendible hashing [29], multi-level grid file [140], The hB-Tree [28] and The BANG file [37]. The BANG file is used extensively in the experiments of this thesis, hence it will be discussed in more detail in the next chapter.

### 2.4.1 Multilevel grid file

The multilevel grid file was designed by Whang and Krishnamurthy [139, 140] to overcome the problem of the multidimensional grid file directory size. It achieves this by making the directory a multilevel balanced tree structure and by redefining the way a grid entry is computed. A directory entry in the lowest level of the tree refers to a data page and represents the region allocated to the data page. A data page contains only those records that belong to the region referred to by directory entry. An entry in a higher level directory refers to a directory page of the next lower level directory and represents the region allocated to it. Figure 2.1 by Whang and Krishnamurthy [140], shows a partitioned data space in which the dashed boxes represent data blocks. A two level directory for the data space is also mentioned as shown in Figure 2.1.

The number of entries which can be stored in a page is limited. When the number of entries which must be stored in a page exceeds its limit, the page

is split into two. In a grid file splitting a directory is based on attribute value. In multilevel grid file, each dimension (attribute domain) has an associated hash function which returns a bit string. A dimension is then split using the bits of its hash function. The directory entries contain bit string prefixes and their associated pointers Figure 2.1.

In multilevel gridfile, searching for records starts from the root node. All the entries matching the search criteria are identified. Then the search for matching entries descends to the next lower level directory using the identified matching entries. The same process is repeated on the remaining directory levels until all the data pages enclosing the required records are retrieved. For example, consider a partial-match query which does not specify a value for the first attribute, but does for the second attribute, for the data structure shown in Figure 2.1. Assume that the value of the second attribute returned by the hash function has a bit string prefix of 01. Thus, we must search for blocks with prefixes (-,01). We start with the root node of the multilevel grid file directory, and find that the first, third and fourth entries match the query. Therefore, three entries at the second level must be searched. In the first of the three, the third and fourth entries match our query. We must retrieve their associated data blocks, and search them for answers to our query. In the second of the second level directory entries we must examine, the third directory entry, both the second and third entries match our query, so their data blocks must be retrieved and searched. In the third second level directory entry we must examine, the fourth directory entry, the first two entries match our query, so both their data blocks must be retrieved and searched for matching entries. Thus, in total we must retrieve six data blocks



for potentially locating the matching records.

Grid regions with no associated data blocks do not appear in the directory hierarchy. For example, in Figure 2.1, the region with the prefixes (00,1) does not have an associated data blocks. Therefore, it does not appear in the second level of the index. Grid regions appear only once at any directory level. For example, the region with the bit string (01,-) has only one data block. Consequently, it has only one entry in the second directory level, the last entry of the first directory block. These two features ensure that the directory will grow at the same rate as the data, even for non-uniform data distributions. Therefore the multilevel grid file does not have the same worst case performance as the standard grid file, in which the directory size can potentially double each time a new data block is required.

We believe the cost of processing queries in multilevel grid files can be minimised, if optimised choice vectors are used in association with the hash functions. Optimised choice vectors will be explained in chapters 3 to 7.

### 2.4.2 The hB-tree

The hB-tree is a multidimensional file structure which has features of K-D-B-tree [119] and k-d tree [9, 10]. It differs from K-D-B-tree by the following two features:

1. Its index nodes are organised as k-d trees.
2. A split of an hB-tree node may be done using more than one attribute.

To understand the hB-tree, let us first discuss both the K-D-B and the k-d trees.

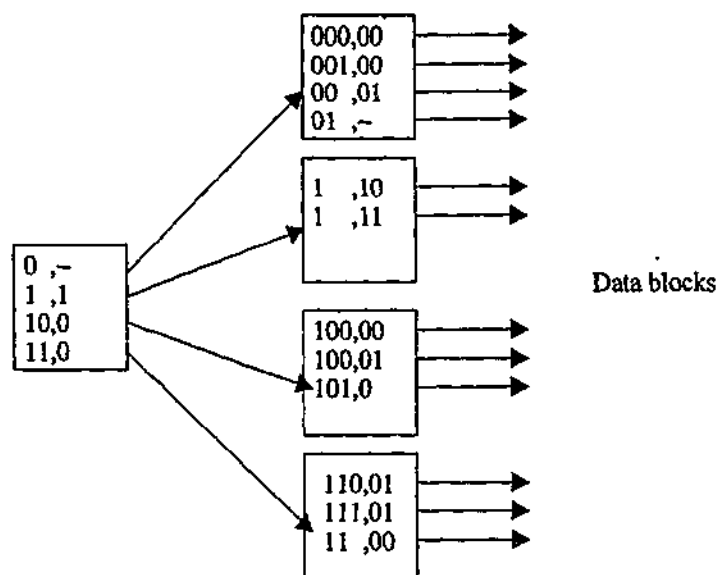
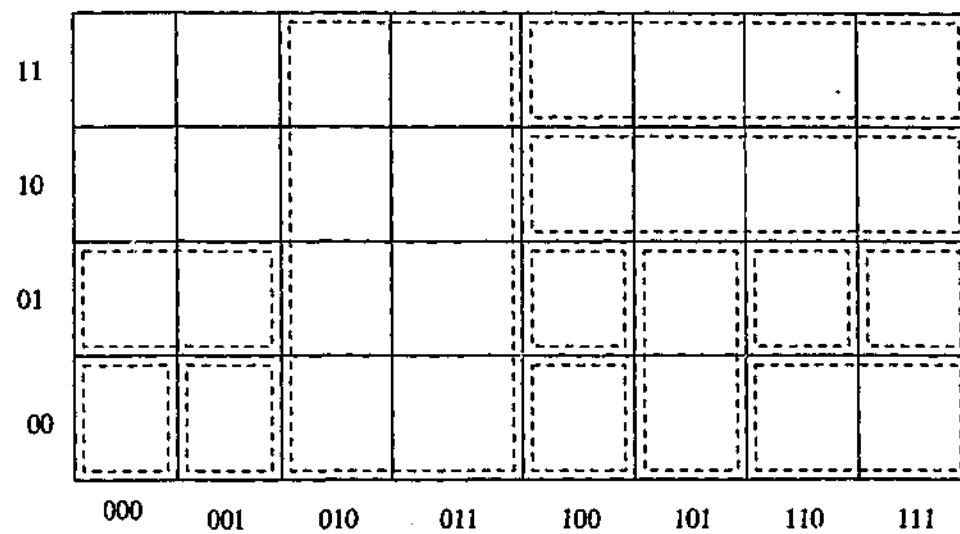


Figure 2.1: Multi-level grid file

K-D-B-tree represents an attempt to generalize B-tree to the multidimensional case. A directory entry of a B-tree contains search values in disjoint intervals of a one dimensional space. A directory entry of a K-D-B-tree represents a region (hypercube) in a  $k$ -dimensional space. Further, like  $B^+$ -tree, data records are stored in leaf nodes, internal nodes contain only index entries which direct search. A K-D-B-tree is a balanced tree, that is, the distance between the root and each leaf node is the same. Regions corresponding to nodes at the same tree level are mutually disjoint; their union is the complete domain space. The leaf nodes store the data points that are located in the corresponding region.

Search queries are answered analogously to the  $k$ -d-tree algorithms. For the insertion of a new data point, first perform a point search to locate the right region. If the region is not full, then the entry is inserted. Otherwise, it is split and about half the entries are shifted to the new data node. If the parent index node does not have enough space left to accommodate the new entries, a new page is allocated and the index node is split by a hyperplane. The entries are distributed among the two pages depending on their position relative to the splitting hyperplane, and the split is propagated up in the tree. The split of the index node may also affect regions at the lower level of the tree, which must be split by this hyperplane as well. Because of this forced split effect, it is not possible to guarantee a minimum storage utilization.

Deletion is done after performing an exact match query on the record to be deleted. If the number of entries drops below a given threshold, the data node may be merged with sibling data node as long as the union remains a  $k$ -dimensional interval. The procedure to find a suitable sibling node to

merge with may involve several nodes. The union of data pages results in the deletion of at least one hyperplane in the parent index node. If an underflow occurs, then the deletion has to be propagated up the tree.

The main difference between the K-D-B-tree and the hB-tree is the way the pages are organised. In K-D-B-tree, regions are disjoint while in hB-tree regions can enclose other regions. Also, in K-D-B-tree the, organization of the directory entries are not clear, but in hB-tree directory is organised as k-d-trees. *K-d-tree* is a multidimensional binary search tree. A search in a standard binary tree is based on one key field. In a k-d-tree this is done using multiple keys,  $K_0, K_1, \dots, K_{n-1}$ . On the first level of the tree, the decision of going to the left child or to the right child is based on the value of  $K_0$ , on the second level the decision is based on the value of  $K_1$  and so on in a cyclic manner.

To optimise query processing in an hB-tree, the search key to be used at each level of the tree can be chosen based on a choice vector.

## 2.5 Current methods of optimising PAM design

Many researches have proposed different multidimensional access methods [61-63, 102, 120, 127], but few attempted to optimally design the access methods for a given set of queries. Even all those who tried, except one [79], were limited to uniform data distribution [52-54, 99]. J. Lee et. al. [79], who attempted to optimise multidimensional access methods for a given set of queries (independent of the data distribution), were limited to range queries.

To our knowledge, for a given set of queries (partial match, range, join or other common relational queries) and their probabilities, this is the first work which attempts to optimally organise multidimensional data, even when the data distribution is skewed. In the rest of this section we will explain the techniques introduced by Lee, et. al. [79].

In multidimensional files, a range query can be expressed by specifying a region, called the *query region*. It is convenient to think of a query region as a cross product of the specified intervals (partial domains) in the query. A *range query* is a conjunction of equality predicates with at least one range predicate.

Query processing can be interpreted as an operation of accessing all the pages intersecting the query region, and then retrieving the required records from these pages. Hence, Lee et. al. [79], suggests that the cost of a query can be minimised if the number of pages intersecting the query region is minimised. They define an *interval ratio* as the ratio of intervals of the attributes. For example, in the following query: *find all employees whose ages are between 30 and 50 and whose salaries are between \$50,000 and \$100000*, the interval ratio for age and salary becomes 1:2500 when the same integer domain is used. Interval ratios are used to represent the shape of a region. It is suggested that, the number of regions intersecting with a query region at a given arbitrary position is minimised when the interval ratio of each page region in the domain space is the same as that of the query region, regardless of the size of the page region. To determine the interval ratio, they use a query pattern given by the user. The query pattern can be obtained by collecting the usage statistics of a database during a

certain time interval [143] or by analyzing the application profiles provided by database administrator [35]. The query pattern is then used for region splitting strategy that partitions the domain space in such away that the interval ratio of a page regions are close to those of query regions.

Lee et. al. [79] didn't discuss how forcing the page region to be equivalent to the query region affects other properties of the multidimensional file, specially storage utilization. Also, there is no discussion on how the other relational operations can benefit from their proposal. Also the query area size over which their proposal is cost effective is not investigated. Query pattern changes with time, and the effect of query changes on the existing design (design which was optimal before query pattern change) is not discussed in the paper.

The approach taken in this thesis is totally different from that taken by Lee et. al. [79]. In this thesis, to find an optimised multidimensional access method the following two items were used.

- Multidimensional data organisations which evenly distribute records among the allocated disk blocks;
- Heuristics and combinatorial optimisation techniques to find optimal data organisations.

Optimising the organisation of multidimensional data in order to minimise the average cost of a given set of queries can be done by using optimal bit allocation. Finding the optimal bit allocation for arbitrary query distribution is NP-hard [94], but the problem can be solved by using heuristics

and combinatorial optimisation techniques which are discussed in the next section, section 2.6.

## 2.6 Combinatorial optimisation techniques

Finding the optimal multidimensional data organisations we are trying to find are NP-hard. To attempt to find good solutions, optimal multidimensional data organisations, we used heuristic and combinatorial optimisation algorithms. These techniques are not guaranteed to find optimal solution to any problem. In the subsequent sections we use the term *minimal* or *optimised* to indicate a solution which is the result produced by one or more of these techniques. These minimal solutions are typically local minima or local optima. For some problems, they are almost optimal. In this section, we introduce the combinatorial optimisation techniques we used to find optimised choice vectors.

### 2.6.1 The optimisation problem

The optimisation solutions that we used can be described in the following way. Consider a set,  $\hat{k}$ , of  $n$  non-negative integers,  $k_i$ , upon which cost function,  $f$ , is defined as [51]

$$C = f(\hat{k})$$

where  $\hat{k} = \{k_0, k_1, \dots, k_{n-1}\}$ . The main objective here is to find  $\hat{k}_{min}$ , such that  $f(\hat{k}_{min}) < f(\hat{k})$ , for all members of  $\hat{k}$ . The constraint

$$\sum_{i=0}^{n-1} k_i = k$$

must be satisfied.

Relating this to a choice vector, section 2.3,  $k$  is the number of elements in the choice vector,  $k_i$  is the number of bits allocated to the  $i$ th attribute,  $\hat{k}$  is a bit allocation, and  $\hat{k}_{min}$  is the optimal bit allocation. As we will find out in the subsequent chapters not only the value of  $k_i$  is important but also the position of these bits within the choice vector.

Finding optimal choice vectors of many of the problems we are trying to solve is NP-hard. To attempt to find good solutions we used heuristic and combinatorial optimisation algorithms. These techniques are not guaranteed to find optimal solution to any problem. Examples of such algorithms are minimal marginal increase and simulated annealing. *Minimal Marginal Increase* (MMI) and *Simulated Annealing* are discussed in Section 2.6.2 and Section 2.6.3 respectively.

## 2.6.2 Minimal Marginal Increase (MMI)

*Minimal Marginal Increase* (MMI) is a greedy heuristic algorithm which we used to come up with optimised choice vectors. MMI works as follows:

Initially nothing is allocated to the elements of the choice vector. Then the first element of the choice vector is allocated to  $b_{i,0,0}$  and the average query cost using the cost functions (which will be discussed in the coming chapters) is computed. The average query cost is repeatedly computed after giving the same choice vector element instead to  $b_{i,1,0}$ , and then to  $b_{i,2,0}$ , and so on, until all the attributes are tried. The attribute which gives the



lowest average query cost is permanently allocated as the first element of the choice vector. The same process is repeated for the second element of the choice vector, then the third, and so on. This process is repeated until all the elements of the choice vector are allocated. The number of choice vector elements will be discussed later in section 3.9.3.

### 2.6.3 Simulated annealing

The second optimisation technique that is used in this thesis to come with the optimal bit allocation is Simulated annealing [1]. Simulated annealing is a class of optimisation algorithms based on Monte Carlo techniques. The algorithm in this thesis is substantially the same as that used in Ramamohanarao et al [115].

The algorithm begins by selecting a random choice vector and computes the cost of the problem on hand using cost functions, which are dependent on the choice vector. Then in each iteration, the algorithm computes new choice vectors and accepts the new choice vector as the basis for further perturbations if it improves the cost or when a *cooling functions* determines that it be accepted. The cooling function is a monotonically decreasing function which specifies the probability of accepting a solution (a choice vector in our case) which does not improve the cost. In the early iteration the probability of accepting a solution that does not improve the cost function is high, but approaches to zero in the later stages. There are a number of parameters which can be used to control the amount of computational resources used by the algorithm. The algorithm terminates when the costs has not improved after pre-specified number of iterations since the last accepted choice vector.

In our implementation, we use a set of random starting allocations (trials) and iterate over each of these, finally selecting the best over the trials.

#### 2.6.4 Combining MMI and simulated annealing

A property that simulated annealing does not share with minimal marginal increase is the dynamic nature of MMI. In simulated annealing, there is no straight forward method to find the optimal bit allocation for  $d+1$  bits, even if the optimal bit allocation of the first  $d$  bits is known. However, MMI may be used in conjunction with simulated annealing to obtain the property of being able to be used for dynamic files. The initial bit allocation can be determined for one file size using simulated annealing. If the size of the file increases, MMI can then be used to determine the attribute to allocate the next bit to. Similarly, if the data file is required to decrease in size, then the technique of maximal marginal decrease (MMD) can be used to provide this ordering. MMD operates in the same way as MMI, except that a single bit is subtracted from each attribute and the cost is recalculated. The aim is still to find the attribute which results in the lowest cost. However, this result of removing a bit from the attribute which results in the largest decrease in cost, instead of allocating a bit to the attribute which results in the smallest increase in the cost.

#### 2.6.5 Other optimisation solutions

Combinatorial optimisation is an active area of research. There are a number of other techniques which could be used in addition to MMI and simulated

annealing to search for optimal bit allocations. These includes iterative improvement, which was used by Swami [133], for join query optimisation, the tabu search [43] and genetic algorithms [88]. Some examples of genetic algorithm that can be used are Genocop (version 2.0) [88] and SGA-C [44, 131]. Additionally, more complex simulated annealing algorithms with sophisticated cooling functions and domain specific knowledge can perform better than more general simulated annealing algorithms [21, 59].

Nurmela in [99] found that simulated annealing typically performed as well or better than iterative improvement and a number of other combinatorial optimisation methods including tabu search, threshold accepting and record to record travel. He also found that simple genetic algorithms which did not use problem specific knowledge did not perform as well as local search algorithms. Implementing a good combinatorial optimisation algorithm for a specific problem is difficult. Each of those algorithms discussed above have a large number of parameters which should be varied, depending on the problem domain. Using domain specific knowledge can also result in a dramatic increase in the performance of the algorithms. We have deliberately used a relatively simple version of the simulated annealing algorithms which use little or no domain specific knowledge. The results generated using this algorithm will show that it is possible to find good solutions to the problem we consider in a reasonable amount of time.

## 2.7 Summary

In relational database systems records are organised as blocks and blocks as files. To speed up query processing, each record in a file is placed by hashing the values of one or more of its attributes. The attributes that determines a placement of records in the file are called the organising attributes. A file whose records are placed using one organising attribute has a uni-dimensional file organisation and a file whose records are placed using more than one attribute has a multidimensional file organisation. When searching using several attributes is more common than using one attribute, it is better to use multi-attribute indexing instead of several single attribute indexes.

The aim of these thesis is to find techniques of organising multidimensional access methods in order to minimise the average cost of a set of queries, whose probability distribution is known. Minimizing the cost of uni-dimensional access methods has been extensively studied, hence our techniques looks at multidimensional access methods. Few researchers studied techniques of minimizing query costs when using multidimensional access methods, but nearly all of them were limited to uniform data distributions. To our knowledge this is the first study which introduces techniques which optimise multidimensional access methods when the data distribution is skewed or uniform.

In this chapter we covered the different types of multidimensional data organisations and the tools which we use in this thesis, to optimise them. Multidimensional access methods are classified into two classes: point access methods (PAM) and spatial access methods (SAM). PAMs are primarily been

designed to perform searches in databases that store only points (records). SAMs however, can manage extended objects, such as lines, polygons and higher dimensional polyhedra.

The time taken to answer a query is mainly measured by the number of disk accesses performed to retrieve the records described by the query. If the described records of a query are scattered in many blocks, then the cost of the query will be high. But if the described records are clustered in smaller number of blocks then the cost of the query will be lower.

In this thesis, to answer a query, we use multidimensional hash functions. A hash key is constructed from the query, using the same hash functions which are used to store the data. The hash functions for each attribute value specified by the query is applied to the value of the attribute, setting the bits of the hash key according to the order specified by the choice vector. The bits in the choice vector which corresponds to the unspecified attributes in the query are not set in the hash key. All the blocks in the data file which match the hash key are retrieved and searched for matching records.

Finding optimal choice vectors of many of the problems we are trying to solve is NP-hard [94]. In this thesis, to attempt to find good solutions we use heuristics and combinatorial optimisation algorithms such as minimal marginal increase and simulated annealing, which are explained in Section 2.6. But before optimising a multidimensional access method using heuristic and combinatorial algorithms, it is important to examine that such reorganisation doesn't change the features of the multidimensional access method, and this is the topic of the next chapter.

## Chapter 3

# Reorganising multidimensional data

### 3.1 Introduction

This chapter discusses new techniques of organising multidimensional data. Although the proposed techniques are applicable to any multidimensional data structure, in this thesis they were applied to a multidimensional file structure known as the Balanced And Nested Grid (BANG) file. The BANG file and why we chose it for our experiments will be discussed in subsequent sections.

As explained in Chapter 1 a query is an expression that describes required records to be retrieved from a file or a database. Blocks that contain at least one required record are transferred to the primary memory. Then the primary memory is searched to retrieve the required records. Since the disk access time is considerably higher than the primary memory retrieval time,

the time to respond to a query is mainly measured in terms of the number of disk access done to answer the query. So having two required records each residing in a separate disk block will cost nearly twice as much time as two or more required records residing in a single disk block.

One way of improving the performance of algorithms manipulating data on secondary storage is to cluster similar data. The rationalization behind this approach is that if one item of data is required to answer a query, a similar item of data is also likely to be required to answer the query. By clustering the related data items together, the amount of time taken to locate and retrieve the data will be reduced. If the amount of data is large, the increase in performance can be substantial.

The clustering of data will be of most benefit if it results in the reduction of the time taken to perform operations which are frequently required of the database management system. To achieve the optimal performance, the frequency, type and the cost of each operation must be taken into account when designing a clustering arrangement. If it is not known, statistics can be kept on the operations performed on existing systems, and can be used to reorganise the data into better clustering arrangement.

Since in all the experiments of this thesis are using the BANG file [37], the main aim of this chapter is to discuss:

1. possible data organisations in the BANG file;
2. new techniques of data organisation in the BANG file;
3. the effect of different types of data organisations on the features of the BANG file, specially its load factor.

This chapter has eleven sections. Section 3.2 briefly explains the structure of the BANG file and why the BANG file was chosen for the experiments in this thesis. The structure of the BANG file and its other main features are covered in the next section. Also, in the next section the two parts of the BANG file, namely, data partitions and directory partitions, are discussed. Section 3.4 concentrates on the data partitions of the BANG file. The section explains how data partitions are created and how they are uniquely identified. Section 3.5 explains how partition identifiers are computed. Directory partitions are discussed in section 3.6. The section also discusses how directory partitions are created and how they are uniquely identified. The way records are searched, deleted and inserted in a BANG file are covered in section 3.7. Section 3.8 explains a technique of partitioning a domain-space. The technique that is explained in this section is known as a *binary division*. It is important to understand the binary division in order to understand how partitions are labeled and how choice vectors are generated. There are three possible types of choice vectors, namely, cyclic, random and optimised. They are explained in detail in section 3.9. Cyclic choice vector, which is the one used by the original BANG file, is discussed in subsection 3.9.1, while random and optimised choice vectors are discussed in subsections 3.9.2 and 3.9.3 respectively. The effect of the choice vectors on the load factor is discussed in section 3.10. This section shows experimental results of how the load factor is affected when a cyclic or a random or an optimised choice vector is used with different data distributions, different number of attributes, different number of records and different number of disk block sizes. The last section is the conclusion of this chapter.



### 3.2 The BANG file

The BANG file is a multidimensional file structure which was first introduced by Micheal Freeston [36, 37]. It has many similarities to the other grid files like multilevel grid file [139, 140] and nested interpolation based grid file [110]. It differs from the other grid files in the way its domain-space is partitioned. The way the BANG file is partitioned together with the rest of its other features will be explained in the subsequent sections.

As mentioned in 2.1, minimizing the cost of single attribute access methods has been studied extensively, hence our techniques is limited to multidimensional access methods, specially Point Access Methods, PAMs. So, for our experiments we required a PAM which supports efficient processing of queries and whose performance does not degrade in the presence of non-uniform data distributions. The BANG file is such a PAM. This is because the BANG file evenly distributes records amongst its partitions even if the distribution of data is extremely skewed [36, 37]. It does so by creating more partitions in the domain subspaces where the density of the records is high and fewer partitions where the density of the data is low. The subsequent sections will explain how the BANG file does this.

Other reasons why we chose the BANG file for study include its:

- multidimensional file structure;
- high load factor ( $\geq 67\%$ );
- fully dynamic in nature;
- no data replication property;

- high fan-out ratio;
- overflow and underflow propagating only upwards the tree;
- maintenance of spatial relationships between objects;
- worst case single object searching, insertion and deletion require no more disk accesses than the height of the tree;
- easy incremental reorganization as the file grows;
- ability to handle range searches, partial match searches and exact match searches.

The techniques of optimising data organisations, which are proposed in this thesis, are not limited to work with the BANG file, they are equally applicable with other multidimensional file structures.

### 3.3 The BANG file structure

The BANG file has a multidimensional file organization. It has a structure of a balanced tree where each leaf node corresponds to a disk block containing data records and each non-leaf node corresponds to a disk block containing entries which contain information about nodes one level down the tree.

You can also envisage a BANG file with  $n$  organizing attributes as an  $n$ -dimensional data space. For example, a relation  $R_i$  which has  $n$  attributes,  $A_{i,0}, A_{i,1}, \dots, A_{i,n-1}$ , organised as a BANG file can be envisioned as an  $n$ -dimensional data space where each dimension corresponds to the domain of an attribute in the relation. The *domain-space* of the relation is the cartesian

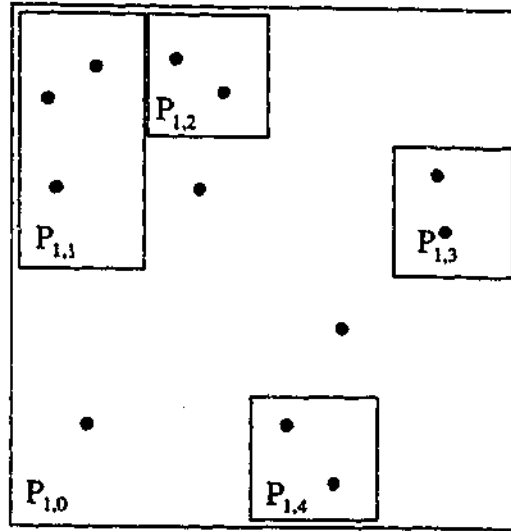


Figure 3.1: A BANG file of 12 records and 5 data pages.

product of  $D_{i,0}, D_{i,1}, \dots, D_{i,n-1}$ , where  $D_{i,j}$  is the domain of  $A_{i,j}$ . A record is represented as a point and a node as an  $n$ -dimensional partition within the domain-space. Figure 3.1 contains an example of a relation which has 2 attributes organised as a BANG file. The relation has 5 partitions (the boxes) and 12 records (the points).

Let  $P_{i,j}$  represent a partition in  $R_i$ , where  $0 \leq j < N$  and where  $N$  is the number of partitions in  $R_i$ .  $P_{i,j}$  is a hyper-rectangle which covers the subspace  $(e_{i,j,0}, e_{i,j,1}, \dots, e_{i,j,n-1})$ , where  $e_{i,j,k}$  is an edge describing the extent of the hyper-rectangle along  $D_{i,k}$ .

As explained in the beginning of this chapter a BANG file has a balanced tree structure. Each node of the tree corresponds to a partition. Partitions which correspond to nodes of the same level of the BANG tree span the domain space. For example, if the height (number of levels) of a BANG tree is 3, partitions at level 1 span the whole domain-space, those at level 2 also span the whole domain-space and so on. Figure 3.2 contains an example of

a BANG file organization. The figure shows the BANG file represented as a tree and as an 2-dimensional domain space. The BANG file in the figure has 3 levels, namely, level 0, 1 and 2. Level 0 is the level of the leaf nodes and level 2 is the level of the root node. In the figure, a node and its corresponding partition have been assigned the same label. The root partition,  $P_{0,0}$ , is at level 2 and it spans the whole domain-space. It has three entries which correspond to partitions which are at level 1, namely,  $P_{0,1}$ ,  $P_{0,2}$ , and  $P_{0,3}$ . Also, the 3 partitions at level 1 span the whole domain-space. Each of the partitions at level 1 contains entries which correspond to partitions at level 0. For example,  $P_{0,1}$  of level 1 contain entries of  $P_{0,4}$ ,  $P_{0,5}$ , and  $P_{0,6}$ . These 3 partition span the subspace spanned by  $P_{0,1}$ . All the partitions at level 0 span the whole domain-space.

Each partition of a relation has a unique identifier. Each identifier consists of two numbers, namely, a partition-number and a partition-level, and is denoted as: **partition-number:partition-level**. The computation and meaning of partition numbers and partition levels will be explained in section 3.5.

In a BANG file, a partition can be *enclosed* by other partitions and it can also enclose other partitions. In Figure 3.3 partition  $P_{0,1}$  encloses partition  $P_{0,2}$  and  $P_{0,3}$ . Also,  $P_{0,2}$  encloses  $P_{0,3}$ .

Partition  $x$  *directly encloses* partition  $y$  if  $x$  is the smallest partition from all the partitions which enclose  $y$ . For example in 3.3,  $P_{0,1}$  directly enclose  $P_{0,2}$  but it doesn't directly enclose  $P_{0,3}$ .  $P_{0,2}$  directly encloses  $P_{0,3}$ .

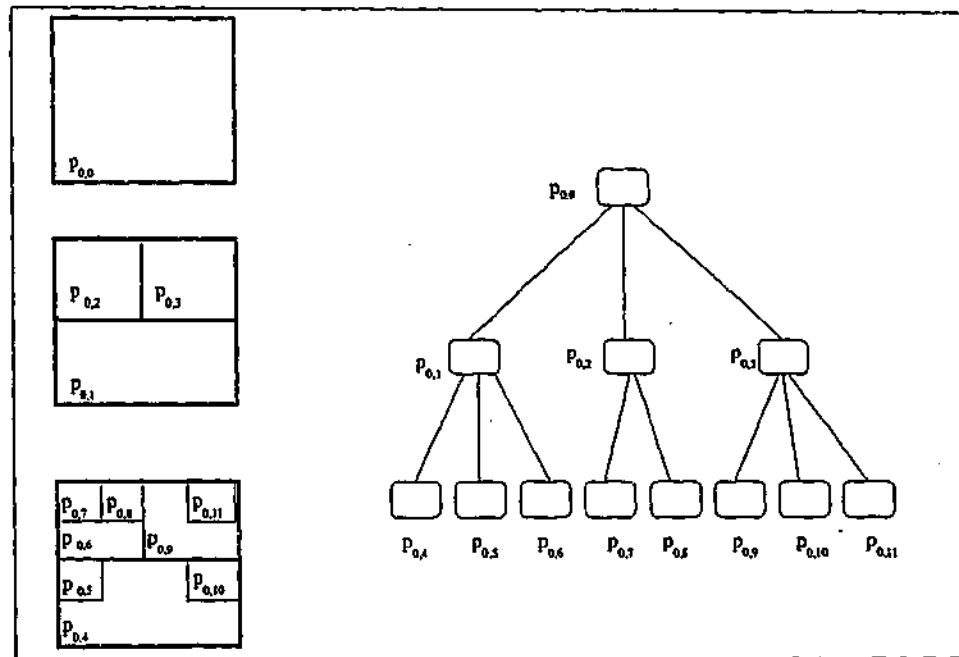


Figure 3.2: The structure of a BANG file.

Following is the formal definition of *encloses* and *directly encloses*.

**Definition 3.1** Partition  $u : v$ , where  $u$  is the partition-number and  $v$  is the partition-level, encloses partition  $x : y$  if  $v \leq y$  and the least significant (rightmost)  $v$  bits of  $x$  are identical to  $u$ .

**Definition 3.2** Partition  $u : v$  directly encloses partition  $x : y$  if there is no other partition  $w : z$  which encloses  $x : y$  and is enclosed by  $u : v$ .

A leaf node of the BANG file contains actual data records. Each entry of a non-leaf node contains a partition identifier and the corresponding disk address of a node which is at the next lower level of the BANG tree. From now on let us call a partition which correspond to a leaf node as a *data-partition* and those which correspond to non-leaf nodes as *directory-partitions*.

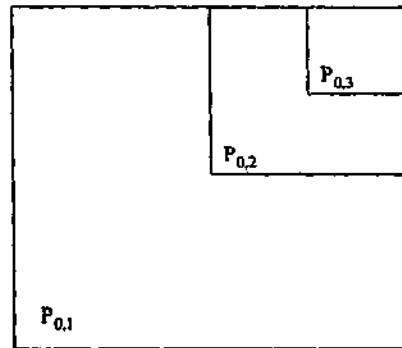


Figure 3.3:  $P_{0,1}$  encloses  $P_{0,2}$  and  $P_{0,3}$ , but it directly encloses  $P_{0,2}$  and not  $P_{0,3}$ .

### 3.4 Data partitions

Initially the BANG file has one leaf node and one non-leaf node (the root). At this point the leaf node has no records. The root has one entry (record) which contains the address of the non-leaf node. When new records are inserted they are stored in the leaf node.

The number of records which can be stored in a leaf node is limited. As the number of records to be stored in a leaf node exceeds its limit, it is split into two new leaf nodes. Its records are then divided between the two new nodes and it ceases to exist. The division of a data-partition into two is a recursive process. First the partition is divided into two equal sized partitions. If the contents of the two new partitions is balanced the division process ends. By *balanced* we mean that the number of records in both new partitions are equal or very close to equal. If the contents of the two new partitions is not balanced, the size of the partition with the higher number of records is successively halved until a balance is achieved. If the split results in two equal sized partitions it is called a *peer-split*, and the two resulting partitions

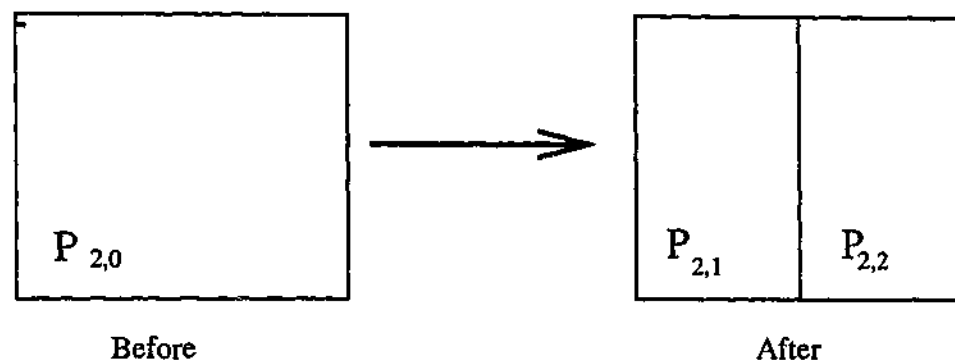


Figure 3.4: Peer-split of  $P$  into  $P_1$  and  $P_2$ .

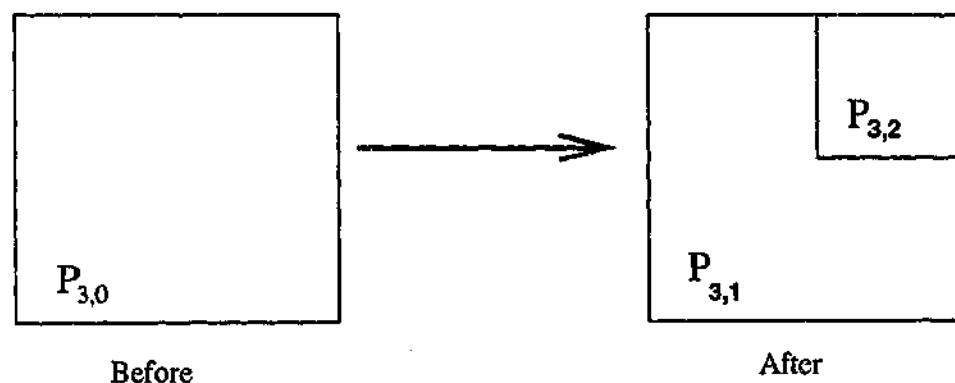


Figure 3.5: Enclosure-split of  $P$  into  $P_3$  and  $P_4$ .

are called *peers*. A non peer-split is called an *enclosure-split* because one of the partitions encloses the other. Figure 3.4 contains an example of peer-split and Figure 3.5 contains an example of an enclosure-split.

In the BANG file, as in many other multidimensional files, a peer-split of a partition is done using a technique known as a binary division. A value, which is a mid point of one of its edges is used to bisect the current partition. We call this value *split-value* and its corresponding attribute the *split-attribute*. Records with value less than the split-value are put in one of the peers and the remainder are put in the other peer. Let us call the former peer as the *low-peer* and the latter as the *high-peer*.

Each time a partition is created, a new record which contains its identifier and its address is inserted into a directory-partition. The number of such records which can be stored in a directory-partition is also limited. As the number of records which must be stored in a directory-partition is above its limit more directory-partitions are created by dividing existing directory-partitions. The process of dividing a directory-partition is explained in section 3.6.

### 3.5 Partition identifier

A partition identifier uniquely identifies a partition in a relation. It consists of two numbers, a partition-number and a partition-level, and is denoted as: **partition-number:partition-level**.

**Partition-level:** denotes the size of a partition relative to the size of the domain-space. As explained in section 3.4, a new partition is created by successively halving an existing partition. Therefore, the size of a partition is  $2^{-\mu}$  that of the domain space, where  $\mu$  is 0, 1, 2, ... The partition-level of a partition is its corresponding  $\mu$  value. For example, in Figure 3.1 the size of the domain-space is equal to  $2^0$ ,  $2^3$ ,  $2^4$ ,  $2^4$ , and  $2^4$  that of  $P_{1,0}$ ,  $P_{1,1}$ ,  $P_{1,2}$ ,  $P_{1,3}$  and  $P_{1,4}$  respectively. Hence the partition-levels of  $P_{1,0}$ ,  $P_{1,1}$ ,  $P_{1,2}$ ,  $P_{1,3}$  and  $P_{1,4}$  are 0, 3, 4, 4 and 4 respectively.

**Partition-number:** denotes the logical address of a partition in a domain-space. The method of assigning partition-numbers to partitions can be easily explained with the help of the binary tree of Figure 3.6. Each node of the binary tree consists of a partition-number, a pointer to a left child and a



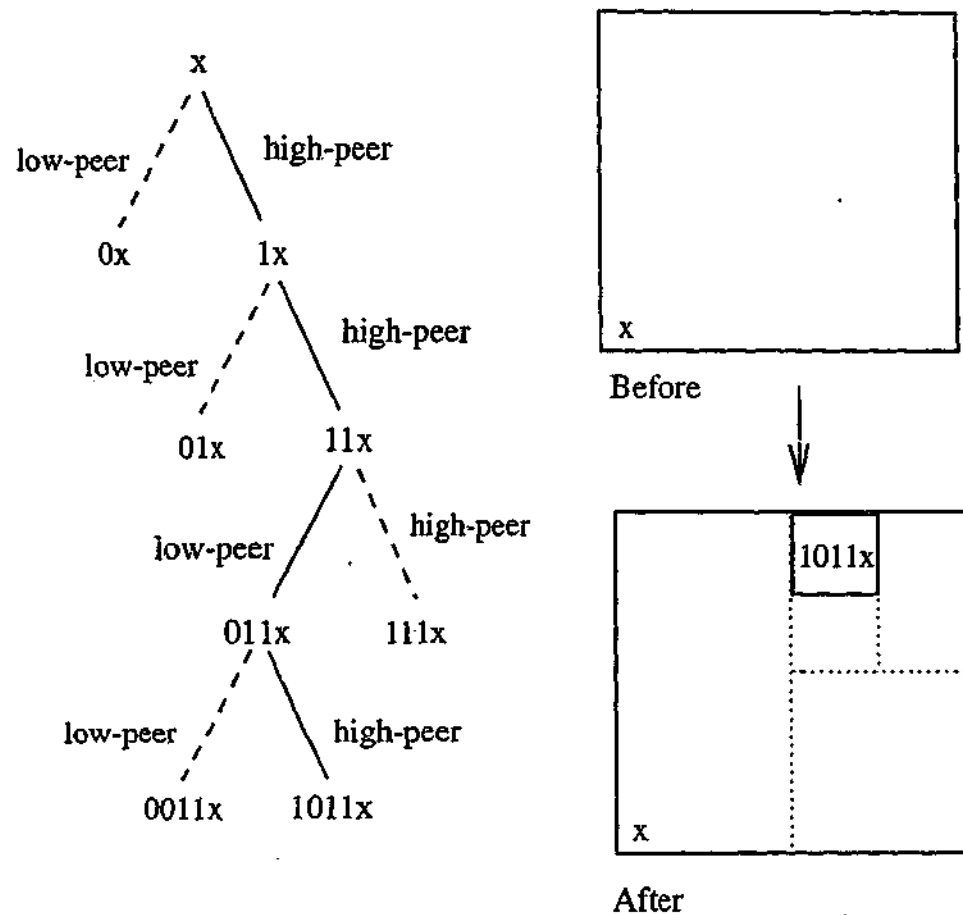


Figure 3.6: Assigning partition-numbers to sub-spaces.

pointer to a right child. The left child corresponds to the low-peer and the right-child to the high-peer. If the partition-number of a node is  $x$  (in binary), then that of its left child is  $0x$  (prepend 0 to  $x$ ) and that of its right child is  $1x$  (prepend 1 to  $x$ ).

An enclosure split can be seen as successive peer-splits performed until a balance is achieved. The partition-number of the enclosing partition is the same as the original partition. The partition-number of an enclosed partition depends on the number of peer splits performed to create it, and whether a low or a high peer was chosen in each peer split. For example, consider

the splitting of a partition with partition-number of  $x$  (in binary) as shown in Figure 3.6. Assume a balance (in number of records) in splitting  $x$  was achieved after four successive splits. In the first split  $x$  was peer splitted into  $1x$  and  $0x$ . Since the number of records in  $1x$  was much more than the rest of  $x$ ,  $1x$  was further split into  $11x$  and  $01x$ . Even after this split a balance was not achieved and the number of records in  $11x$  was much higher than the rest of  $x$ . So  $11x$  was split into  $011x$  and  $111x$ . Even after this split a balance was not achieved for the number of records in  $011x$  was much higher than the rest of  $x$ . Finally,  $011x$  was split into  $0011x$  and  $1011x$  and a balance was achieved between  $1011x$  and the rest of  $x$ . Hence,  $x$  was split between  $1011x$  and the rest of  $x$ . This is shown in Figure 3.6. The solid path of the figure shows the path that were taken in the successive splits.

### 3.6 BANG directory

A BANG file has the structure of a balanced tree, as shown in Figure 3.2. Each non-leaf node contains entries which correspond to some nodes in the next lower level of the tree. If there is an entry in  $P_{i,j}$  which corresponds to  $P_{i,k}$  then  $P_{i,j}$  is the *parent* of  $P_{i,k}$  and  $P_{i,k}$  is the *child* of  $P_{i,j}$ . Each entry consists of a child identifier and its corresponding disk block address.

Like a data-partition, a directory-partition is uniquely identified by a partition-number and a partition-level. These values are computed in the same way as those of the data-partitions. A partition which corresponds to a parent node *encloses* all the partitions of its children.

The number of entries that can be stored in a directory partition is limited. When the number of entries that has to be stored in a directory-partition is greater than its capacity, the directory-partition is divided into two. In the parent node, the entry for the old node is replaced by entries for each of the two new nodes. This process is repeated if as a result the parent node overflows. If the partition that is divided is the root, a new root is created and entries for each of the two new nodes are placed into it.

The algorithm which is used to split a directory-partition is different from that of the data-partition. A data partition encloses data records. Data records are points, so that when a data-partition is split, each of its data records has to be in one side or the other side of the division boundary. A directory-partition is a partition which encloses other partitions. Partitions are not points but subspaces, so when a directory-partition is split, it is possible that one or more of its component partitions lie on both sides of the division boundary.

For example, let us consider the BANG file given in Figure 3.7(a), where  $P_{4,3}$ ,  $P_{4,4}$ ,  $P_{4,5}$ ,  $P_{4,6}$  and  $P_{4,7}$  are data-partitions and  $P_{4,0}$  is the root partition. Let us assume that the maximum number of entries that can be stored in a directory-partition is 5.

If after few insertions  $P_{4,4}$  is split into  $P_{4,8}$  and  $P_{4,9}$  as in Figure 3.7(b), the number of entries in  $P_{4,0}$  will be six hence  $P_{4,0}$  will overflow. If we divide  $P_{4,0}$  using the division algorithm of data-partitions, it could be divided into  $P_{4,1}$  and  $P_{4,2}$ .  $P_{4,1}$  encloses  $P_{4,6}$  and  $P_{4,7}$ , while  $P_{4,2}$  encloses partitions  $P_{4,5}$ ,  $P_{4,8}$  and  $P_{4,9}$ .  $P_{4,3}$  lies on both sides of the division boundary. The unshaded part of  $P_{4,3}$  will lie in  $P_{4,1}$  and the shaded part of  $P_{4,3}$  will lie in  $P_{4,2}$ . The following

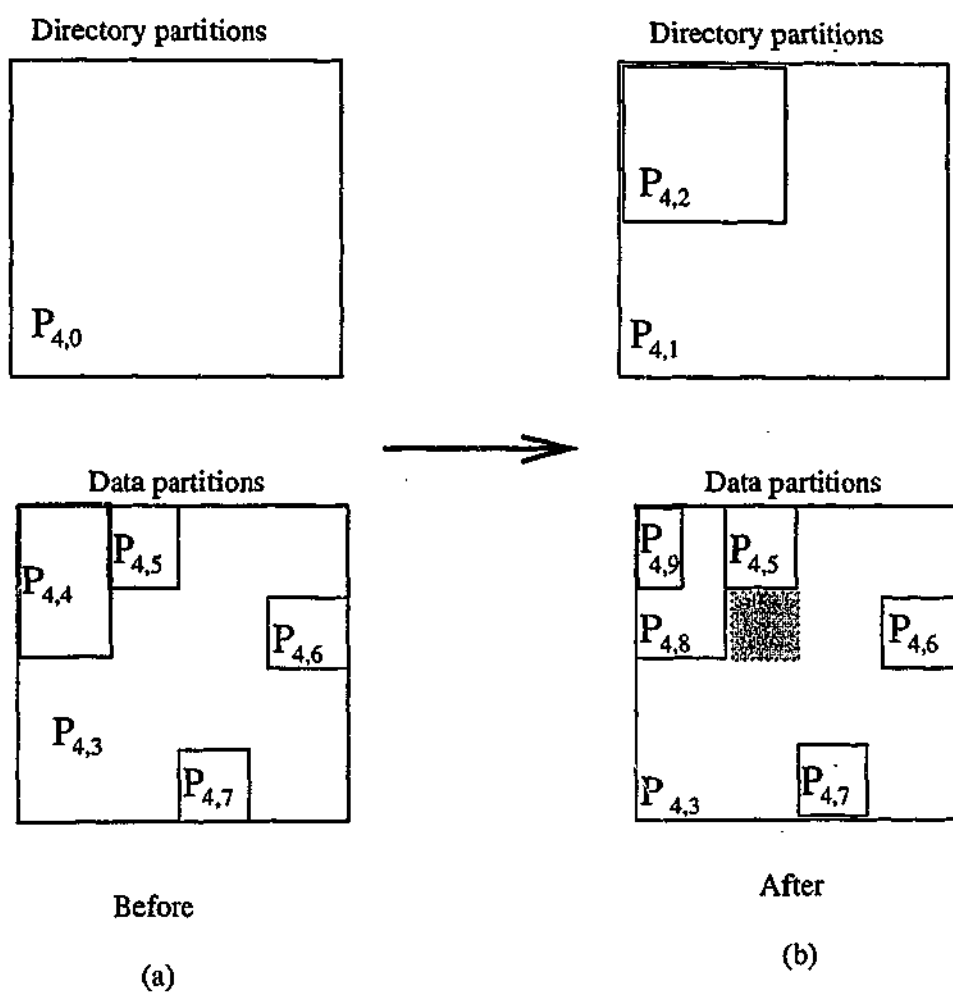


Figure 3.7: Splitting a directory.

are two approaches (there can be plenty others) which can be used to solve the problem of partitions which lie on both sides of the division boundary.

- Divide the partitions which lie on both sides of the division boundary along the division boundary [36]. This process can recursively propagate to lower level nodes if the same situations occurs, forcing the division of partitions even if they have few entries. Using this approach, to split the partitions of Figure 3.8, we need to split  $P_{4,3}$  even if it is not full. This approach can result in a lower load factor (number of entries per partition) and higher insertion cost [37].
- Choose a division boundary such that no partition lies on both sides of the boundary. One such splitting algorithm was also presented Freeston in [37] and it works as follows:

Assume  $P_{i,j}$  is the partition to split. An initial boundary which best splits the content of  $P_{i,j}$  is chosen. Let us name this boundary as B1. If there is no partition in  $P_{i,j}$  which encloses B1 or there is one partition whose boundary coincides with that of B1, then  $P_{i,j}$  is split along B1. Otherwise, search for two partitions in  $P_{i,j}$ , one which directly encloses B1, and the second one which does not enclose B1 but which encloses the highest number of partitions in  $P_{i,j}$ . Let us call the boundary of the first partition B2 and that of the second one B3. Then split  $P_{i,j}$  along B2 or B3 depending on which one of them best splits  $P_{i,j}$ , that is, provides the best balance of entries within the resulting partitions.

For example,  $P_{0,0}$  of Figure 3.8a, is a parent of the 7 partitions shown in the figure. If the maximum number of entries that can be stored in a parent

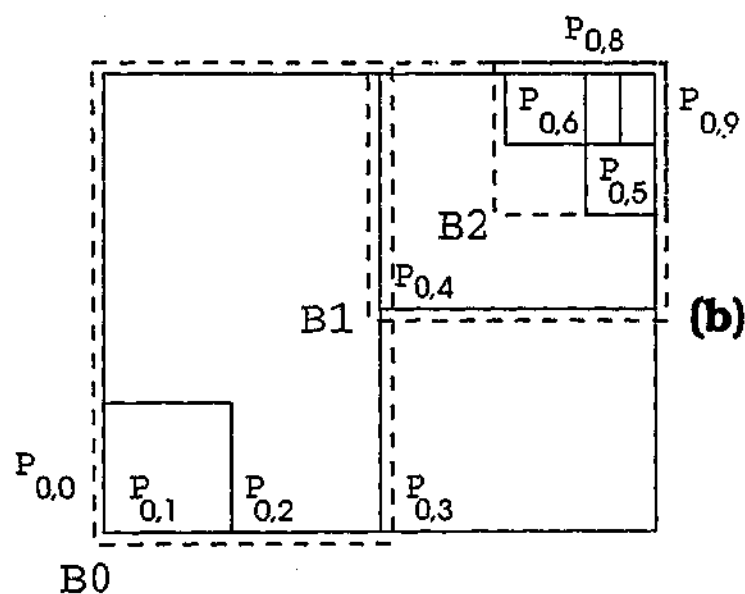
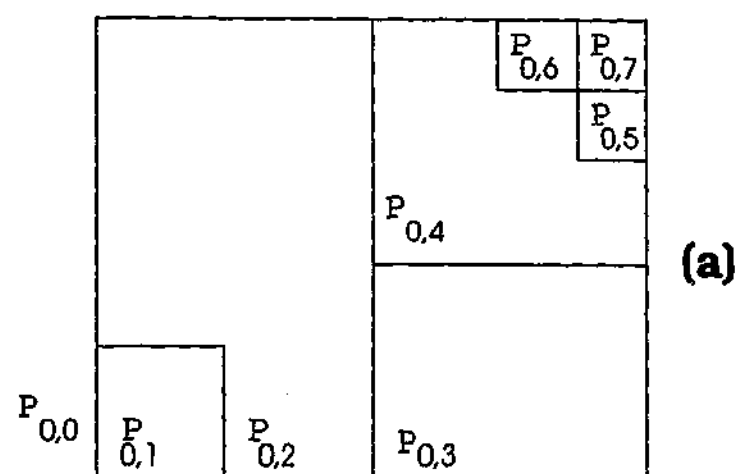


Figure 3.8: Splitting a directory partition.

partition is 7, then  $P_{0,0}$  is full. If as a result of insertion  $P_{0,7}$  is divided into  $P_{0,8}$  and  $P_{0,9}$ , as shown in Figure 3.8b, then the number of entries in  $P_{0,0}$  will be 8, hence it must be split. The initial boundary, B1, which best splits  $P_{0,0}$  is shown in Figure 3.8b. This boundary contains four of the eight partitions, namely,  $P_{0,5}$ ,  $P_{0,6}$ ,  $P_{0,8}$  and  $P_{0,9}$ . But B1 cannot be used to split  $P_{0,0}$  because:

1. it doesn't coincide with the boundary of a partition;
2. it is enclosed by  $P_{0,4}$ .

Hence we search for two other boundaries:

1. B2 which directly encloses B1 and
2. B3 which doesn't enclose B1 but which encloses the highest number of other partitions.

B2 and B3 are shown in Figure 3.8b. B2 splits  $P_{0,0}$  into two partitions. The first one will contain five partitions,  $P_{0,4}$ ,  $P_{0,5}$ ,  $P_{0,6}$ ,  $P_{0,8}$  and  $P_{0,9}$ , and the second one will contain the rest,  $P_{0,1}$ ,  $P_{0,2}$  and  $P_{0,3}$ . Also, B3 splits  $P_{0,0}$  into two partitions. The first one will contain six partitions,  $P_{0,3}$ ,  $P_{0,4}$ ,  $P_{0,5}$ ,  $P_{0,6}$ ,  $P_{0,8}$  and  $P_{0,9}$ , and the second one will contain two partitions,  $P_{0,1}$  and  $P_{0,2}$ . Since B2 best splits  $P_{0,0}$  than B3,  $P_{0,0}$  will be split along B2.

Out of the two above mentioned approaches of splitting directory partitions, we chose the latter one because it results in a higher load factor and lower insertion cost than the former [37].

### 3.7 Searching, insertion, deletion and merging

The search for a record starts at the root directory node. From the root entries, a partition which encloses the record is chosen. If there are two or more partitions which enclose the record, the one which directly encloses the record is chosen. This is the one with the higher partition-level. The search then descends to the next lower level of the BANG file directory tree, following the page identifier from the chosen entry. The search procedure is repeated within the current level. This search procedure is applied at each level of the tree until the data-partition which directly encloses the record is found.

Insertion of a record starts by first searching for a partition which will directly enclose the record. The search for the partition starts at the root directory node. From the root entries, a partition which encloses the record is chosen. If there are two or more partitions which enclose the record, the one which directly encloses the record is chosen. As explained in the previous paragraph, this is the one with the higher partition-level. The search then descends to the next lower level of the BANG file directory tree, following the page identifier from the chosen entry. The search procedure is repeated within the current level. This search procedure is applied at each level of the tree until the data partition which directly encloses the record is found.

The deletion of a particular record starts by first searching for the data partition that directly encloses the record. The search for the data partition is done as explained above in this same section. Then the record is searched



within that page and deleted.

If as a result of many deletions, the number of records in a partition falls below certain threshold, the partition can be merged with others were possible. To merge an underpopulated partition, an attempt is made to first merge it with one of the partitions it immediately encloses. If such a partition cannot be found or the merging results in an overflow an attempt is made to merge it with its peer. Again if the merging with its peer was not successful, a last attempt is done to merge it with a partition which directly encloses it. A merge of two partitions means a deletion of a partition. Hence a successful merge can again result in underpopulated partition of the next upper directory level. As a result a successful merge can propagate upwards the tree [36].

### 3.8 Binary division

In the BANG file as in many other PAMs, splitting a partition is done by a process known as a *binary division*. In the context of this thesis a *binary division* is the division of a domain into edges (sub-domains) whose sizes are  $2^{-k}$  (where  $k = 0, 1, 2 \dots$ ) of the domain. To simplify the understanding of binary division, let us assume  $D_{i,j}$  as a domain of unsigned integers ranging from 0 to  $2^{d_{i,j}} - 1$ . So the minimum number of bits needed to represent any value in  $D_{i,j}$  is  $d_{i,j}$ . Let  $b_{i,j,k}$ , where  $0 \leq k < d_{i,j}$ , represent the  $k^{th}$  most significant bit of a value in  $D_{i,j}$ . For example, if  $d_{i,j}$  is 6, then  $D_{i,j}$  will range from 0 to  $2^6 - 1 = 63$ . The values of  $b_{i,j,5}$ ,  $b_{i,j,4}$  and  $b_{i,j,3}$ ,  $b_{i,j,2}$ ,  $b_{i,j,1}$  and  $b_{i,j,0}$  of 7 (000111 in binary) are 0,0,0,1,1 and 1 respectively.

The value of the most significant bit position can be used to divide the values in  $D_{i,j}$  into two equal halves. For example, if  $D_{i,j}$  values range from 0 to 63, then a minimum of 6 bits is needed to represent all its values. For values less than 32, the value of the most significant bit position,  $b_{i,j,5}$ , is 0, and for those which are greater or equal to 32 it is 1. The two most significant bit positions,  $b_{i,j,5}$  and  $b_{i,j,4}$ , can be used to divide the values of  $D_{i,j}$  into 4 equal intervals. Values of  $b_{i,j,5}$  and  $b_{i,j,4}$  are:

- 0 and 0 respectively for the values  $\geq 0$  and  $< 16$ ,
- 0 and 1 respectively for the values  $\geq 16$  and  $< 32$ ,
- 1 and 0 respectively for the values  $\geq 32$  and  $< 48$ ,
- 1 and 1 respectively for the values  $\geq 48$  and  $< 64$ .

Once  $D_{i,j}$  is divided into 4 equal intervals, the next less significant bit position can be used to further divide each of the 4 intervals into two equal halves and so on. In short the  $x^{th}$  most significant bit can be used to bisect each interval that was created as the result of the  $x - 1^{th}$  most significant bit. The process of dividing domains in such a way is called binary division.

### 3.9 Choice vectors

As was discussed in section 3.4, peer-splitting a partition is done by bisecting one of its  $n$  edges (sub-domains). An (edge) is the extent of a partition along a particular domain. The edge to be bisected cannot be chosen randomly for the following two reasons.

1. The partition-number as it was discussed in the section 3.5 will be useless. So it will be impossible to identify a partition.
2. As will be discussed in this and the next few chapters, the order by which a partition is divided significantly affects query cost.

The order by which a partition is divided is maintained in an ordered list known as a *choice vector*. Each element of a choice vector is a bit position. The mapping between an element of a choice vector (a bit position) and an edge of a partition was discussed in section 3.8. Since peer-splitting a partition is done by bisecting one of its edges, is in fact a binary division. Therefore peer-splitting a partition can be done by using bit positions and that is why each element of a choice vector is a bit position. An element of a choice vector is labeled as  $b_{i,j,k}$  and represents the  $k^{th}$  most significant bit of a  $D_{i,j}$  value. Elements of a choice vector are ordered and are used accordingly. Bit positions which create larger intervals come first in the order, which means an element of a choice vector which divides a domain into  $2^x$  must be used before the one that divides the same domain into  $2^{x+1}$ . We know from section 3.5 that the partition-level of a partition shows the size of a partition relative to the domain-space. For example, if the size of the partition is  $\frac{1}{2^k}$  that of the domain-space, then its partition-level is  $k$ . The choice vector element to be used for the next split is specified by the partition-level of the partition. So when we divide a partition whose partition-level is  $k$  we use the  $k^{th}$  element of the choice vector. For example, if  $b_{0,0,0}, b_{0,1,0}, b_{0,0,1}, b_{0,1,1}, b_{0,0,2}$  is a choice vector of  $R_0$  which has 2 attributes,  $A_{0,0}$  and  $A_{0,1}$ , then a partition with partition-level of 2 is peer-split by  $b_{0,0,1}$ .

The First choice vector element, the left most one, creates partitions which are half the size of the domain-space, while the second element creates partitions which are a quarter the size of the domain-space, the third element creates  $\frac{1}{8}$  and the  $x^{th}$  element creates  $\frac{1}{2^x}$  that of the domain-space and so on.

Choice vector affects the way a domain-space is partitioned which in return affects the cost of a query. Bad choice vector significantly add to the cost of a query and good choice vector significantly minimize query cost. Chapters 4 to 7 will explain, in details, the relationship between choice vectors and query cost and how to find a choice vector which results in the minimal average query cost. But first lets us discuss the two main types of choice vectors used in this thesis, namely, cyclic choice vectors and optimized choice vectors.

### 3.9.1 Cyclic choice vector

A *cyclic choice vector* is a choice vector whose elements are assigned bit positions from different attributes in a cyclic fashion. It is the choice vector that is used by the existing BANG file [37].  $b_{i,0,0}b_{i,1,0}b_{i,2,0}b_{i,0,1}b_{i,1,1}b_{i,2,1}b_{i,0,2}b_{i,1,2}b_{i,2,2}$ , is an example of a cyclic choice vector for a relation,  $R_0$ , which has 3 attributes, labeled  $A_{i,0}$ ,  $A_{i,1}$ , and  $A_{i,2}$ . The most significant bit position of  $A_{i,0}$  is assigned to the first element of the choice vector,  $b_{i,0,0}$ , then the most significant bit position of the second attribute,  $A_{i,1}$  is assigned to the second element of the choice vector, then the most significant bit position of the third attribute,  $A_{i,2}$  is assigned to the third element of the choice vector and so on.

As was mentioned in section 1.2, the way the data is organised affects performance. Since the way data is organised is enforced by a choice vector, choosing the right choice vector is very important. When there is no clue of the type and distribution of the queries to be used, the data can be organised using the cyclic choice vector. In Chapters 4 to 7, we will compare the performance, in terms of query cost, of the cyclic choice vector and that of the optimal choice vectors. Unlike that of the cyclic choice vector the finding of an optimal choice vector takes into consideration the query distribution. The elements of an optimal choice vector are assigned by using heuristic algorithms and some cost functions and will be discussed in the next section.

### 3.9.2 Optimized choice vector

Choice vectors affect query cost. For queries with known probabilities, it is possible to approximate the choice vector which results in their minimum average cost. Such a choice vector is called an *optimized choice vector*. Moran in [94] showed that for the general problem when the probability of an attribute appearing in a query is not independent of the other attributes, finding the optimal bit allocation is NP-hard. Hence, in this thesis heuristic algorithms, which are explained in section 2.6, together with some cost functions (which will be explained in the coming chapters) will be used to determine good choice vectors.

For example, finding optimized choice vector using minimal marginal increase, which was explained in section 2.6.2, and a given cost function  $F$ , is done as follows. Initially nothing is allocated to the elements of the choice vector. Then the first element of the choice vector is allocated to  $b_{i,0,0}$  and

the average query cost using cost function  $F$  is computed. The average query cost is repeatedly computed after giving the same choice vector element instead to  $b_{i,1,0}$ , and then to  $b_{i,2,0}$ , and so on, until all the attributes are tried. The attribute which gives the lowest average query cost is permanently allocated the first element of the choice vector. The same process is repeated for the second element of the choice vector, then the third, and so on. This process is repeated until all the elements of the choice vector are allocated.

From now on the term *minimal* or *optimal* will be used to indicate a solution which is the result produced by the heuristic algorithms and cost functions. These minimal or optimal solutions are typically local minima or local optima respectively.

### 3.9.3 Choice vector size

To create a partition whose size is  $\frac{1}{2}$  of that of the domain-space, the first element (the right most) of the choice vector is used. To create a partition whose size  $\frac{1}{4}$  that of the domain-space, the second element of the choice vector element is used. The third element of the choice vector creates partitions whose sizes are  $\frac{1}{8}$  that of the domain-space. In short to create a partition whose size is  $2^{-k}$  that of the domain-space,  $k^{th}$  element of the choice vector is used. Hence the size of the choice vector is decided by the size of the smallest partition in the domain-space.

In the BANG file the size of a choice vector needed becomes smaller and smaller as we go up the directory level. This is because partitions of a higher directory level encloses many lower level partitions. This makes the number

of partitions of an upper level directory lower and their sizes larger than those which are in the lower directory level.

Each BANG file uses one choice vector for both its data and directory partitions. But as we go up the directory level, more of the elements on the right end of the choice vector are not used. For example in a BANG file whose smallest data partition is  $2^{-8}$  of the domain-space, the minimum size of the choice vector is 8. If the size of smallest partition in the lowest directory level is  $2^{-6}$ , the last two right most elements of the choice vector will not be used in this directory level.

### 3.10 Effect of choice vectors on the load factor

The data of the original BANG file [37] was organised using a cyclic choice vector. In this thesis, and for the first time, the BANG file was extended using non-cyclic choice vectors. The main objective of using non-cyclic choice vectors is that most optimal choice vectors are non-cyclic.

Inorder to prove that none of the original BANG file features were compromised as a result of using non-cyclic choice vectors, we conducted a number of experiments. In the experiments three types of choice vectors were used, namely, cyclic, optimised and random. The cyclic and the optimised choice vectors were explained in sections 3.9.2 and 3.9.1 respectively. The random choice vectors were generated randomly.

One of the main features of the original BANG file is its load factor. It is a main feature of the BANG file because change in the load factor affects

many of the other BANG file features that are mentioned in section 3.1. So to analyse the impact of a choice vector on the load-factor of the BANG file, four sets of experiments were conducted. In each set of experiment all the three types of choice vectors and one of four parameters, namely, data distribution, number of attributes, number of records and disk block size were used. Then we compared the results with that of original BANG file [37], which only uses cyclic choice vectors. The experimental results are presented in Figures 3.10 to 3.19. They indicate that none of the aforementioned parameters have a noticeable effect on the load-factor of the BANG file. Therefore, for a given query distribution we can use the optimal choice vector to obtain the best performance in terms of the average query cost without increasing the storage cost.

### 3.10.1 Experimental results and analysis

This subsection presents the experimental results. The results are logically divided into four sets. The first set of results shows how the load factor of a BANG file is affected by using non-cyclic choice vectors and different data distributions. The second set of results shows how the load factor of the BANG file is affected by non-cyclic choice vectors and different number of attributes. The third and the fourth sets of results show the effect of non-cyclic choice vector on the load factor when each non-cyclic choice vector is used with different number of records and different block size respectively.



Unless specified each experiment is conducted using:

- SPARCstation 20 using SunOS 5.5.1;
- a BANG file of a million records;
- relations of 4 attributes;
- domains of unsigned integers;
- a disk block size of 1 Kbytes.

#### Non-cyclic choice vector vs data distribution

This subsection presents the first set of results. The results are obtained by experimenting each type of a choice vector with different data distributions. The data distributions used were chosen from the set: *uniform*, *clustered*, *sinusoidal*, and *linear*. Figure 3.9 provides examples of each of these data distributions.

The final results are shown in Figures 3.10, 3.11, 3.12 and 3.13. The horizontal axis of each figure represents the number of records and the vertical axis represents the load factor. Each figure shows three results and are represented as dotted, dashed and solid lines. The dotted line represents the average load factor when a cyclic choice vector is used. The dashed line represents the average load factor when a random choice vector is used. The average load factor when an optimised choice vector is used is represented by the solid line. The three results shown in Figure 3.10 were generated using a uniformly distributed data. those shown in Figures 3.11, 3.12 and 3.13 were

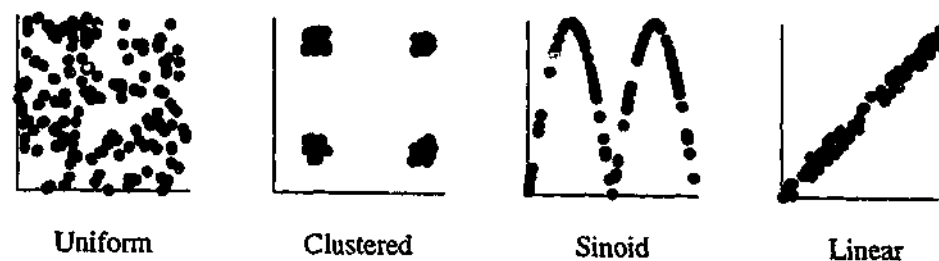


Figure 3.9: The four data distributions used in generating the results.

generated using clustered, sinusoidal, and linear data distributions respectively. The results show that choice vector has no effect on the load-factor when used with different data distributions.

The results of the uniform and clustered distribution have a sinusoidal shape while that of the other two have a strait line shape. The reason for this is that in a uniform and in a clustered distributions, nearly all the data blocks become full in nearly the same time and as a result they will be split at nearly the same time. That is why the load factor goes up and down as is shown in Figures 3.10 and 3.11. The data blocks in a linear and sinusoidal distributions don't become full in the same time, hence they don't split at the same time thus on the average the load factor remains the same all the time.

#### Non-cyclic choice vectors vs number of attributes

To study how the load factor is affected when used with different number of attributes and different choice vectors, we experimented using relations of 2, 3, 4 and 8 attributes. Figures 3.14 to 3.17 show the results of the experiments. The horizontal axis of the figures represents the number of records and the vertical axis represents the load factor.

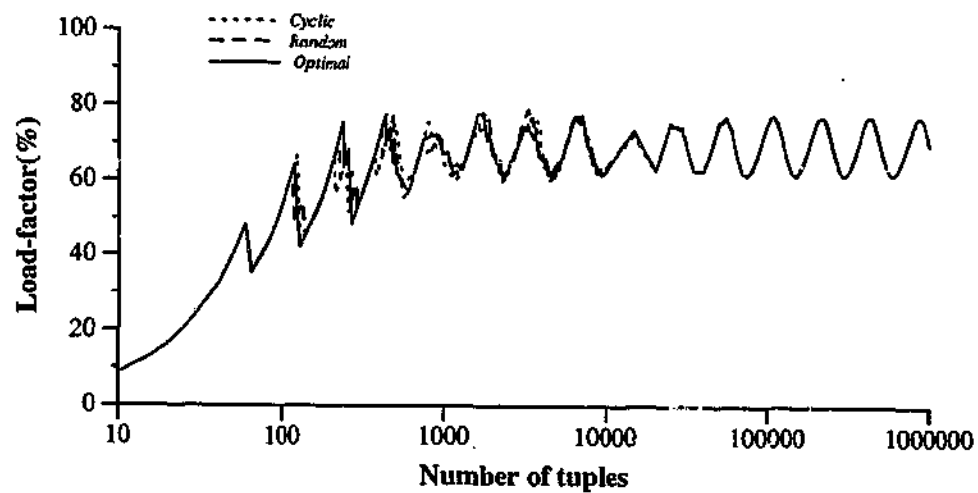


Figure 3.10: Effect of the choice vector on the load-factor. Number of attributes = 4, page size = 1024 bytes, data distribution = uniform.

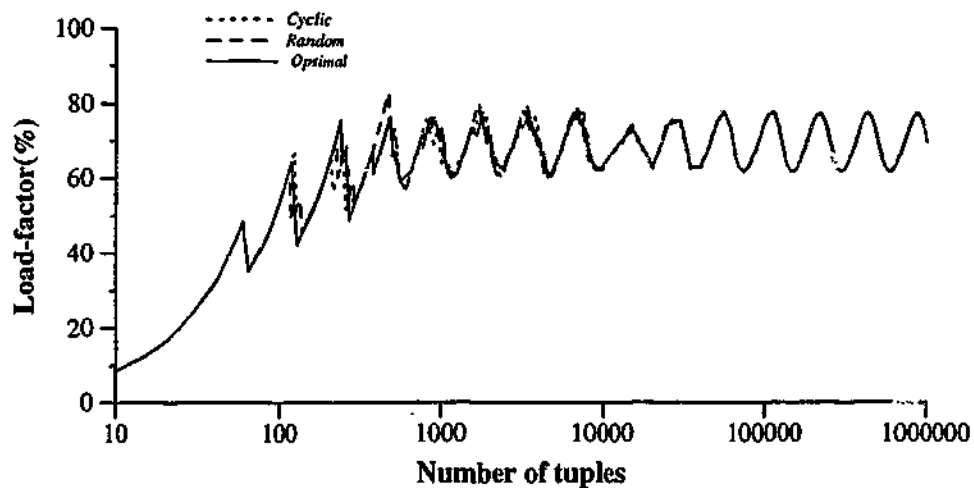


Figure 3.11: Effect of the choice vector on the load-factor. Number of attributes = 4, page size = 1024 bytes, data distribution = clustered.

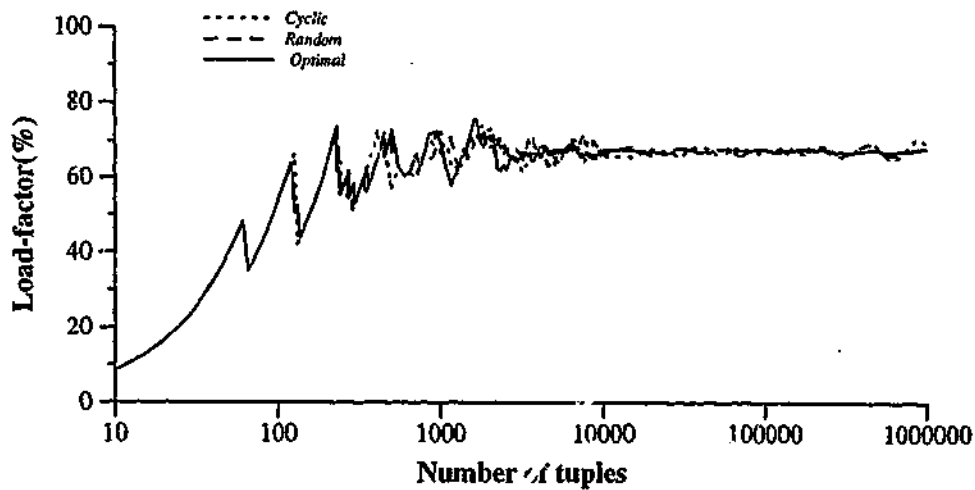


Figure 3.12: Effect of the choice vector on the load-factor. Number of attributes = 4, page size = 1024 bytes, data distribution = sinusoidal.

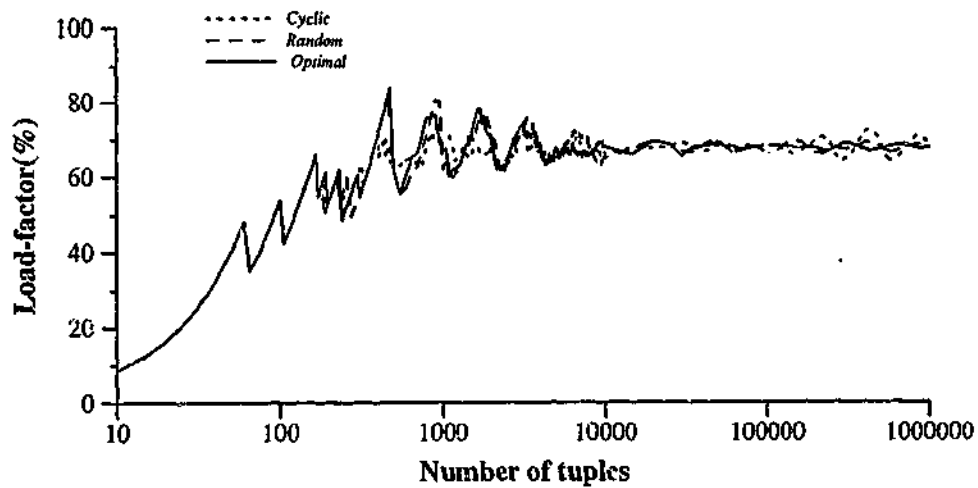


Figure 3.13: Effect of the choice vector on the load-factor. Number of attributes = 4, page size = 1024 bytes, data distribution = linear.

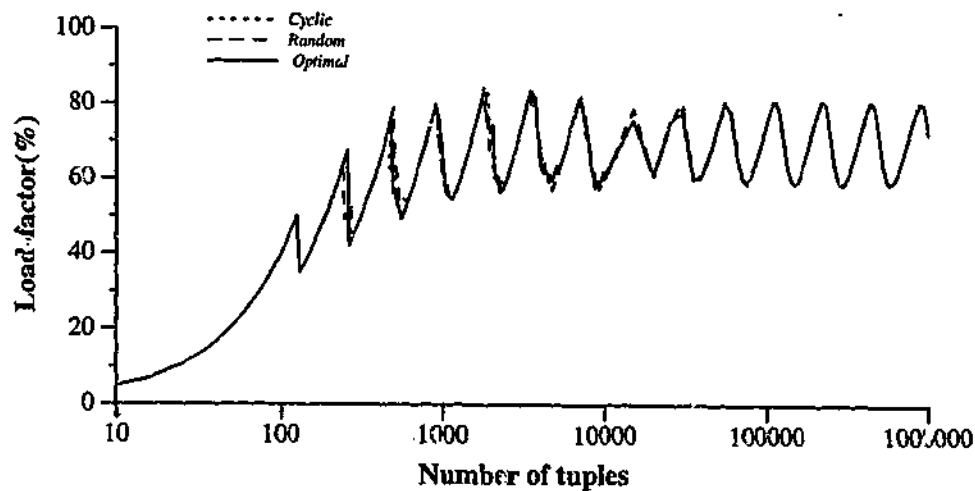


Figure 3.14: Effect of the choice vector on the load-factor. Number of attributes = 2, page size = 1024 bytes, data distribution = uniform.

Figure 3.14 shows three results as dotted, dashed and solid lines. The dotted line represents the results obtained when cyclic choice vectors are used. The dashed and the solid lines represent the results obtained when random and optimal choice vectors were used respectively. All the three results were generated using relations having two attributes. Similar results were obtained when relations with three, four and eight attributes were used as shown in Figures 3.15 3.16 and 3.17 respectively. These results show that choice vectors have no effect on the load-factor even when different number of attributes are used. The load-factor always remained around 67% as that obtained by Freeston in [37].

#### Non-cyclic choice vectors vs number of records

To study the effect of choice vectors on load-factor when used with different number of records we experimented using relations having from 100000 to 1000000 records. Figures 3.10 to 3.19 show the results of the experiments.

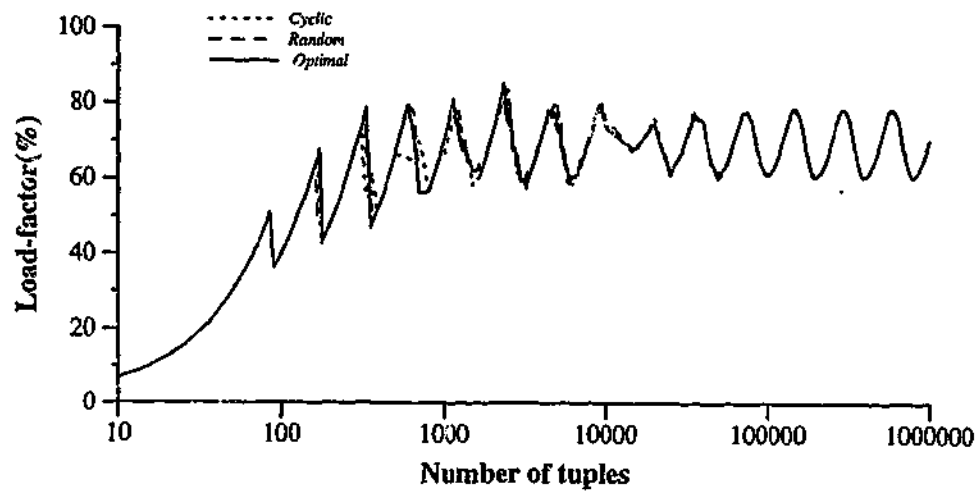


Figure 3.15: Effect of the choice vector on the load-factor. Number of attributes = 3, page size = 1024 bytes, data distribution = uniform.

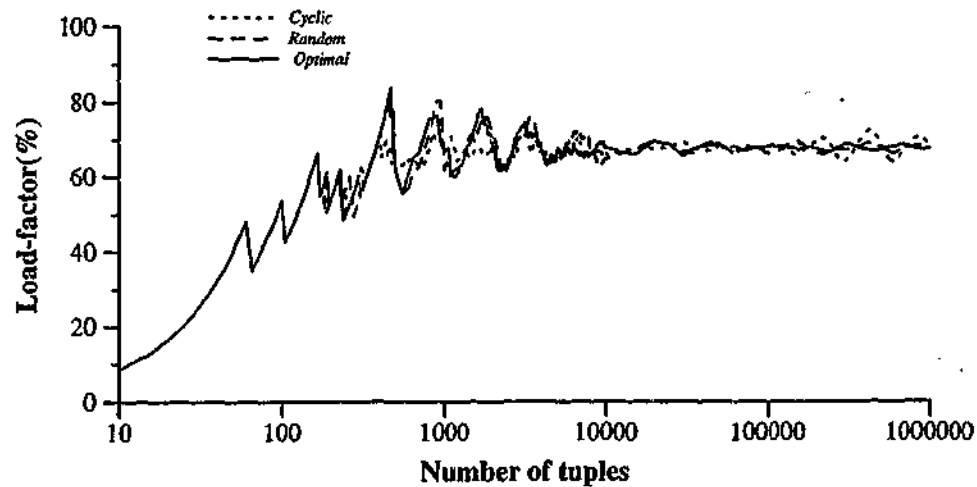


Figure 3.16: Effect of the choice vector on the load-factor. Number of attributes = 4, page size = 1024 bytes, data distribution = linear.

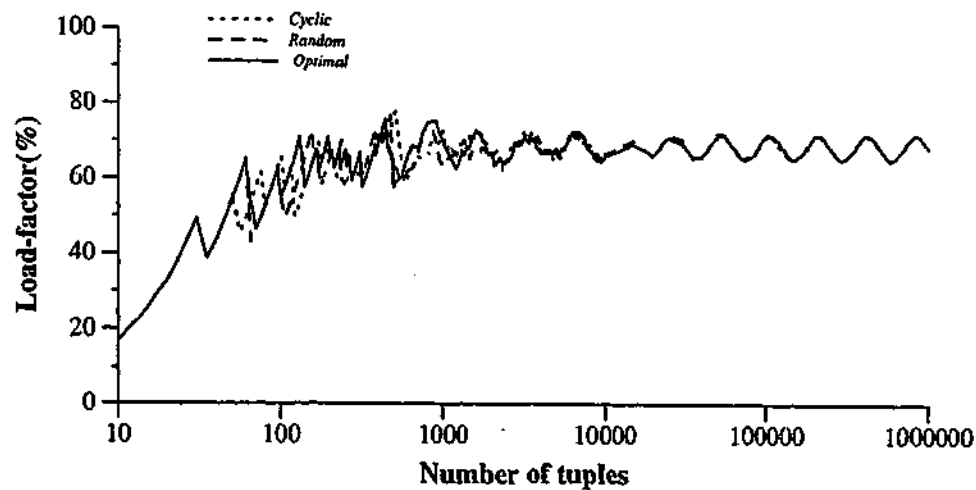


Figure 3.17: Effect of the choice vector on the load-factor. Number of attributes = 8, page size = 1024 bytes, data distribution = uniform.

The horizontal axis of the figures represents the number of records and the vertical axis represents the load factor. Each figure shows three results which are represented as dotted, dashed and solid lines. The dotted line was generated using cyclic choice vectors, The dashed line was obtained using random choice vectors and the solid line was obtained using optimised choice vectors. In all the cases the load factor remained around 67%. This implies that choice vectors have no effect on the load-factor even when used with different number of records.

#### Non-cyclic choice vectors vs block size

The fun out, which is the number of directory records that can be put in a disk block, affects performance. If the number of directory records that can be put in a disk block is small, the BANG tree will have more number of levels. Increasing the level of the BANG file by one, causes the number disk accesses done to answer a query to increase by one.

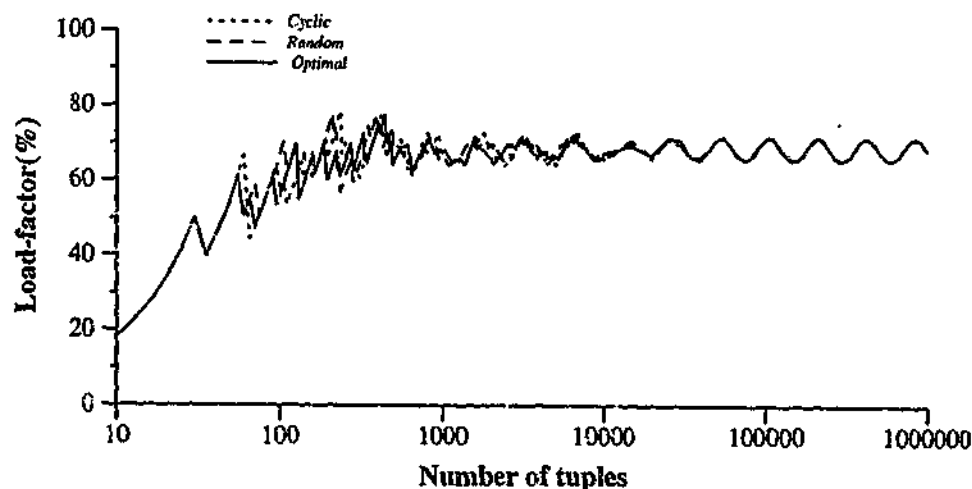


Figure 3.18: Effect of the choice vector on the load-factor. Number of attributes = 4, page size = 512 bytes, data distribution = uniform.

To study the effect of choice vectors on load-factor when used with different block sizes, we experimented with block sizes of 512, 1024, 2048, 4096 and 8192 bytes. Figure 3.18 to 3.19 show some of the results of the experiments. The horizontal axis of the figures represents the number of records and the vertical axis represents the load factor. Each figure shows three results which are represented as dotted, dashed and solid lines. The dotted line was generated using cyclic choice vectors, The dashed line was obtained using random choice vectors and the solid line was obtained using optimised choice vectors.

The results show that the load-factor is around 67%. This implies that choice vectors have no effect on the load-factor even when used with different block sizes.



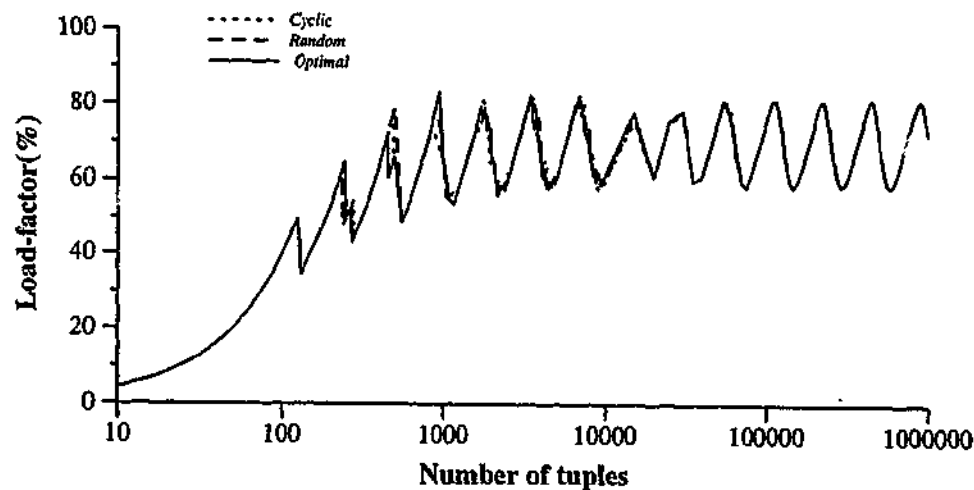


Figure 3.19: Effect of the choice vector on the load factor. Number of attributes = 4, page size = 2048 bytes, data distribution = uniform.

### 3.11 Conclusion

A BANG file structure is a dynamic multidimensional file structure. It has a directory structure of a balanced tree. It is similar to the other grid files but it differs from them in the way its partitions are created.

There are two types of partitions in the BANG file, namely, data partitions and directory partitions. Both types of partitions are labeled in the same way. The unique identifier (label) of a BANG file partition has two parts, namely, partition-level and partition-number. Partition number denotes the position of the partition in the domain-space of BANG file, and partition level denotes the size of the partition relative to that of the whole domain-space.

The main reason that the BANG file was chosen for the experiments of this thesis is because of its ability to distribute records amongst available disk blocks (with a load-factor of 67%) even when the data distribution is skewed. One of the reasons that the BANG file has such an attribute is

because, in the BANG file a partition can enclose other partitions and it can also be enclosed by other partitions.

One way of improving the performance of algorithms manipulating data on secondary storage is to cluster similar data. The clustering of data will be of most benefit if it results in the reduction of the time taken to perform operations which are frequently required of the database management system. To achieve the optimal performance, the frequency, type and the cost of each operation must be taken into account when designing a clustering arrangement.

In the BANG file as in many other PAMs, clustering related data is achieved by using choice vectors, which is a structure which maintains the order by which a domain-space is split. Splitting the domain-space of the BANG file is done by a process known as a *binary division*. In a binary division each edge of a partition has a size of  $2^{-k}$  (where  $k = 0, 1, 2, \dots$ ) that of its corresponding domain.

There are three types of choice vectors, namely, cyclic, random and optimized. The original BANG file uses only the cyclic choice vector. In this thesis the BANG file was extended by using non-cyclic choice vectors. To make sure that such an extension was not done at the expense of other BANG file properties, specially its load factor, a series of experiments were performed. Experiments were done using non-cyclic choice vectors together with different data distribution, different number of attributes, different number of records and different disk block sizes. In all the experiments the load factor remained as that of the original BANG file, 67%. This shows that choice vectors have no effect on the load factor of the BANG file.

Choice vectors affect performance. So if the distribution of a set of queries is known, then it is better to use a choice vector which will minimize the average cost of the set. Such a choice vector is known as optimized choice vector. Techniques of finding optimised choice vectors for partial match queries will be discussed in the next chapter and that of the other relational operations will be discussed in subsequent chapters.

## Chapter 4

# Optimising Partial-match Queries

### 4.1 Introduction

Partial-match retrieval is one of the most important class of queries in a database system retrieval. It is concerned with the retrieval of records in a file when a limited amount of information is provided to identify those records. Answering a partial-match query requires reading all the disk blocks that may contain matching records. The following query is an example of a partial-match query:

```
SELECT studentID  
FROM student  
WHERE major = 'Computer Science' (q0)
```

where *student* is a relation, and *studentID* and *major* are some of its attributes.

A common method used to evaluate the performance of a database system is to count the average number of disk accesses made to answer a query. To minimize the average query cost, the average number of disk accesses needed to answer a query must be minimized. One way to achieve this is by using efficient access structures to cluster records that are frequently accessed together in the smallest possible number of disk blocks.

The number of disk blocks retrieved depends on the algorithm used to place the records in the blocks within the file. The average number of disk blocks accessed per query can be minimized if an efficient record placement algorithm which takes the query distribution into account is used.

In this chapter, we present a technique of clustering records in multidimensional structures which minimizes the average cost of partial-match query. Researchers have proposed different ways of clustering records in multidimensional file structures, but few have tried to optimize their clustering technique in order to reduce the average query cost. Those that have done so [3, 51, 52, 83, 84, 116] were limited to uniform data distribution. To avoid this limitation, we use a multidimensional file organization that evenly distribute records among the allocated disk blocks even when the data distribution is skewed. This is the first study of optimizing partial-match queries when the data distribution is non-uniform. Data distributions are often non-uniform in real application domains and therefore this study is important [138].

Although the techniques described in this chapter were experimentally tested using the BANG file, they can be used to optimize other multidimensional file structures.

This chapter has 7 sections. Section 4.2 is an introduction to partial match queries. Section 4.3 explains a partial-match retrieval algorithm using the BANG file. Section 4.4 discusses a technique of optimising partial-match queries using minimal marginal increase (MMI) together with the associated cost functions. The cost functions are explained in detail in Section 4.5. Section 4.6 presents the analysis and the experimental results of the proposed technique. Section 4.7 is the conclusion.

## 4.2 Partial-match retrieval

A partial-match query is a specification of the values of zero or more attributes in a record. Answering partial-match queries requires accessing all the disk blocks which may hold records satisfying the condition specified in the WHERE clause of the query. For example, in the following query,  $q_1$ , which uses relation  $R_0$  of Figure 4.1, the values of the attribute  $A_{0,0}$  is specified as 40. Let us call partitions which contain records which satisfy the WHERE condition of a partial-match query as  $\Phi$ -partitions.

```
SELECT  $A_{0,1}$ 
FROM  $R_0$ 
WHERE  $A_{0,0} = 40$                                 ( $q_1$ )
```

The answer to the query,  $q_1$ , consists of records whose  $A_{0,0}$  value is 40. These records can be found in four disk blocks:  $P_{0,4}$ ,  $P_{0,5}$ ,  $P_{0,6}$  and  $P_{0,7}$ . So these four disk blocks are the  $\Phi$ -partitions of  $q_1$ .

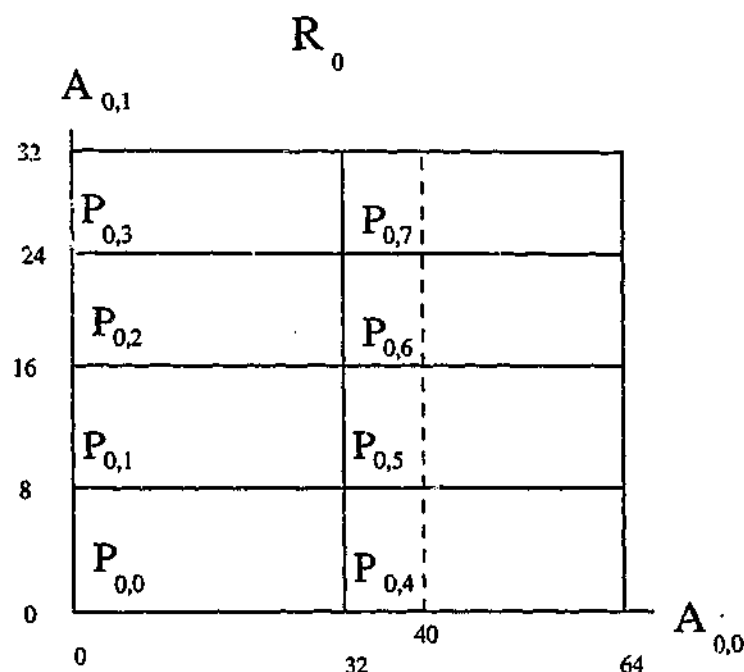


Figure 4.1: relation  $R_{0,0}$ .

### 4.3 Partial-match retrieval Algorithm using The BANG file

The partial-match retrieval algorithm starts by creating a *search string*. Each element of a search string is either 0 or 1 or "\*". The length of a search string is equal to the size of the choice vector which belongs to the relation used in the query. Each element of a search string corresponds to a choice vector element. Each element of the search string which corresponds to a specified attribute is assigned a 0 or a 1, depending on the value specified. Each element which corresponds to the unspecified attribute is assigned a "\*". In Sections 2.3 it was discussed that the most significant bit of an attribute  $A_{i,j}$  is represented as  $b_{i,j,0}$  and the second most significant bit is represented as

$b_{i,j,0}$  and so on. In short the  $k^{th}$  most significant bit of  $A_{i,j}$  value is represented as  $b_{i,j,k}$ . For example, consider the above mentioned query  $q_1$  which uses  $R_0$  of Figure 4.1. The values of  $A_{0,0}$  range from 0 to 31, so a maximum of 6 bits is enough to represent a value in  $A_{0,0}$ . The most significant bit is of a value in  $A_{0,0}$  is represented by  $b_{0,0,0}$  and the second most significant bit is represented by  $b_{0,0,1}$  and the least significant bit by  $b_{0,0,5}$ . Similarly a maximum of 5 bits is enough to represent any value in  $A_{0,1}$ . Let the choice vector of  $R_0$  be  $b_{0,0,0}b_{0,1,0}b_{0,1,1}$ . In the example query,  $q_1$ ,  $A_{0,0}$  is specified as 40 (101000 in binary) therefore  $b_{0,0,0}$  is 1. Since  $A_{0,1}$  is not specified its corresponding bits in the choice vector,  $b_{0,1,0}$  and  $b_{0,1,1}$ , are each assigned a "\*". Hence the search string (search index) is 1\*.\*

In a BANG file the search for the  $\Phi$ -partitions starts from the root partition. The partition-number in each entry of the root is converted to binary, then inverted and then matched with the search index. Only the left most  $m$  (where  $m$  is equal to the partition-level of the partition) bits of the inverted partition-number are matched with their corresponding elements of the search string. A partition to be an  $\Phi$ -partition, every bit of its inverted partition-number must match its corresponding element from the search string whose value is not "\*". The search for the  $\Phi$ -partitions descends to the next lower level directory using the entries of the  $\Phi$ -partitions identified so far. These process is repeated until all the  $\Phi$ -partitions are identified.



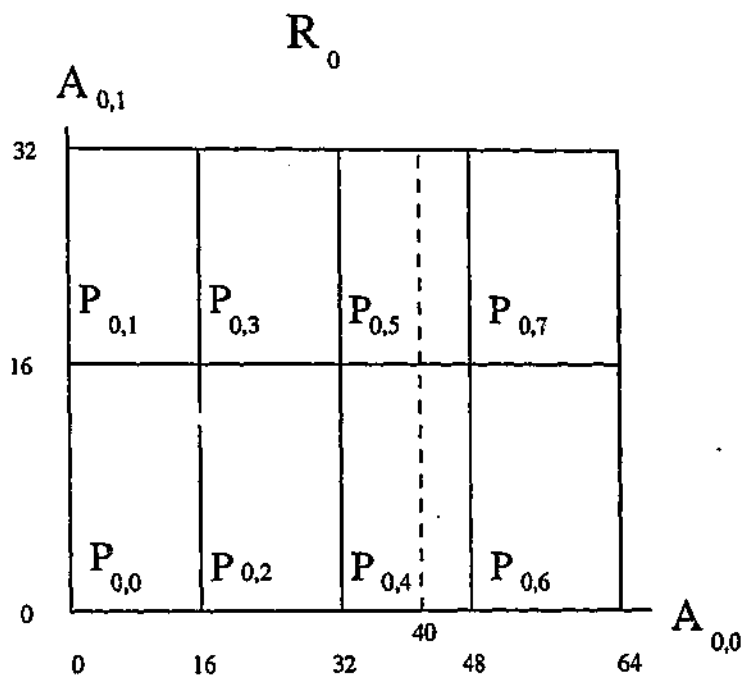


Figure 4.2:  $A_{0,0}$  is specified as 40.  $R_{0,0}$  has more intervals based on  $A_{0,0}$ .

#### 4.4 Optimising partial-match retrieval

The manner in which a BANG file is partitioned significantly affects the number of disk blocks accessed to answer a partial-match query. For example, in the above query,  $q_1$ , if  $R_0$  of Figure 4.1 was replaced by  $R_0$  of Figure 4.2 only 2 partitions ( $P_{0,4}$  and  $P_{0,5}$ ) instead of 4 will be accessed. Although the number of disk blocks in both relations is the same, fewer disk blocks are accessed when the query uses  $R_0$  of Figure 4.2. This is because the later relation has more divisions based on the specified attribute,  $A_{0,0}$ . This means, in the choice vector of  $R_0$  of Figure 4.2, more elements allocated to  $A_{0,0}$  than that of  $A_{0,1}$ .

For a given set of queries, if the queries which specify  $A_{0,i}$  are more likely than those which specify  $A_{0,j}$ , a lower average query cost is more likely to

result if more element of the corresponding choice vector are allocated to  $A_{0,i}$  than to  $A_{0,j}$ . For an arbitrary set of queries, finding optimal choice vectors, which results in the minimum average query cost, is NP-hard [94]. Hence, heuristic algorithms and partial-match query cost functions are used to find optimal or near optimal choice vectors. The heuristic algorithm used in this chapter is minimal marginal increase, MMI. MMI was extensively discussed in Section 2.6.2. The partial-match query cost functions are discussed in the next section, Section 4.5.

## 4.5 Cost functions

This section discusses the cost functions used with MMI when searching for choice vector which results in the minimal average query cost.

A relation,  $R_i$ , which has a choice vector of length  $d_i$  has a maximum of  $2^{d_i}$  partitions. This is because each element of a choice vector represents a bit position, so it can only have a value of zero or one. In Section 4.3 it was discussed that in the search string of a query, each element of an unspecified attribute is assigned a "\*". The choice vector elements which correspond to the specified attribute are assigned a zero or a one depending on the specified value. For example, if the specified value is 40 (101000 in binary) then the value of the first search index which correspond to the specified attribute is assigned a one and the second one a zero and an so on. If in the search sting, the number of elements which correspond to the specified attribute is  $d_{i,j}$ , then the number of elements which belong to the unspecified attributes (those assigned "\*") is  $d_i - d_{i,j}$ . Therefore the number

of partitions which can hold the value specified are at most  $2^{d_i - d_{i,j}}$ . Hence the cost of that query is  $2^{d_i - d_{i,j}}$ . In short, if the number search string which are assigned is  $m$  then the cost of the query is  $2^m$ .

A typical BANG file has more than one directory level. The partitions in each level span the whole domain-space. Directory levels are labeled as  $0, 1, \dots, \Gamma$ , where  $\Gamma$  is the level of the root directory and 0 is the level of the data partitions. Let the partition-level of the smallest partition at level  $k$  be denoted as  $l_k$ . Since the smallest partition at level  $k$  is bigger in size than the smallest partition at level  $k - 1$ , the number of choice vector elements used at level  $k$  will be smaller. In fact at level  $k$ , only the first  $l_k$  elements of the choice vector are used. From the first  $l_k$  elements of the choice vector, let  $d_{i,j}^k$  be the number bits which correspond to  $A_{i,j}$ . Then the cost of a partial-match query can be estimated as:

$$c_q = \sum_{k=0}^{\Gamma} \prod_{A_{i,j} \in q} 2^{d_{i,j}^k} \quad (4.1)$$

where  $q$  corresponds to the attributes with values specified in the query.

For example, consider a relation  $R_1$  which has three attributes,  $A_{1,0}$ ,  $A_{1,1}$  and  $A_{1,2}$ . Let the choice vector of  $R_1$  be:  $b_{1,1,0}$   $b_{1,0,0}$   $b_{1,2,0}$   $b_{1,0,1}$   $b_{1,1,1}$   $b_{1,0,2}$   $b_{1,2,1}$   $b_{1,0,3}$   $b_{1,2,2}$   $b_{1,0,4}$   $b_{1,1,2}$   $b_{1,2,3}$ . If a partial-match query  $q_2$  specifies (in binary)  $A_{1,0}$  and  $A_{1,2}$  as 0100110010001 and 1110010001100 respectively, the search index will be \*011 \* 01011 \* 0. Assume  $R_1$  has 3 directory levels with  $l_0$ ,  $l_1$  and  $l_2$  as 0, 4, 12 respectively. By applying Equation 4.1 to calculate the cost of performing  $q_2$ , 8, 2 and 1 partitions are retrieved from directory levels 0, 1 and 2 respectively. Therefore, the cost of  $q_2$ , which is the number of disk

accesses done to answer  $q_2$ , is 11.

Equation 4.1 differs from those described in [3, 51, 52, 83, 84, 116] because they describe the cost for multi-attribute hash files, which do not include any directory blocks. Equation 4.1 takes into account both the data and directory blocks of the BANG file, and assumes that the cost at each level is uniform. In general, this is not the case. However, at the data block level, the BANG file ensures that, on average, the density of data blocks is uniform, and our cost formula reflects this. At higher (directory) levels, this is not guaranteed. However, the number of blocks at higher levels is significantly smaller than those at the data block level, and therefore the resulting error will generally be small.

For a set of queries,  $Q$ , the average cost of a query is given by

$$Cost = \sum_{q \in Q} p_q c_q \quad (4.2)$$

where  $p_q$  is the probability of query  $q$  being asked, and  $c_q$  is the cost of query  $q$  and is given by Equation 4.1. Combining Equations 4.1 and 4.2, the average cost of a partial-match query is

$$Cost = \sum_{q \in Q} p_q \sum_{k=0}^{\Gamma} \prod_{A_{i,j} \notin q} 2^{a_{i,j}^k} \quad (4.3)$$

To ensure a minimal average query cost, elements of a choice vector be allocated using MMI and the cost function of Equation 4.3.

## 4.6 Performance Evaluation

In this section we present six sets of experimental results which show the performance of a BANG file constructed using a choice vector determined by the MMI. We refer to these choice vectors as optimized choice vectors, although choice vectors found using MMI are not guaranteed to be optimal.

The first set of results compares the performance of the optimized and cyclic choice vectors. The cyclic choice vector does not take the query distribution into account. It is a choice vector having an equal number of bits from each attribute, allocated in a cyclic order. Cyclic choice vectors are discussed in Section 3.9.1.

The second, third and fourth sets of results show the effect of the page size, file size and number of attributes on the performance of query processing using the optimized choice vector.

The fifth set of results shows whether the average query costs using optimized choice vectors are local minima. This is achieved by evaluating the cost of choice vectors which differ in 1 or 2 elements from the optimized choice vector. Let the choice vectors which differ in 1 or 2 elements be called *neighbours* in our discussion.

Query distributions change over time. Since generating an optimal choice vector every time when the query distribution is changed is an expensive operation, we don't want to change our choice vector whenever the query distribution changes slightly. A solution, based on MMI and Equation 4.3, is called *stable* if a slight change in the query distribution doesn't affect the optimality of the solution. The last set of results deals with the stability of

the optimized solution. It shows the change in the average query cost of the optimized choice vector as the query probabilities change.

#### 4.6.1 Environment

Our implementation of the BANG file includes our extension of splitting using a choice vector. We ran our experiments on a SUN SPARCstation.

For a relation containing  $n$  attributes, we generated  $2^n - 1$  partial-match queries with different randomly generated probability distributions. As the cost of the open query (the query not specifying any attributes, which retrieves all records) is independent of the choice vector, we assume that the probability of specifying the open query was zero.

Unless specified, the page size was set to be 1024 bytes, four (integer) attributes per relation were used, and each BANG file (relation) contained one million randomly generated records.

#### 4.6.2 Results

A number of experiments were performed to study the performance of the optimized choice vectors determined using MMI. The experiments were done using different query and data distributions. The data distributions used are shown in Figure 3.9.

The four query distributions used are shown in Table 4.1. They are labelled as  $\Theta_1$ ,  $\Theta_2$ ,  $\Theta_3$  and  $\Theta_4$ . In Table 4.1, the first column indicates which of the four attributes are specified in each query (a 1 if the attribute is specified

Attributes				Query distributions			
$A_{i,3}$	$A_{i,2}$	$A_{i,1}$	$A_{i,0}$	$\Theta_1$	$\Theta_2$	$\Theta_3$	$\Theta_4$
0	0	0	0	0.0	0.0	0.0	0.0
0	0	0	1	0.06413	0.02005	0.11619	0.04092
0	0	1	0	0.32996	0.00140	0.04203	0.26666
0	0	1	1	0.12622	0.05364	0.03352	0.17226
0	1	0	0	0.06384	0.00104	0.16105	0.04550
0	1	0	1	0.02950	0.03718	0.12118	0.03057
0	1	1	0	0.12566	0.00268	0.04360	0.19335
0	1	1	1	0.05201	0.09943	0.03466	0.12524
1	0	0	0	0.03208	0.00195	0.09484	0.00991
1	0	0	1	0.01803	0.07287	0.07241	0.00768
1	0	1	0	0.05738	0.00518	0.02875	0.03141
1	0	1	1	0.02718	0.19476	0.02373	0.02150
1	1	0	0	0.01790	0.00357	0.09878	0.00819
1	1	0	1	0.01282	0.13518	0.07527	0.00663
1	1	1	0	0.02709	0.00949	0.02961	0.02364
1	1	1	1	0.01618	0.36160	0.02439	0.01653

Table 4.1: Query distributions.

and a 0 if it is not), while the remaining columns show the probability of the query in each of the query distributions.

#### Optimized vs cyclic choice vectors

Tables 4.2, 4.3, 4.4 and 4.5 show the performance of the cyclic and optimized choice vectors for four query and data distributions. Column 1 of each table shows the query distribution used for each BANG file. Columns 2 and 4 show the average query cost, in disk accesses and elapsed time in milliseconds respectively, for BANG files built using the cyclic choice vector. Columns 3 and 5 show the average query cost for a BANG file built using the optimized choice vector. The final two columns show the improvement in performance

Query Distribution	Pages accessed		Time (msec)		Improvement	
	Cyclic	Optimized	Cyclic	Optimized	Pages	Time
$\Theta_1$	1444	888	13453	7821	1.63	1.72
$\Theta_2$	163	70	1569	638	2.33	2.46
$\Theta_3$	1138	1004	10491	9013	1.13	1.16
$\Theta_4$	1150	537	10518	4472	2.14	2.35

Table 4.2: Average query costs for a uniform data distribution.

Query Distribution	Pages accessed		Time (msec)		Improvement	
	Cyclic	Optimized	Cyclic	Optimized	Pages	Time
$\Theta_1$	692	445	5944	3628	1.56	1.64
$\Theta_2$	83	37	722	313	2.24	2.30
$\Theta_3$	554	499	4679	4187	1.11	1.12
$\Theta_4$	548	300	4799	2573	1.83	1.87

Table 4.3: Average query costs for a clustered data distribution.

achieved by using the optimized choice vector instead of the cyclic choice vector to build the BANG file.

The data distributions used to build the BANG files in Tables 4.2, 4.3, 4.4 and 4.5 were uniform, clustered, sinusoidal and linear (see Figure 3.9), respectively.

The results show that the average query cost can be significantly reduced by using the optimized choice vector in the construction of the BANG file compared with the cyclic choice vector. The largest improvement that we observed for these distributions with four attributes was a factor of 2.33 in the number of disk accesses and 2.46 in elapsed time. In Section 4.6.2 we will show an improvement by a factor of 6.41 for a distribution with eight attributes.



Query Distribution	Pages accessed		Time (msec)		Improvement	
	Cyclic	Optimized	Cyclic	Optimized	Pages	Time
$\Theta_1$	225	161	2003	1364	1.40	1.47
$\Theta_2$	26	15	216	95	1.74	2.27
$\Theta_3$	187	168	1709	1540	1.11	1.11
$\Theta_4$	175	113	1616	1019	1.55	1.59

Table 4.4: Average query costs for a sinusoidal data distribution.

Query Distribution	Pages accessed		Time (msec)		Improvement	
	Cyclic	Optimized	Cyclic	Optimized	Pages	Time
$\Theta_1$	207	148	2035	1324	1.40	1.54
$\Theta_2$	22	14	180	90	1.61	1.99
$\Theta_3$	170	155	1575	1384	1.09	1.14
$\Theta_4$	160	104	1491	883	1.53	1.69

Table 4.5: Average query costs for a linear data distribution.

### Page size

We performed experiments to study the effect of the page size on the performance of the optimized and cyclic choice vectors. Page sizes of 1, 2, 4, 8, 16, 32 and 64 kbytes were used. The experiments were repeated using the uniform, clustered, sinusoidal and linear data distributions.

We found that an improvement in performance was achieved by using the optimized instead of the cyclic choice vector for all page sizes, as shown in Figure 4.3.

To study the effect of page size on the CPU time we performed experiments using different data and query distributions. The results, shown in Figure 4.4, indicate that the optimal page size is between 4 and 8 kbytes for the configuration that we tested.

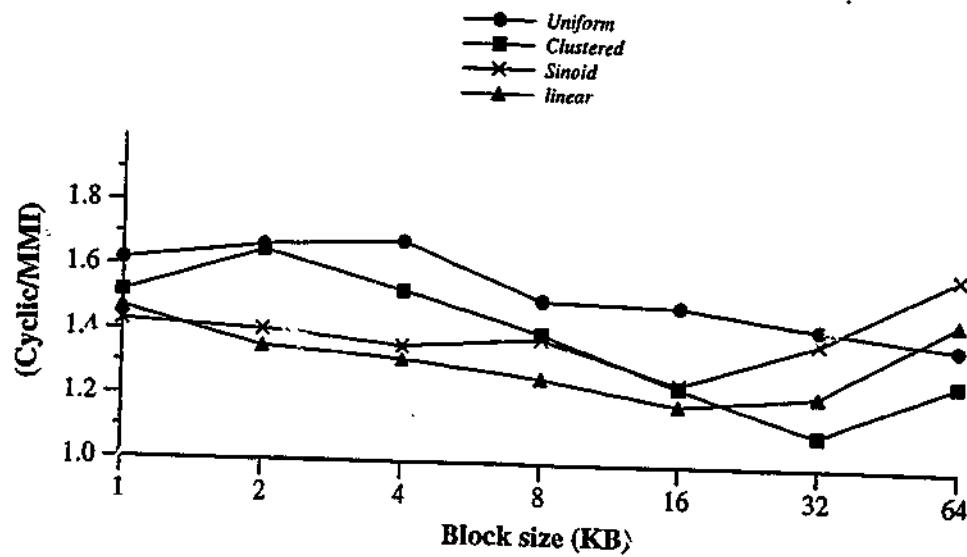


Figure 4.3: Effect of the page size on performance.

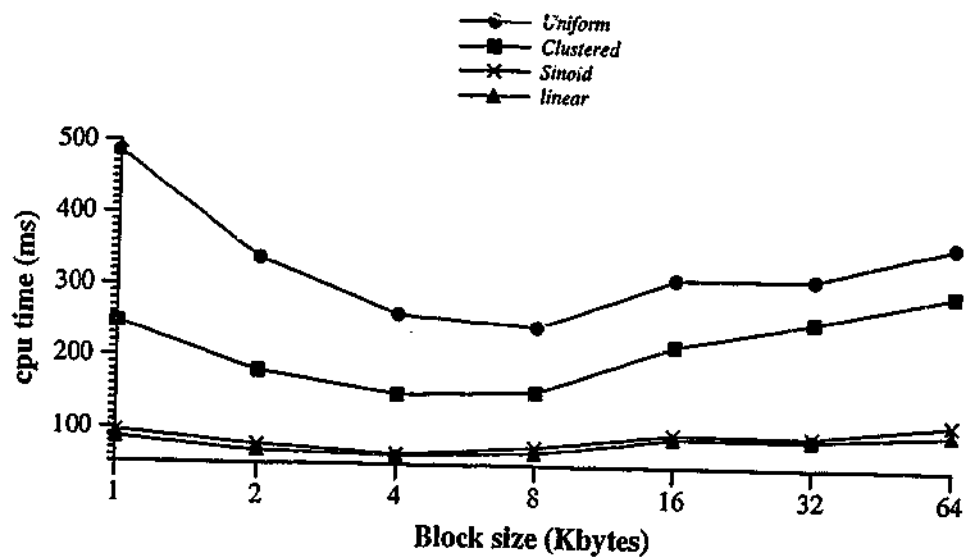


Figure 4.4: Effect of page size on CPU time.

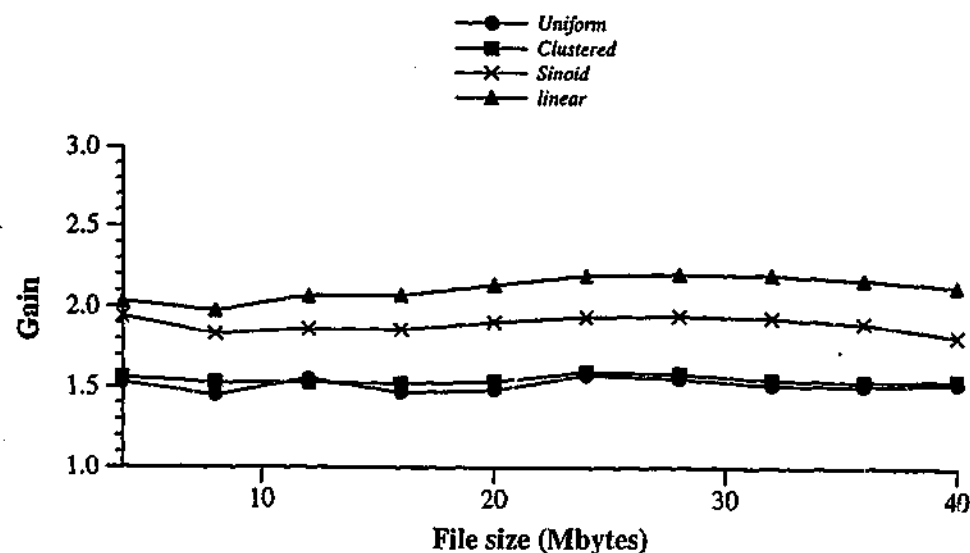


Figure 4.5: Effect of file size on performance.

#### File size

A number of experiments were performed to study the effect of the file size on the performance of the optimized and cyclic choice vectors. File sizes ranging from 4 Mbytes to 40 Mbytes were used. Again, the experiments were repeated using the uniform, clustered, sinusoidal and linear data distributions.

The experimental results are shown in Figure 4.5. They indicate that the improvement in performance gained by using the optimized instead of the cyclic choice vector was reasonably consistent for all file sizes for a given data distribution.

#### Number of attributes

To investigate how the number of attributes affect the performance of the optimized choice vector we performed experiments on BANG files with different numbers of attributes. Table 4.6 shows results when BANG files of 2, 3, 4 and

Number of attributes	Cyclic cost	Optimised cost	Improvement
2	88	82	1.07
2	47	23	2.04
2	108	95	1.14
2	75	64	1.17
3	383	318	1.20
3	134	45	3.0
3	441	374	1.18
3	276	218	1.27
4	1442	877	1.64
4	165	68	2.43
4	1151	1023	1.13
4	1148	531	2.16
8	2285	1246	1.83
8	263	41	6.41
8	4453	2661	1.67
8	2142	639	3.35

Table 4.6: Effect of the number of attributes on the average query cost.

8 attributes are used. Each of these BANG files were built using a uniform data distribution. Each row in a table corresponds to a different query distribution. Column 1 shows the number of attributes per record. Columns 2 and 3 show the average number of pages accessed to answer queries using a BANG file built using the cyclic and optimized choice vectors, respectively. The final column shows the improvement in the average query cost if the BANG file was built using the optimized choice vector instead of the cyclic choice vector.

If a query distribution is non-uniform then, in general, we expect that the improvement in performance achieved by using an optimized choice vector instead of the cyclic choice vector will increase with the increase in the number of attributes. This is because with more attributes there is a greater

chance of allocating a larger proportion of the indexing bits to unimportant attributes using the cyclic choice vector. The results shown in Table 4.6 support this hypothesis.

### Local minima

As we mentioned in Section 4.5, for an arbitrary query probability distribution, finding the optimal choice vector is NP-hard [94], so our approach is not guaranteed to find the optimal choice vector. To determine whether we produce good choice vectors, we ran a series of experiments to compare the average query cost using the optimized choice vector with the average query cost using neighbouring choice vectors.

We created four sets of neighbouring choice vectors by changing some of the bits in the choice vector obtained using MMI. The first set of choice vectors were created by swapping a single pair of bits in positions  $m$  and  $m + 1$ , where  $m$  is an even position. The second set was created swapping a single pair of bits in positions  $m$  and  $m + 2$ . The third set was created by changing the allocation of a single bit position from one attribute to each of the others. The final set was created by rotating the positions of the first 12 bits.

For a BANG file of 4 attributes and a choice vector of 30 elements, the number of neighbouring choice vectors generated in each of the four sets was 29, 28, 90 and 12 respectively, totalling 159 choice vectors. The average query cost using each of these choice vectors was found experimentally.

We found that when the data distribution is uniform, the average query cost using the optimized choice vector is either a local minima or nearly a

local minima. For non-uniform data distributions, the average query cost using the optimized choice vector was never more than 2.5% greater than that of any of the 159 neighbouring choice vectors.

### Stability

Query patterns can change over time. A BANG file which is optimal for a given query pattern may not remain optimal if the query pattern changes.

In order to study the stability of the optimized choice vectors, we performed a series of experiments to determine the change in performance when the query distribution changes. These experiments were similar to those performed in [53] and were performed using different data and query distributions.

We define a change of  $x\%$  in a query distribution to be the result of randomly changing each query probability,  $p$ , to be in the range  $p \times (1 \pm \frac{x}{100})$  prior to the whole query distribution being normalized.

Figures 4.6 to 4.21 show how the average query cost is affected when the probability of each query changes by up to 80%. The horizontal axis of each figure denotes the percentage change in the query distribution. The vertical axis denotes the ratio of the average query cost of one choice vector with the average query cost of a choice vector optimized for this changed query distribution.

In each figure, three average query cost ratios are shown, using dotted, dashed and solid lines. The dotted line ("Cyclic/New") corresponds to a BANG file built using a cyclic choice vector. The dashed line ("Old/New") corresponds to a BANG file which was built using the optimized choice vector

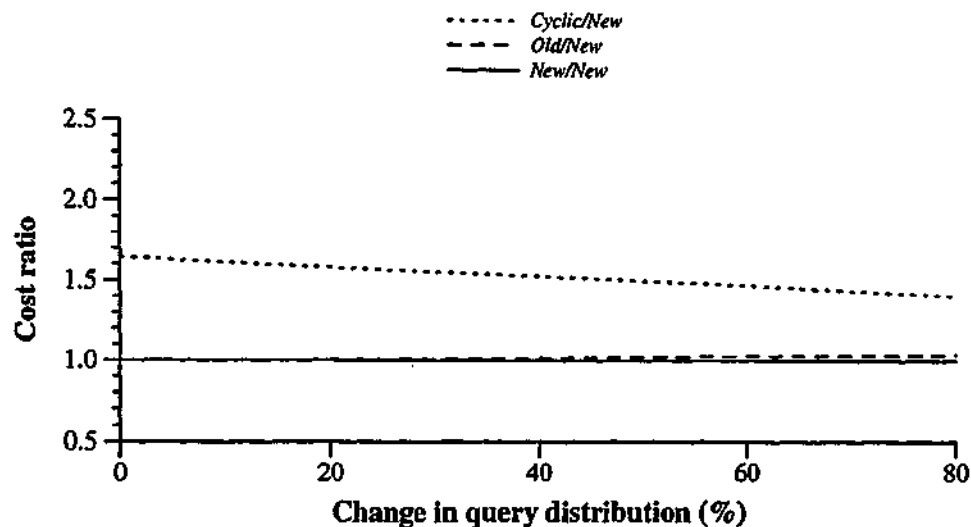


Figure 4.6: Stability of optimized solution. Query distribution =  $\Theta_1$ , data distribution = uniform.

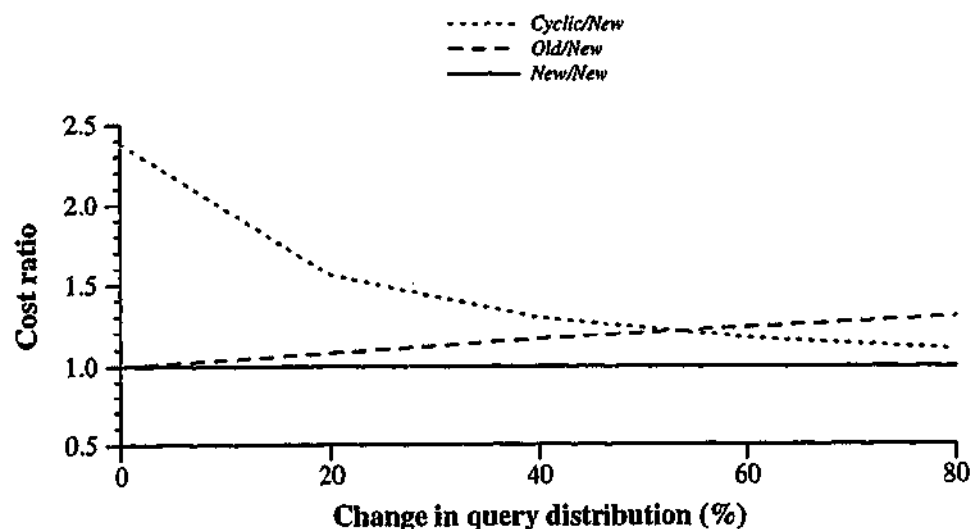


Figure 4.7: Stability of optimized solution. Query distribution =  $\Theta_2$ , data distribution = uniform.

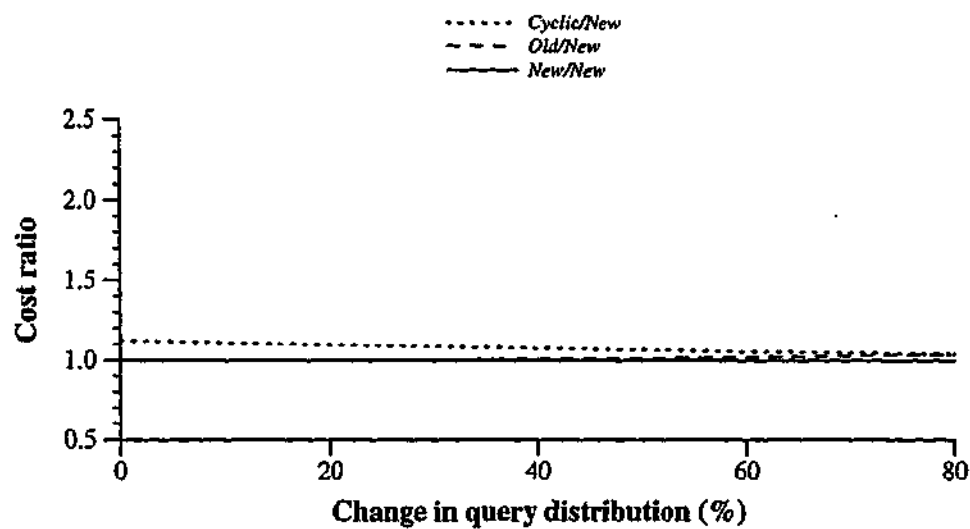


Figure 4.8: Stability of optimized solution. Query distribution =  $\Theta_3$ , data distribution = uniform.

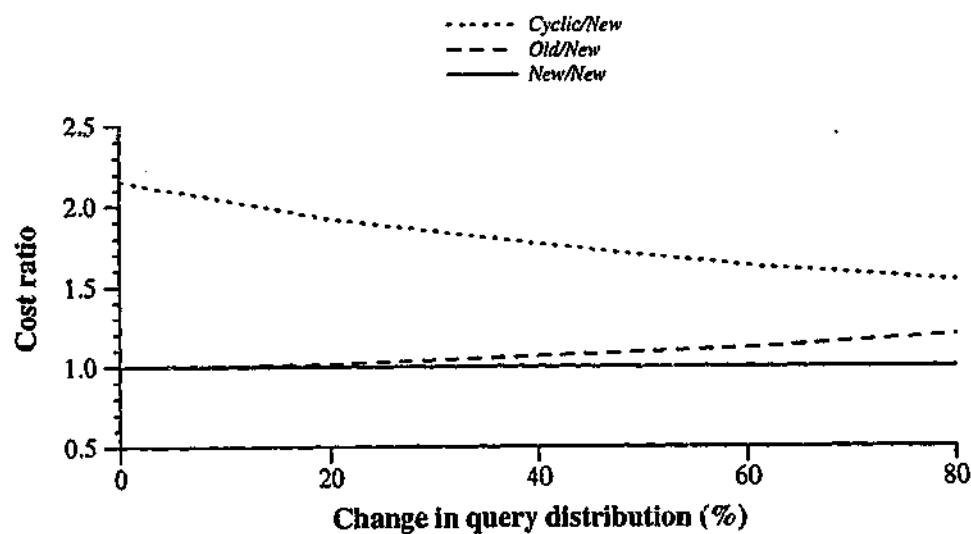


Figure 4.9: Stability of optimized solution. Query distribution =  $\Theta_4$ , data distribution = uniform.



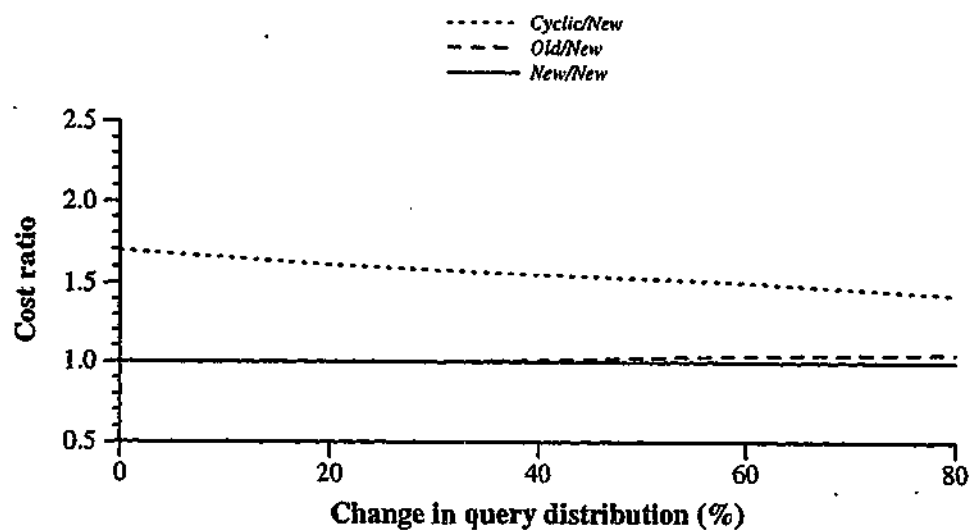


Figure 4.10: Stability of optimized solution. Query distribution =  $\Theta_1$ , data distribution = clustered.

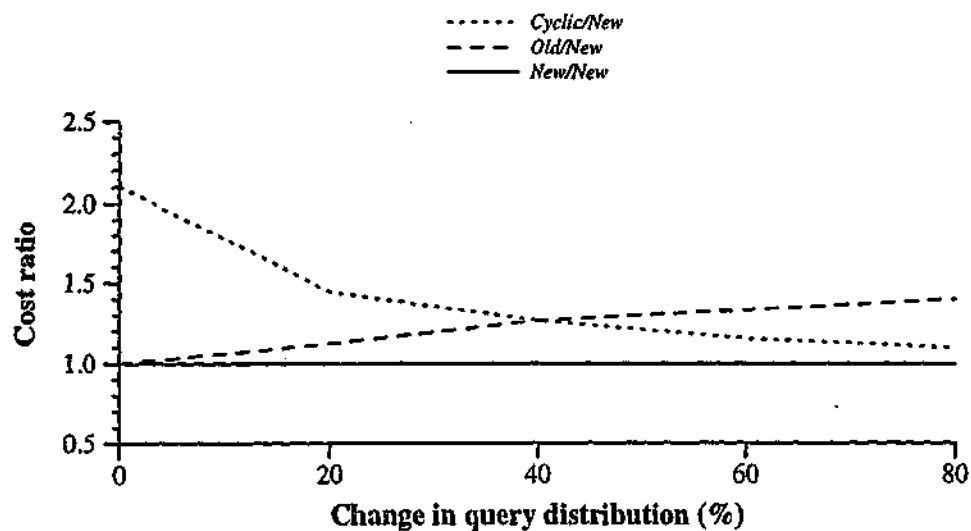


Figure 4.11: Stability of optimized solution. Query distribution =  $\Theta_2$ , data distribution = clustered.

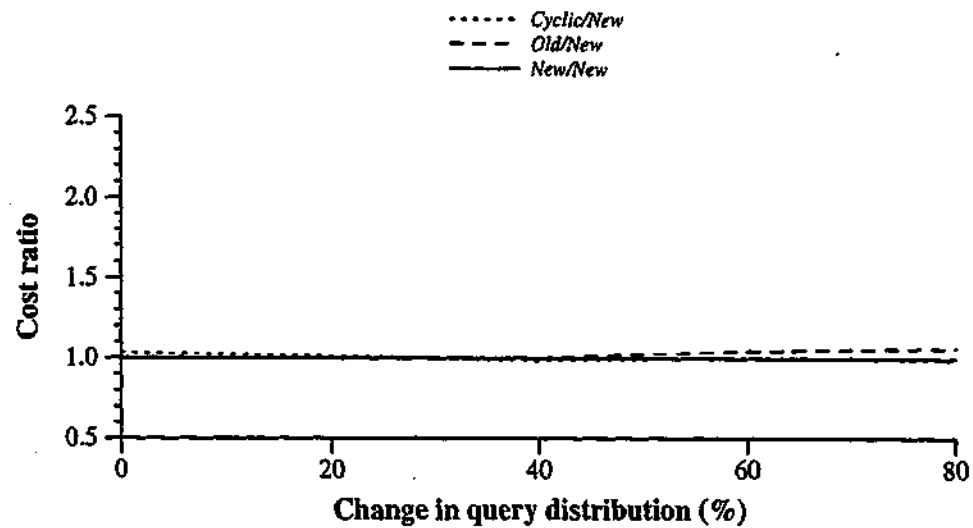


Figure 4.12: Stability of optimized solution. Query distribution =  $\Theta_3$ , data distribution = clustered.

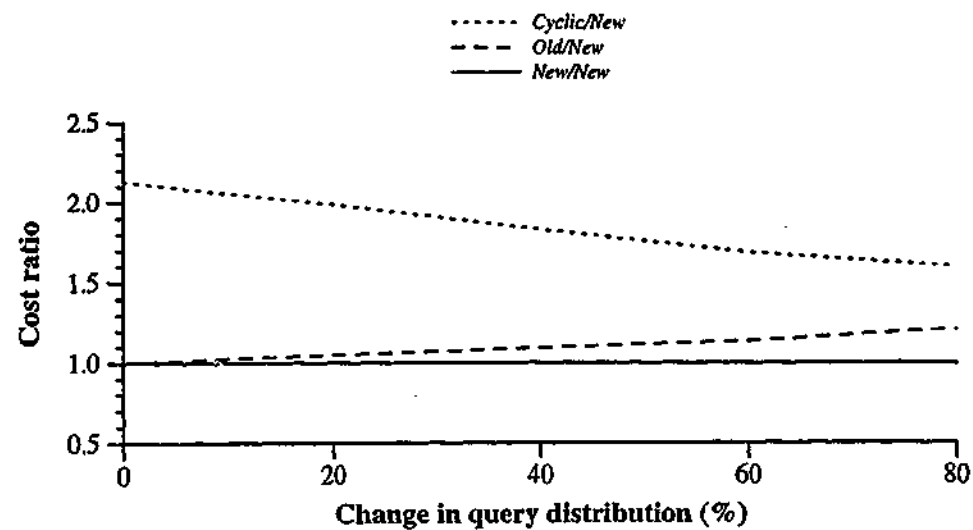


Figure 4.13: Stability of optimized solution. Query distribution =  $\Theta_4$ , data distribution = clustered.

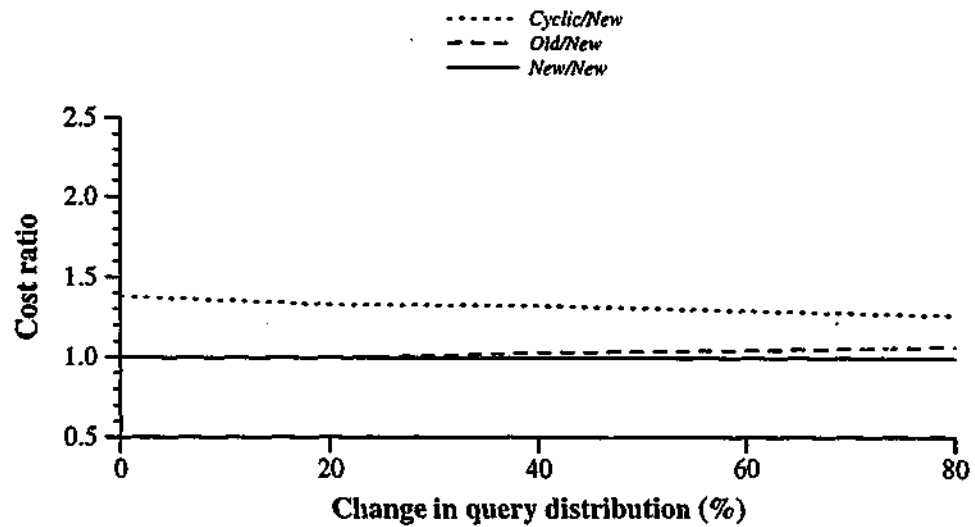


Figure 4.14: Stability of optimized solution. Query distribution =  $\Theta_1$ , data distribution = sinusoidal.

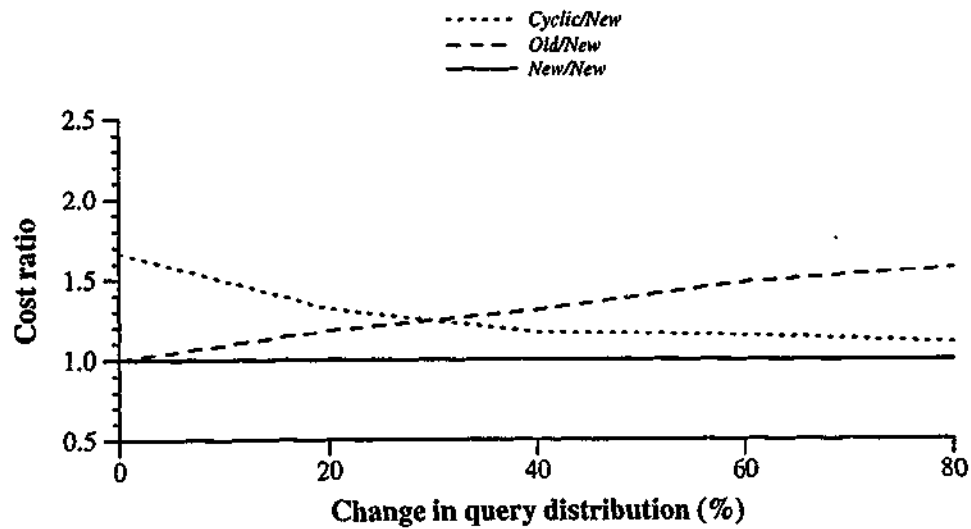


Figure 4.15: Stability of optimized solution. Query distribution =  $\Theta_2$ , data distribution = sinusoidal.

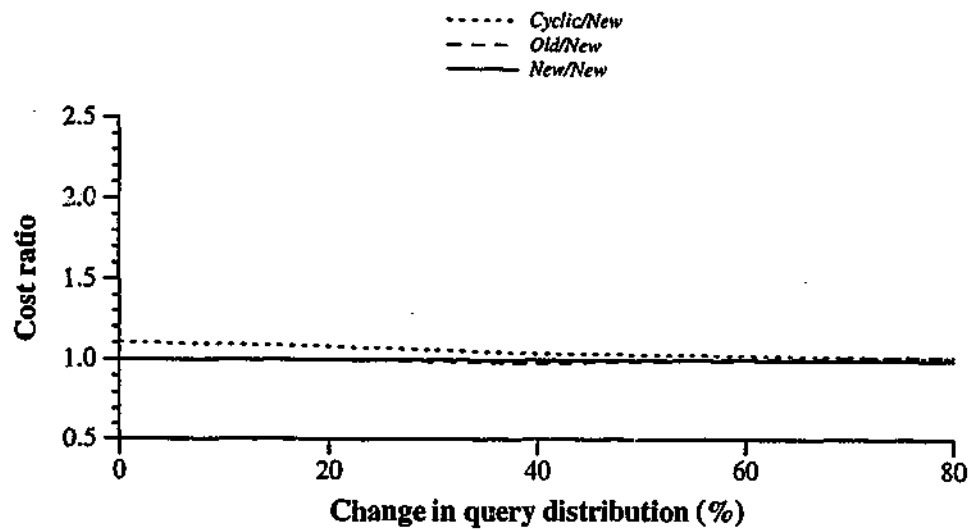


Figure 4.16: Stability of optimized solution. Query distribution =  $\Theta_3$ , data distribution = sinusoidal.

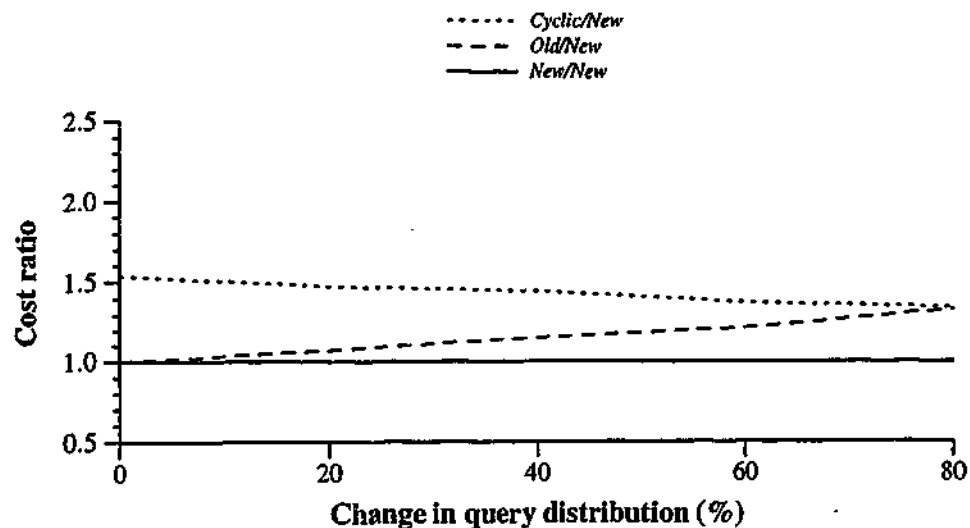


Figure 4.17: Stability of optimized solution. Query distribution =  $\Theta_4$ , data distribution = sinusoidal.

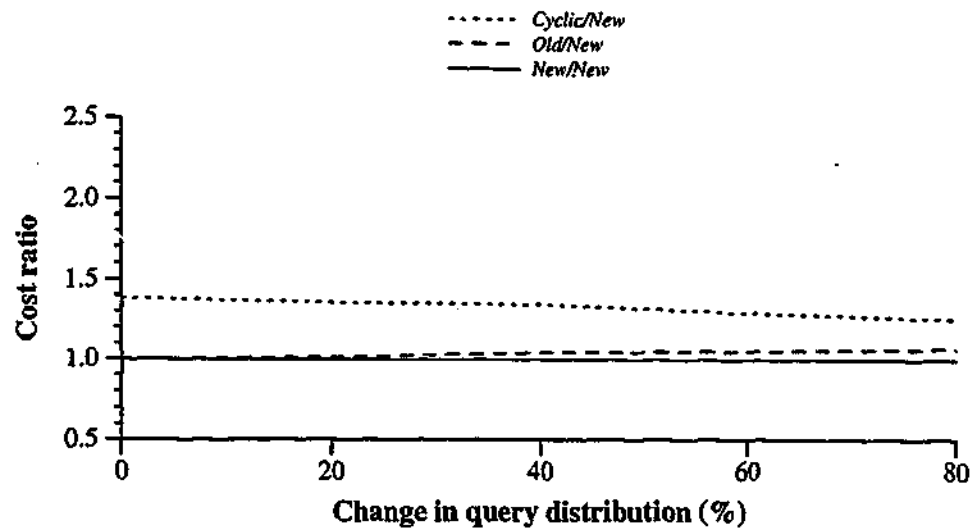


Figure 4.18: Stability of optimized solution. Query distribution =  $\Theta_1$ , data distribution = linear.

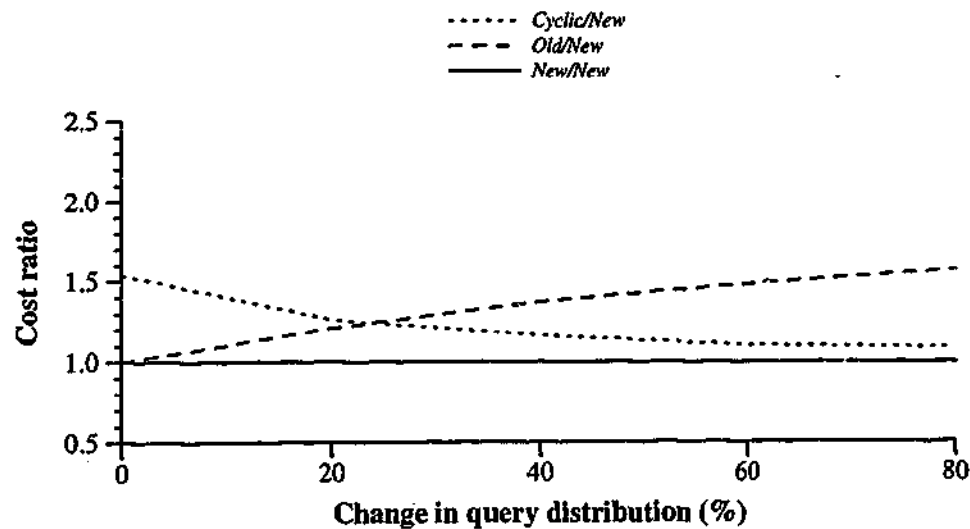


Figure 4.19: Stability of optimized solution. Query distribution =  $\Theta_2$ , data distribution = linear.

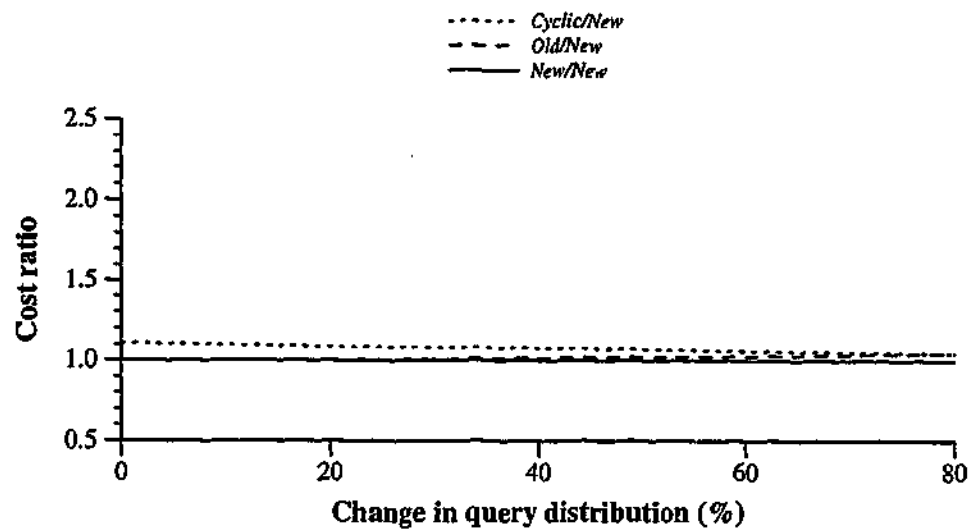


Figure 4.20: Stability of optimized solution. Query distribution =  $\Theta_3$ , data distribution = linear.

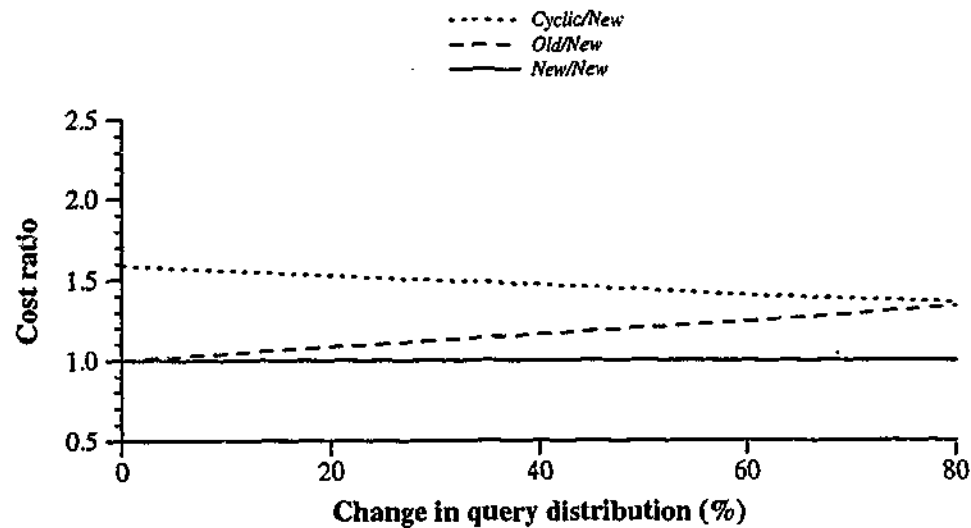


Figure 4.21: Stability of optimized solution. Query distribution =  $\Theta_4$ , data distribution = linear.

determined using the original query distribution. The solid line ("New/New") corresponds to a BANG file built using an optimized choice vector determined using the new (changed) probability distribution.

Our results show that when a query probability distribution changes by 20%, the degradation in performance of a BANG file built using an original optimized choice vector is always better than that of the cyclic choice vector. On many occasions the degradation is less than 20% even if there is an 80% change to the query distribution. As a result, we can conclude that we need only reorganise a BANG file to use a new optimized choice vector when the query distribution changes substantially (by at least 20%).

## 4.7 Conclusion

This chapter discusses in detail a new approach of optimising partial-match queries for multidimensional file structures. Unlike previous similar work using multi-attribute hash files, the approach proposed does not assume a uniform data distribution, instead it uses file structure which evenly distributes records amongst disk pages, even when the data distribution is skewed. The proposed strategy takes the query distribution into account and finds optimised choice vectors which result in an average query cost which is significantly less than that of the strategy which assumes that all attributes are of equal importance.

To determine the performance of the proposed approach we ran an extensive series of experiments. These experiments were conducted using a range of different data and query distributions. Our experiments show that an op-

timized choice vector results in a significant reduction in the average query cost compared with alternative (cyclic choice vector) policy which do not take the query distribution into account. In one case, for a relation with 8 attributes, we observed an improvement in performance by a factor of 6.4. In general, the improvement in performance was greater for relations containing a larger number of attributes.

Finding an optimal choice vector is NP-hard. However, the experiments indicate that the optimized choice vectors found using MMI and Equation 4.3 are either local minima or very close to the local minima. Further more, the stability of the optimized choice vector is excellent. A change in the query distribution of up to 20% has a minimal impact on the performance (less than 5%) when using an optimized choice vector found for the original query distribution. Often the degradation is not significant (less than 20%) even when the original query distribution is changed by 80%.



## Chapter 5

# Optimizing Range Query Retrieval

### 5.1 Introduction

Range queries are a common database operation. In a range query, a range of values is specified for one or more attributes of a relation. The result of a range query is the retrieval of all the records with values within the specified ranges.

The cost of a range query is mainly measured in the number of disk blocks accessed to answer the query. The number of disk blocks retrieved depends on the algorithm used to place the records within the file. The average number of blocks accessed per query can be minimised if an efficient record clustering algorithm, which takes the query distribution into account, is used.

The aim of this chapter is to describe a method of clustering related records into fewer disk blocks such that the average number of disk blocks

accessed, over all expected range queries, is minimised. The method is applicable to multidimensional file structures and the query distribution is known in advance.

Several methods have been proposed to optimally cluster records for range queries [17, 52, 79], but all of them were limited to uniform data distribution. The methods described in this chapter avoid this limitation, by using multidimensional file organisations which distribute records evenly even if the data distribution is non-uniform. Although the method is tested using the BANG file [36, 37], it can be used with a number of other multidimensional file structures such as the nested interpolation based grid file [110], the multi-level grid file [140] and other similar file structures [86, 98].

This chapter has six sections. Section 5.2 discusses range queries in general. A way of minimising range query cost is discussed in section 5.3. Section 5.4 discusses cost functions related to range queries. The results of our experiments are presented in section 5.5 and section 5.6 is the conclusion.

## 5.2 Range Queries

As is mentioned in the previous section, in a range query, a range of values is specified for one or more attributes of a relation. The result of a range query is the retrieval of all the records with values within the specified ranges. The following query,  $q_0$ , is an example of a range query:

```
SELECT  $A_{0,0}$ 
FROM  $R_0$ 
WHERE  $A_{0,1}$  BETWEEN 35 AND 48
```

( $q_0$ )

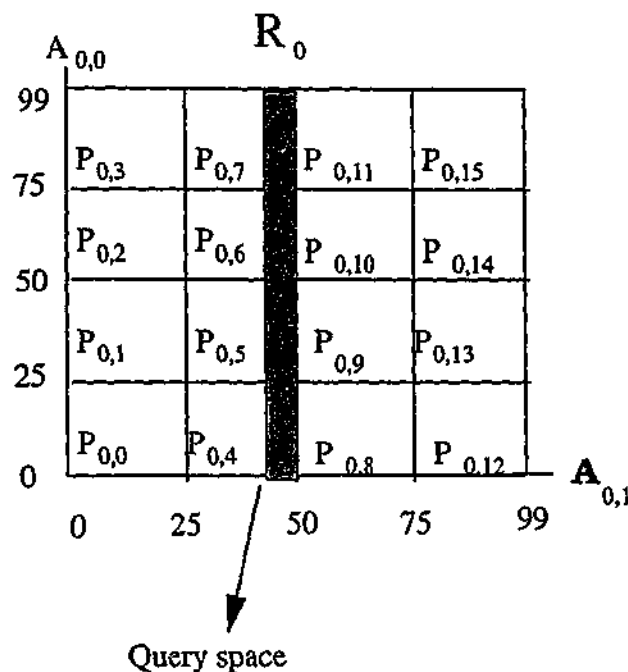


Figure 5.1: Query-space intersecting four partitions.

Where  $R_0$  is the BANG file representative of the relation as shown Figure 5.1.

A range query defines a subspace within the domain-space of a relation. We call such a subspace a *query-space*. For example, the shaded area in Figure 5.1 is the query-space of  $q_0$ .

The records which satisfies a range query can be found in the partitions that overlap with the query-space. We call a record which satisfies a query a  $\Phi$ -record of the query, and its corresponding partition a  $\Phi$ -partition. For example, the  $\Phi$ -partitions of  $q_0$  in Figure 5.1 are  $P_{0,4}$ ,  $P_{0,5}$ ,  $P_{0,6}$ , and  $P_{0,7}$ .

If a  $\Phi$ -partition is not totally enclosed by the query space, some of its records may lie outside the query-space and so are not  $\Phi$ -records. For example, a record with a value of 30 for  $A_{0,0}$  and a value of 40 for  $A_{0,1}$  is not a

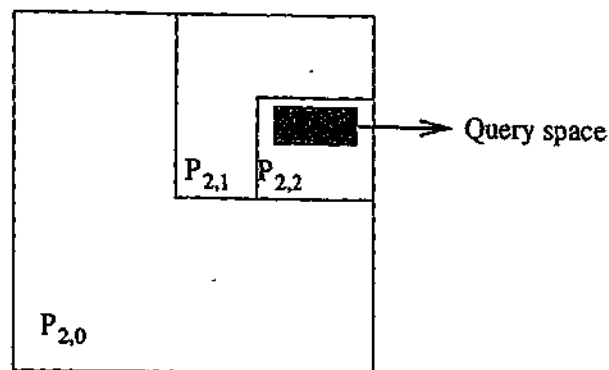


Figure 5.2:  $P_{2,2}$  is a  $\Phi$ -partition but  $P_{2,0}$  and  $P_{2,1}$  are not

$\Phi$ -record even though it is found in partition  $P_{0,5}$ , which is a  $\Phi$ -partition of  $q_0$ , Figure 5.1.

If there are two or more partitions which totally enclose a query-space, only the partition which directly encloses the query-space is the  $\Phi$ -partition. The others will not contain  $\Phi$ -records, so there is no need to retrieve them. For example, in Figure 5.2 all three partitions,  $P_{2,0}$ ,  $P_{2,1}$  and  $P_{2,2}$  enclose the query space but only  $P_{2,2}$  directly encloses it, so it is the only  $\Phi$ -partition.

In a BANG file the search for the  $\Phi$ -partitions starts from the root partition. Once the  $\Phi$ -partitions in the root are identified, the search for the  $\Phi$ -partitions descends to the next lower level directory using the entries of the  $\Phi$ -partitions identified so far. These process is repeated until all the  $\Phi$ -partitions are identified. Then the  $\Phi$ -records are searched from the  $\Phi$ -partition which contain data records.

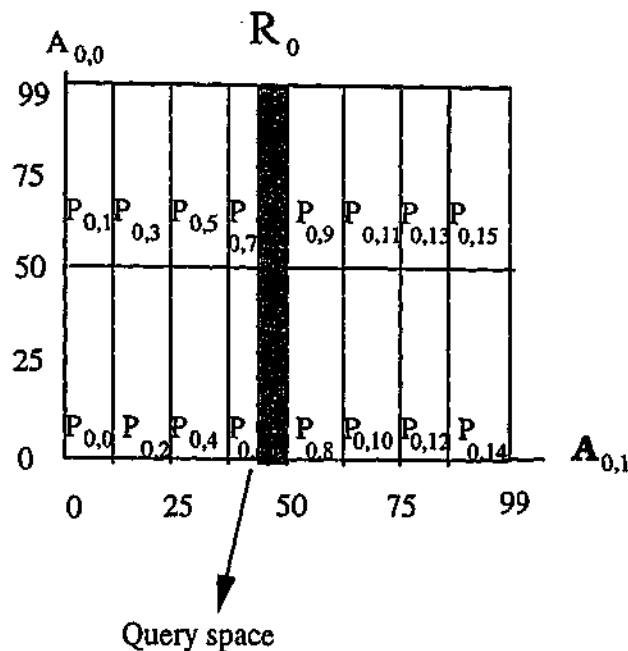


Figure 5.3: Query-space intersecting four partitions.

### 5.3 Minimising range query costs

The cost of a query depends on the number of disk accesses performed to answer the query. The cost of a query can be reduced if its  $\Phi$ -records are clustered in fewer partitions (disk pages). In other words the cost of a query is minimised if the number of  $\Phi$ -partitions is minimised.

The search for  $\Phi$ -records of a query is performed using the values of the attributes specified in the WHERE clause of the query. For example, in searching for the  $\Phi$ -records of  $q_0$ , we use the values of  $A_{0,0}$ , because  $A_{0,0}$  is the specified attribute in the WHERE clause of  $q_0$ . We call the attributes that we use to search for the  $\Phi$ -records the *significant attributes*.

A domain-space which is partitioned to a greater degree on the most significant attributes will potentially cost less than the one that is partitioned

on the less significant attributes or one which treats all attributes as equally significant. For example,  $q_0$  has four  $\Phi$ -partitions if used with  $R_0$  of Figure 5.1, but only two  $\Phi$ -partitions if used with  $R_1$  of Figure 5.3. This is because the domain-space in Figure 5.3 is partitioned to a greater degree on  $A_{0,0}$ , which is the most significant attribute, compared with Figure 5.1 in which the domain-space is partitioned using a cyclic choice vector.

Figures 5.1 and 5.3 show how the partitioning order affects the cost of one query in a two dimensional domain-space. The problem we are addressing here is that given an  $n$  dimensional domain-space, an arbitrary number of queries and their probabilities, can we construct the choice vector which results in the minimal average query cost. For an arbitrary set of queries, finding the optimal choice vector is NP-hard [94]. As a result, we use an heuristic technique, *minimal marginal increase* (MMI), together with the cost functions presented in section 5.4 to minimise the average query cost. MMI, which is described in section 2.6.2, is a greedy algorithm and choice vectors obtained using MMI do not guarantee optimality, although they have often been optimal or near-optimal for other problems in the past [54].

## 5.4 Cost functions

We define the cost of a query to be the number of disk pages accessed to answer the query. For the set of all queries  $Q$ , the average cost of a query is given by

$$C_{avg} = \sum_{q \in Q} p_q C(q), \quad (5.1)$$

where  $p_q$  is the probability of  $q$  being asked, and  $C(q)$  is the cost of answering the query  $q$ .

For range queries, the average cost of a query can be approximated as follows. Assume that the number of data blocks of a BANG file is  $2^d$ , where  $d$  is equal to the length of the choice vector, which is also the highest partition-level in the domain-space. Assume also that attribute  $A_{i,j}$  appears  $\bar{d}_{i,j}$  times in the choice vector. Let  $n$  be the number of attributes, then we have

$$\sum_{i=0}^{n-1} \bar{d}_{i,j} = d. \quad (5.2)$$

Let  $r_{i,j}(q)$  be the proportion of the total range of  $A_{i,j}$  that query  $q$  specifies. For example, if the domain of  $A_{i,j}$  is  $[1, 100]$ , and a query,  $q$ , specifies the range  $[2, 7]$  then

$$r_{i,j}(q) = (7 - 2 + 1)/100 = 0.06. \quad (5.3)$$

We assume that the average number of data pages accessed in answering  $q$  can be approximated by

$$C^0(q) = \prod_{i=0}^{n-1} [r_{i,j}(q) 2^{\bar{d}_{i,j}}]. \quad (5.4)$$

As in the previous chapter let  $l_h$  the partition-level of the smallest partition at directory level  $h$ . Out of the  $l_h$  choice vector elements at level  $h$  let  $\bar{d}_{i,j}^h$  belong to attribute  $A_{i,j}$ . Then the average number of directory level  $h$

pages accessed in answering  $q$  can be approximated by

$$C^h(q) = \prod_{i=0}^{n-1} [r_{i,j}(q) 2^{d_{i,j}^h}]. \quad (5.5)$$

Combining Equations 5.4 and 5.5, if there are  $\Gamma$  directory levels in addition to the data pages, the average cost of  $q$  can be approximated by

$$C(q) = \sum_{h=0}^{\Gamma} \prod_{i=0}^{n-1} [r_{i,j}(q) 2^{d_{i,j}^h}]. \quad (5.6)$$

Combining Equations 5.1 and 5.6, the average cost of the set of all queries  $Q$  can be approximated by

$$C_{avg} = \sum_{q \in Q} p_q \sum_{h=0}^{\Gamma} \prod_{i=0}^{n-1} [r_{i,j}(q) 2^{d_{i,j}^h}]. \quad (5.7)$$

Previously, there was no good model available to estimate a query cost in a BANG file. Experimental studies [36] show that the BANG file evenly distributes data records among disk blocks even when the data distribution is highly non-uniform, which makes Equation 5.7 a reasonable approximation of the cost.

## 5.5 Experimental Results

In this section we present the results of experiments comparing the performance of the optimised and cyclic choice vectors.

The first set of results shows the performance of the optimised and the cyclic choice vectors on different data and query distributions. The second



set of results shows the effect of the number of attributes on the performance of both choice vectors. The third and fourth sets show the effect of the file size and page size on the performance of the cyclic and optimised choice vectors.

Query distributions change over time. A choice vector optimised for one query distribution may not perform as well if the query distribution changes. A solution, based on MMI and Equation 5.7, is called *stable* if a slight change in the query distribution doesn't affect the optimality of the solution. The sixth set of results demonstrate the stability of the optimised choice vector.

The final set of results show how the performance of the optimised choice vector is affected as the size of query-space changes. The size of a query space is the size of all the partitions (subspaces) which overlap the query in the domain-space.

### 5.5.1 Environment

We implemented the BANG file with our extension (discussed in section 3.10) of using choice vectors during partition splitting. In each experiment we used 16 randomly generated queries and assigned each of them a randomly generated probability. Unless specified, we used a page size of 1024 bytes, four integer attributes per record and one million randomly generated records per relation (BANG file). We ran all our experiments on a SPARC station 20.

The data distributions used were uniform, clustered regions, a linear correlation, and a non-linear correlation function (a sine wave). Examples of

Attributes				Query distribution
$A_{i,0}$	$A_{i,1}$	$A_{i,2}$	$A_{i,3}$	
506	366	1005533	377939	0.064619
2334	64	827174	941936	0.064450
3694	1386	847200	666416	0.064505
2854	2514	680605	119300	0.064466
161	2667	249084	39314	0.064374
2147	1780	948659	308398	0.064613
773	305	390266	1002868	0.064647
833	452	417366	232725	0.064434
441	1168	616981	449444	0.064586
447	10	274510	15654	0.064350
1283	1813	73602	25405	0.064309
87	319	885838	524852	0.064591
8093	219	104153	18256	0.032452
337	3519	826968	399079	0.064361
3478	1570	131805	852635	0.064595
26	1957	53432	652329	0.064648

Table 5.1: Query distribution  $\Theta_1$ .

these are shown in Figure 3.9. We refer to them as uniform, clustered, linear and sinusoidal, respectively.

In the experiments in which a relation of four attributes was involved, up to four different query distributions were used. These query distributions were generated randomly and are referred to as  $\Theta_1$ ,  $\Theta_2$ ,  $\Theta_3$  and  $\Theta_4$ . They are shown in Tables 5.1 to 5.4. Each entry in one of the first four columns represents a range specified for the corresponding attribute,  $A_{i,0}$ ,  $A_{i,1}$ ,  $A_{i,2}$  or  $A_{i,3}$ . The domain of each attribute is between 0 and 1048575 ( $2^{20} - 1$ ). The first four columns of each row represents the query and the last column is its probability.

Attributes				Query distribution
$A_{i,0}$	$A_{i,1}$	$A_{i,2}$	$A_{i,3}$	
1759	2041	635273	1025206	0.076838
42817	287843	308082	12032	0.000554
81	1779	334939	353540	0.076743
1076	217	853330	499153	0.076803
1964	467	491378	594646	0.076848
2322	133052	258460	581514	0.038611
362957	336	138907	527880	0.038717
1	3532	696988	512476	0.076695
621	1636	814401	832879	0.076971
661341	2058	758929	1028152	0.038691
204	28	739366	869555	0.077001
15434	92	190968	338957	0.038494
2575	2259	308333	835018	0.076760
751	1841	1007087	577772	0.076824
426	913	458917	766551	0.076750
1656	1893	21487	127925	0.076700

Table 5.2: Query distribution  $\Theta_2$ .

Attributes				query distribution
$A_{i,0}$	$A_{i,1}$	$A_{i,2}$	$A_{i,3}$	
7385	284251	505581	498772	0.000853
1188	818	572743	456734	0.099394
295974	230166	237306	964901	0.000543
33435	160798	822957	454672	0.000675
136513	4163	280042	480255	0.050236
3495	43	32554	644906	0.099919
60813	20	1004336	69033	0.050140
825	977	58904	397042	0.099485
338	850	457631	282380	0.099446
56	741	23639	476335	0.099390
168742	473	595126	320760	0.050392
1163	1206	483057	798659	0.099902
495	56647	796117	688349	0.050078
18790	91289	849767	896630	0.000716
2030	510	31842	346635	0.099328
298	676	152878	215281	0.099504

Table 5.3: Query distribution  $\Theta_3$ .

Attributes				Query distribution
$A_{i,0}$	$A_{i,1}$	$A_{i,2}$	$A_{i,3}$	
2403	1528	213373	306645	0.071582
772	2052	577372	392585	0.071303
2825	422601	607591	406208	0.035911
5053	1467	589654	547291	0.035704
26	811	334025	902371	0.071297
1823	2244	203907	697994	0.071349
515	83	501258	858766	0.071391
119228	1848	595616	934196	0.035824
2464	503	588547	888557	0.071551
85	378	27409	453937	0.071253
375	1645	73284	565287	0.071268
2313	152687	716835	122699	0.036023
283	543	600556	267813	0.071475
1356	173	752214	319236	0.071362
1800	2602	412213	639519	0.071354
456	1005	497024	775743	0.071292

Table 5.4: Query distribution  $\Theta_4$ .

### 5.5.2 Effect of data and query distributions

The effect of using the optimised and cyclic choice vectors on the average query cost using different data and query distributions is shown in Tables 5.5 to 5.8. The first column in each of these tables shows the query distribution used. The second and the fourth columns correspond to the cyclic choice vector and show the cost in disk page accesses and time taken, respectively. Similarly, the third and fifth columns show the costs corresponding to the optimised choice vector. The improvement in the number of disk page accesses and time taken when using the optimised choice vector rather than the cyclic choice vector is shown in the final two columns.

Query Distribution	Pages accessed		Time (msec)		Improvement in	
	Cyclic	Optimised	Cyclic	Optimised	Pages accessed	Time
$\Theta_1$	77.06	7.32	624.76	80.54	10.52	7.75
$\Theta_2$	166.96	18.96	1147.5	172.67	8.80	6.64
$\Theta_3$	75.72	14.52	599.87	147.75	5.22	4.06
$\Theta_4$	118.34	13.78	914.36	138.12	8.59	6.62

Table 5.5: Average query cost for a uniform data distribution.

Query Distribution	Pages accessed		Time (msec)		Improvement in	
	Cyclic	Optimised	Cyclic	Optimised	Pages accessed	Time
$\Theta_{11}$	30.05	5.33	226.55	58.26	5.64	3.88
$\Theta_{12}$	73.28	13.46	474.66	127.4	5.44	3.73
$\Theta_{13}$	31.29	11.21	232.49	106.5	2.79	2.18
$\Theta_{14}$	49.30	10.87	339.09	95.58	4.54	3.55

Table 5.6: Average query cost for a clustered data distribution.

Query Distribution	Pages accessed		Time (msec)		Improvement in	
	Cyclic	Optimised	Cyclic	Optimised	Pages accessed	Time
$\Theta_1$	12.16	4.95	96.98	61.16	2.45	1.59
$\Theta_2$	21.46	8.66	172.80	84.36	2.48	2.05
$\Theta_3$	10.54	6.53	97.93	67.98	1.62	1.44
$\Theta_4$	16.59	7.92	146.4	82.77	2.09	1.77

Table 5.7: Average query cost for a sinusoidal data distribution.

Query Distribution	Pages accessed		Time (msec)		Improvement in	
	Cyclic	Optimised	Cyclic	Optimised	Pages accessed	Time
$\Theta_1$	7.17	5.05	66.39	61.86	1.42	1.07
$\Theta_2$	21.02	7.47	168.27	70.08	2.81	2.40
$\Theta_3$	9.16	6.79	76.17	62.64	1.35	1.22
$\Theta_4$	17.25	9.02	140.39	87.26	1.91	1.61

Table 5.8: Average query cost for a linear data distribution.

In all the experiments the optimised choice vector performed better than the cyclic one. The improvement is lower for the non-uniform data distributions due to correlation of the attributes. The value of the first attribute was randomly generated and then used to generate the values of the remainder of the attributes.

### 5.5.3 Number of attributes

As the number of attributes increases, the number of attributes that are specified in few or no queries (*nonsignificant* attributes) is likely to increase. Therefore, if the cyclic choice vector is used, peer-splitting based on these attributes also increases. As a result, the performance improvement achieved by using the optimised choice vector instead of the cyclic choice vector should also increase.

Table 5.9 shows experimental results obtained using BANG files with different numbers of attributes for different query distributions. The first column of the table shows the number of attributes in each BANG file. The second column shows the average number of disk page accesses required when the cyclic choice vector was used. The third column shows the average number of disk page accesses required when the optimised choice vector was used. The last column shows the improvement achieved using the optimised choice vector rather than the cyclic choice vector. The table shows that as the number of attributes increases the improvement increases, as we expect.

The ratio of the significant attributes to the nonsignificant attributes is an important factor in the improvement that is achieved. As this ratio increases, the performance improvement of the optimised choice vector over the cyclic

Number of Attributes	Cyclic cost	Optimised cost	Gain
2	107.14	22.58	4.74
2	101.15	18.80	5.38
2	112.51	17.98	6.25
2	96.13	22.76	4.22
3	260.72	22.89	11.39
3	244.80	29.94	8.18
3	304.52	23.86	12.76
3	317.59	19.52	16.26
4	77.06	7.30	10.52
4	166.96	18.96	8.80
4	75.72	14.52	5.22
4	118.34	13.78	8.59
8	255.06	7.05	36.17
8	274.88	11.21	24.45
8	260.67	15.01	17.37
8	214.97	10.16	21.16

Table 5.9: Effect of the number of attributes on the average query cost.

choice vector decreases. That is, the total number of attributes in a relation is not as important as the proportion of significant attributes. In Table 5.9 the optimised choice vector performs better when the number of attributes is three than when it is four. This is because one of the three attributes ( $\frac{1}{3}$ ) was done significant in the former case and two of the four ( $\frac{2}{4}$ ) attributes was done significant in the latter case.

#### 5.5.4 File size

To study the effect of the file size on the performance of the optimised and cyclic choice vectors, experiments with files ranging in size from 4 Mbytes to 40 Mbytes were performed. The experiments were repeated using the uniform, clustered, sinusoidal and linear data distributions. The results are

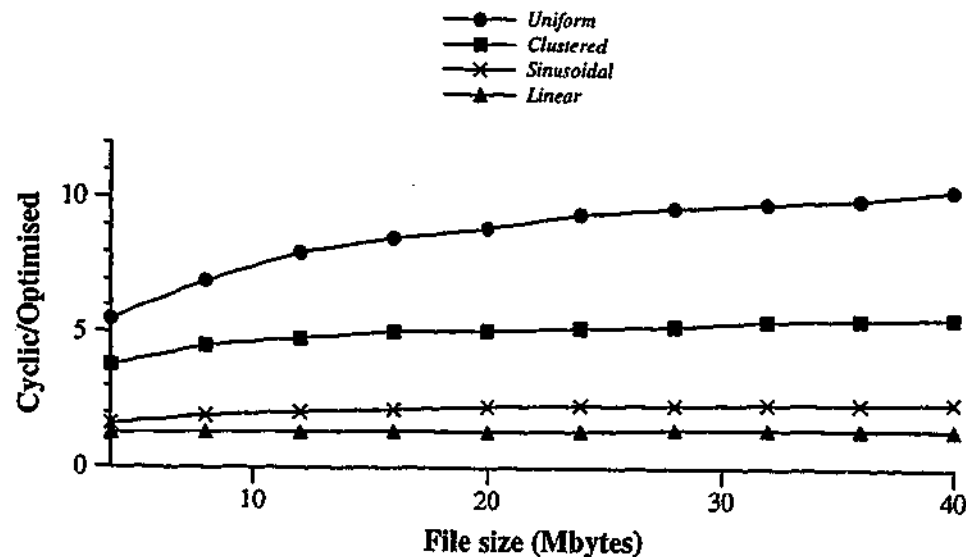


Figure 5.4: Effect of file size on relative performance.

shown in Figure 5.4. The vertical axis of Figure 5.4 represents the average query cost ratio  $\frac{\text{cyclic}}{\text{optimised}}$  and the horizontal axis represents the file size in Mbytes.

In all the experiments the optimised choice vector consistently performed better than the cyclic choice vector, as can be seen in Figure 5.4. The improvement increased (of the order of 200% for sinusoidal to 600% for uniform) as the file size increased.

### 5.5.5 Page size, elapsed time and CPU time

Experiments were conducted to study the effect of the page size on the performance of the optimised and cyclic choice vectors. Page sizes between 1 and 64 kbytes were used. The experiments were repeated using the uniform, clustered, sinusoidal and linear data distributions. The results are shown in Figure 5.5.



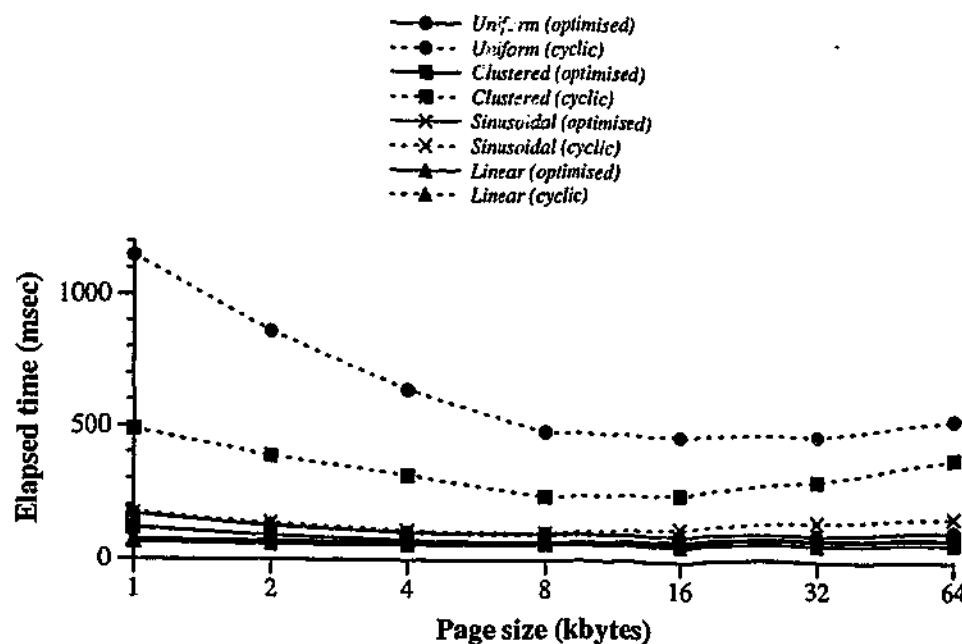


Figure 5.5: Effect of page size on performance.

As can be seen in Figure 5.5, the optimised choice vector performs better than the cyclic choice vector for all page sizes. The results show that the performance improvement is greater with smaller page sizes. This is because smaller pages result in higher number of pages and page splits, so the improvement gained by using a better splitting policy is greater.

In these results, the minimum elapsed time was achieved when the page size was 8 kbytes.

Experiments were also performed to determine which page size results in the minimum CPU time, that is, the minimum time spent searching the contents of pages rather than waiting for the disk. Again, experiments were performed using pages between 1 and 64 kbytes. The results are shown in Figure 5.6.

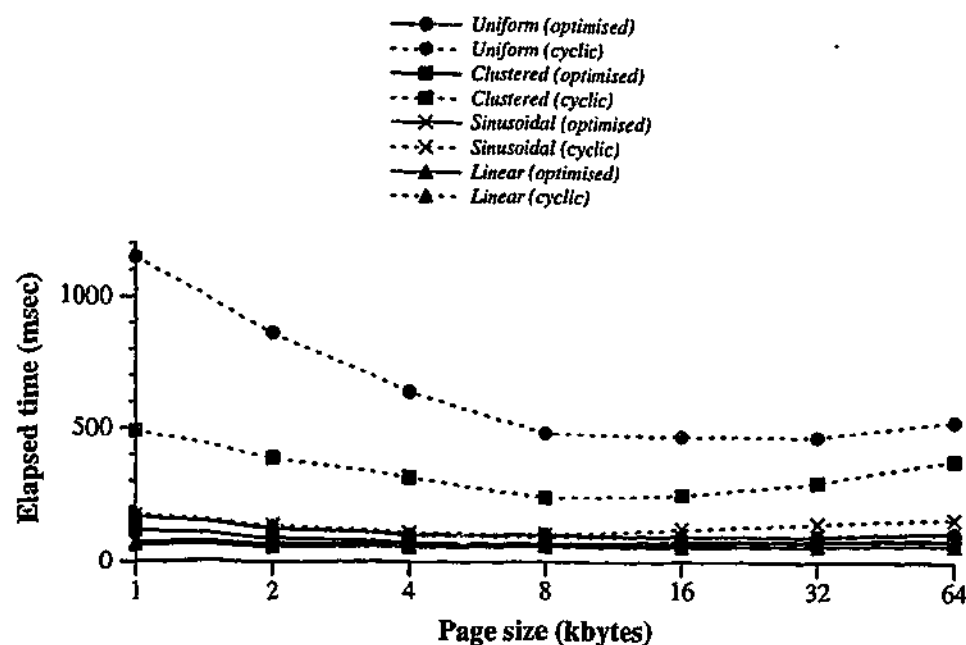


Figure 5.5: Effect of page size on performance.

As can be seen in Figure 5.5, the optimised choice vector performs better than the cyclic choice vector for all page sizes. The results show that the performance improvement is greater with smaller page sizes. This is because smaller pages result in higher number of pages and page splits, so the improvement gained by using a better splitting policy is greater.

In these results, the minimum elapsed time was achieved when the page size was 8 kbytes.

Experiments were also performed to determine which page size results in the minimum CPU time, that is, the minimum time spent searching the contents of pages rather than waiting for the disk. Again, experiments were performed using pages between 1 and 64 kbytes. The results are shown in Figure 5.6.

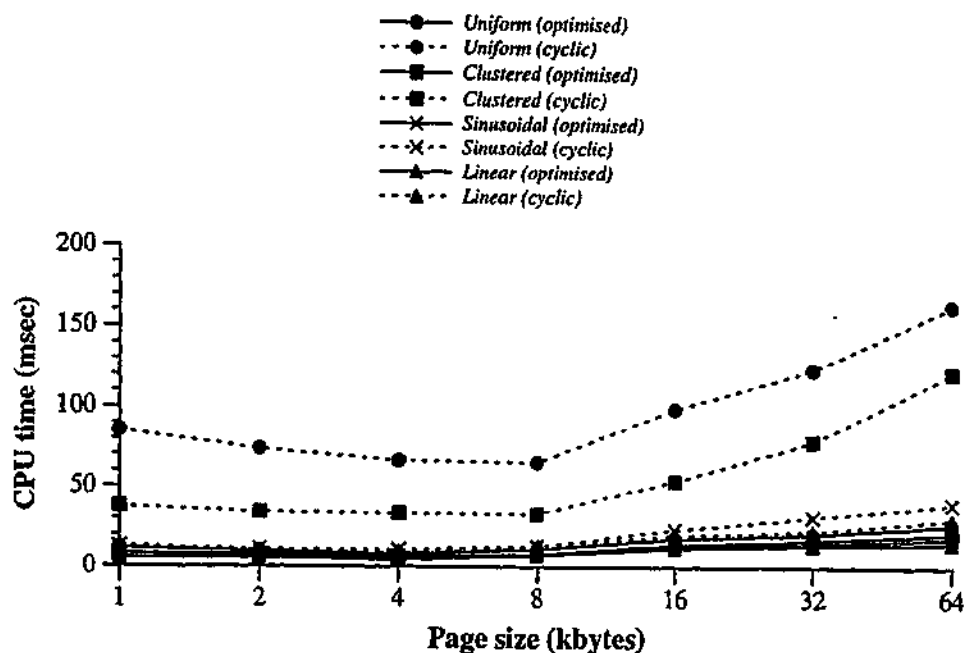


Figure 5.6: Effect of page size on CPU time.

The results indicate that for all the distributions, the minimum CPU time is generally achieved when the page size is 4 kbytes.

### 5.5.6 Stability

Query distributions can change over time. A choice vector optimised for a given query distribution may perform worse than the cyclic choice vector if the query distribution changes significantly. In order to study the stability of our optimised choice vectors, experiments were done to determine the change in performance when the query distribution changes.

We state that each query distribution is changed by  $x\%$  if each query probability,  $p$ , is randomly changed to be in the range  $p \times (1 \pm \frac{x}{100})$  prior to the whole query distribution being normalised.

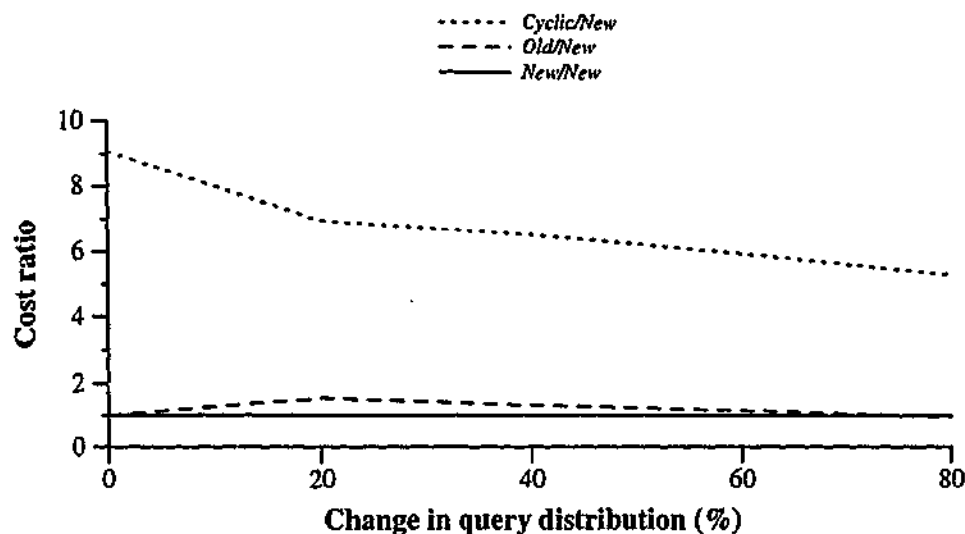


Figure 5.7: Stability of the optimised choice vector using  $\Theta_1$  and the uniform data distribution.

Figures 5.7 to 5.10 show how the average query cost is affected when the probability of each query changes by up to 80%. In each figure, three average query cost ratios are shown, using dotted, dashed and solid lines. The dotted line ("Cyclic/New") corresponds to comparing the average query cost of a BANG file built using a cyclic choice vector with that of a BANG file built using a choice vector optimised for the new, changed, query distribution. The dashed line ("Old/New") correspond to a BANG file which was built using an optimised choice vector determined by using the original query distribution. The solid line ("New/New") corresponds to BANG files built using an optimised choice vector determined using the changed probability distribution.

In some of the experiments, such as in Figure 5.10, using a choice vector produced using the original query distribution was better than using one produced by the changed query distribution. This is because the new choice

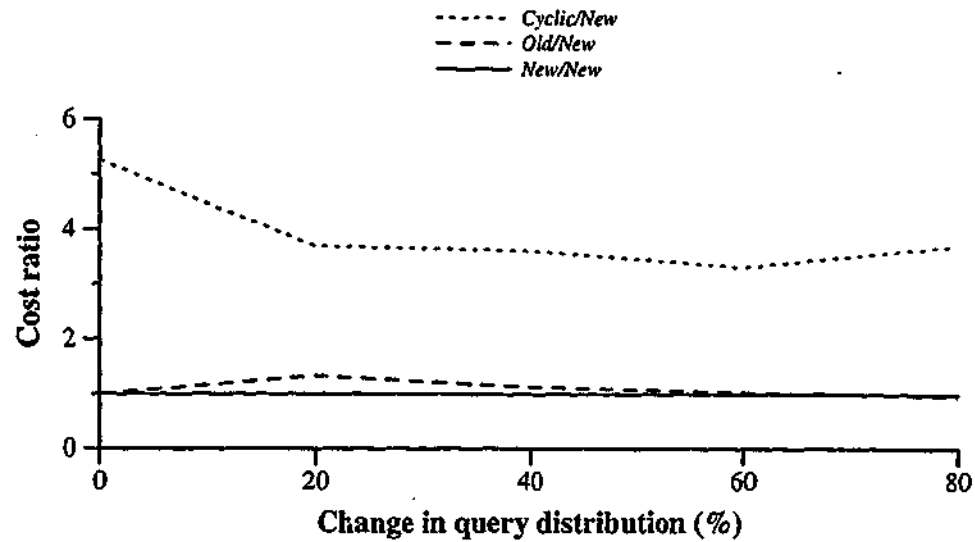


Figure 5.8: Stability of the optimised choice vector using  $\Theta_1$  and the clustered data distribution.

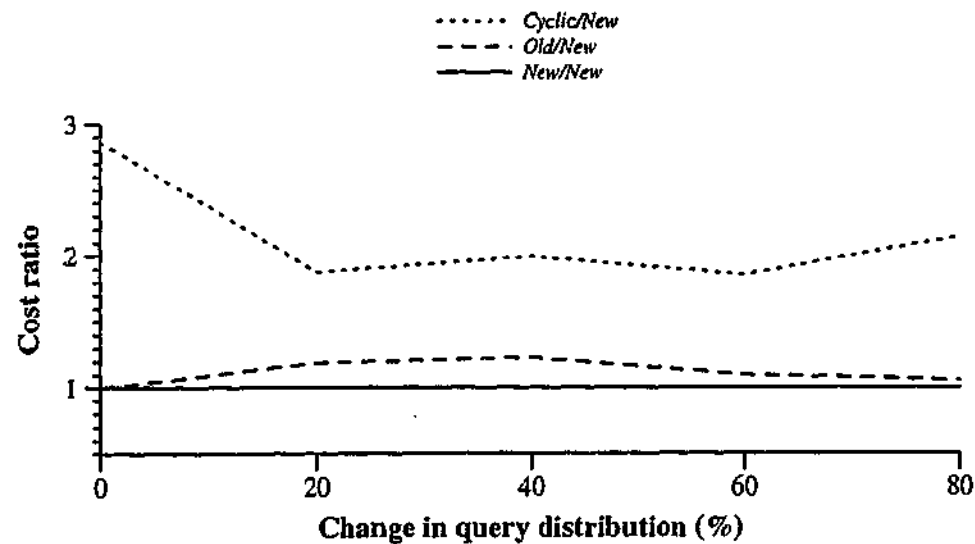


Figure 5.9: Stability of the optimised choice vector using  $\Theta_1$  and the sinusoidal data distribution.

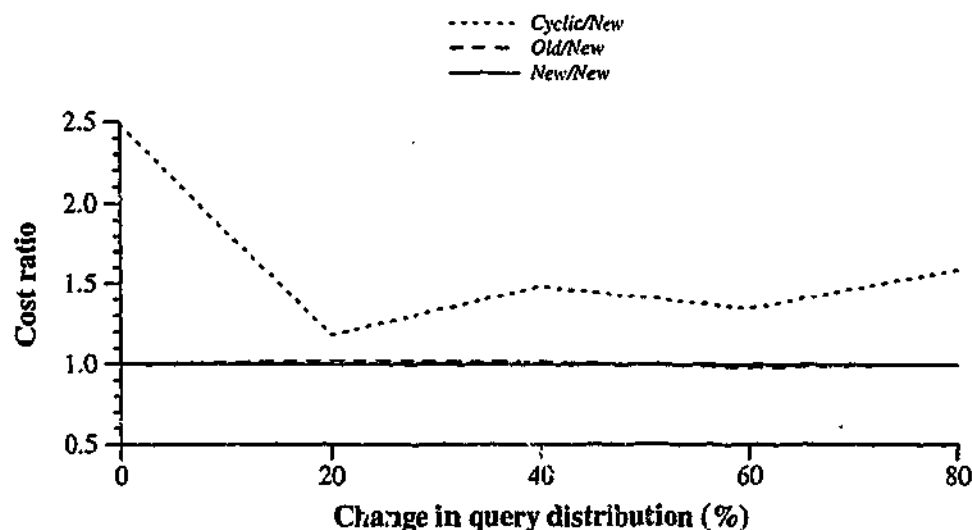


Figure 5.10: Stability of the optimised choice vector using  $\Theta_1$  and the linear data distribution.

vector is not optimal. This can occur because MMI does not guarantee to produce the optimal choice vector, and Equation 5.7 defines a good approximation of the actual cost, not the exact cost.

Our results show that the performance of the optimised choice vector of the original query distribution is almost as good as that of the changed query distribution even when the distribution was changed by 80%. As a result, we can conclude that an optimised BANG file needs to be reorganised rarely, only when the query distribution changes drastically.

### 5.5.7 Query-space size

When the query-space is the whole domain-space, we must access all the pages of the file regardless of the choice vector used. When the query is a point query the number of page accesses performed using both the optimized and cyclic choice vectors will be the same because there is precisely one

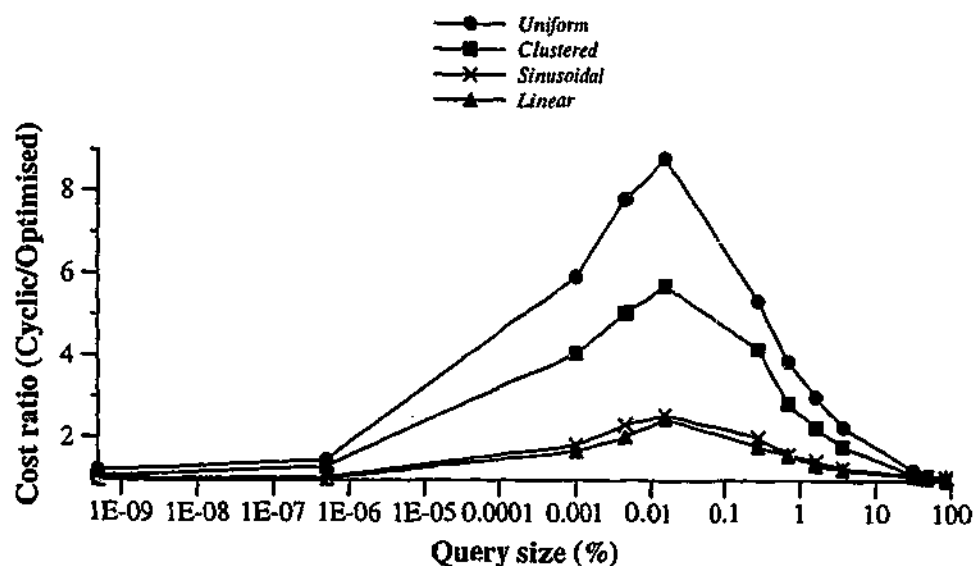


Figure 5.11: Effect of query-space size on relative performance.

destination page. Therefore, there is no advantage in using an optimised choice vector for these two extreme queries.

We performed experiments in order to study the effect of the query-space size on the performance using the optimised choice vector. Figure 5.11 shows the performance of the optimised choice vector compared to the cyclic choice vector as the query-space size changes. Each line in the figure corresponds to a different data distribution. The vertical axis of the figure represents the cost ratio  $\frac{\text{cyclic}}{\text{optimised}}$  and the horizontal axis represents the percentage the query space size is to the domain-space size. The results show that there is a large range of query-space sizes where using an optimised choice vector produces far better results than using the cyclic choice vector.

## 5.6 Conclusion

Our study shows that given a probability distribution of range queries, an efficient physical database design can be created by using minimal marginal increase and Equation 5.7. Unlike previous approaches, our approach is not limited to a uniform data distribution or to independently specified attributes, and the precise nature of any non-uniformity does not need to be known. We avoid these limitations by using a file structure which distributes records evenly amongst disk pages even when the data distribution is highly non-uniform. For our experiments, we used the BANG file.

When compared to the cyclic choice vector, our results show that the optimised choice vector produces more efficient physical database designs, reducing the average query cost. For example, in one of our experiments in which a BANG file of eight attributes was used, the optimised choice vector resulted in an improvement of a factor of 36 over the cyclic choice vector.

The improvement gained by using an optimised choice vector instead of the cyclic choice vector increases as the number of attributes increases. This is because as the number of attributes increase, the likelihood of dividing the domain space using attributes which do not occur frequently in queries is higher when the cyclic choice vector is used. This results in an inefficient physical database design. For example, in the experiments that we performed, the improvement in performance was greater when there were eight attributes in the relation than when there were two, three or four attributes. Similarly, as the ratio of attributes which occur frequently in queries to attributes which do not increases, the improvement in the performance of the



optimised choice vector over the cyclic choice vector decreases.

We found that the optimised choice vector consistently performs better than the cyclic choice vector across a wide range of file and page sizes. The improvement is greater as files get larger and pages get smaller. For both types of choice vector, the CPU time taken searching pages is minimal when the page size is 4 kbytes.

The relative size of the query-space also affects the performance of the optimised choice vector. For a query space size which is either equal to the whole domain-space or is a point query-space, both the optimised and cyclic choice vectors perform the same. However, the optimised choice vector performs better than the cyclic choice vector when the query-space size is between these two extremes.

There is no need to rearrange the optimised choice vector whenever the query distribution changes by a reasonable amount. Our experiments show that the optimised choice vector for a given query distribution will remain almost as good as the optimised choice vector built for a variation on the query distribution even when the query distribution changes by 80%.

We conclude that if the query distribution is known and a file structure which evenly distributes records amongst disk pages is used regardless of the data distribution, an optimised choice vector produces an efficient physical database design. To our knowledge, this is the most practical method of storing multidimensional data in order to best exploit a known query distribution. We therefore recommend that such structures be incorporated into new generation database systems.

## Chapter 6

# Join query processing for skewed data distributions

### 6.1 Introduction

The join operation is one of the database operations which is used to combine tuples from two or more relations based on a condition known as *join-condition*. Tuples of the input relations are combined when they satisfy the specified condition. The result of a join operation is a relation which has some or all of the attributes of the input relations. Because a join query takes two or more relations as input, it is much more costly than a range or a partial match query. The following is an example of a join query.

```
SELECT emp.name, dept.name  
FROM emp, dept  
WHERE emp.dept-no = dept.dept-no
```

(q<sub>0</sub>)

In  $q_0$ , `emp` and `dept` are the input relations, `emp.dept-no = dept.dept-no` is the join-condition and `dept-no` is the join-attribute. A *join-attribute* is an attribute specified in a join-condition.

The join operation has been extensively discussed and researched because it is frequently used and is one of the most time consuming and data-intensive operations in relational query processing. Also many of its optimizations implicitly include the optimization of other common relational operations such as selection, projection, union, intersection, difference and division.

This chapter discusses the optimization of join queries whose probability distribution is known. Based on this assumption we will introduce new optimized join algorithms for multidimensional file structures. Unlike the existing algorithms [52-54, 99], we don't assume the data distribution of the files to be uniform. Other main difference between the proposed join algorithms and the existing algorithms is that the existing algorithms assume that all partitions in a relation have the same partition level. This assumption makes many partitions to be read several times. In the proposed algorithm each partition assumes its actual partition level and an attempt is done to read it once. Although the proposed optimization algorithms can be used with any multidimensional file structures, experimental results were collected using the BANG file because of the reasons discussed in section 3.2.

This chapter consists of five sections. Section 6.2 discusses the the proposed join algorithms. Section 6.3 explains how the join query processing can be optimized. The experimental results and analysis of the proposed join algorithms is presented in section 6.4. Section 6.5 is the conclusion of this chapter.

## 6.2 The proposed join algorithms

The join algorithms discussed in this chapter have two main modules, a *selection-module* and a *matching-module*. The selection-module selects some partitions from each input relation based on the join-condition, and passes the selected relations to the matching-module. The matching-module matches the tuples of the selected partitions for join. Tuples satisfying the join-condition are inserted into the result table. The two parts are performed alternatively several times, each time with a different set of partitions, till the join of the input partitions is complete. The join of these set of partitions is equivalent to the join of the input relations. The next two subsections will discuss the selection and matching modules in detail. But first let us define the terms, *join-attribute domain*, *join-attribute edge* and *join-compatible partitions*, which are frequently used in the rest of this chapter. Examples of the terms will be given in the next section.

**Definition 6.1** *join-attribute domain is the domain of the join-attribute in the query.*

**Definition 6.2** *join-attribute edge is the side of a partition which corresponds to the join-attribute.*

**Definition 6.3** *Join-compatible partitions are partitions from different input relations whose join-attribute edges share at least one common value.*

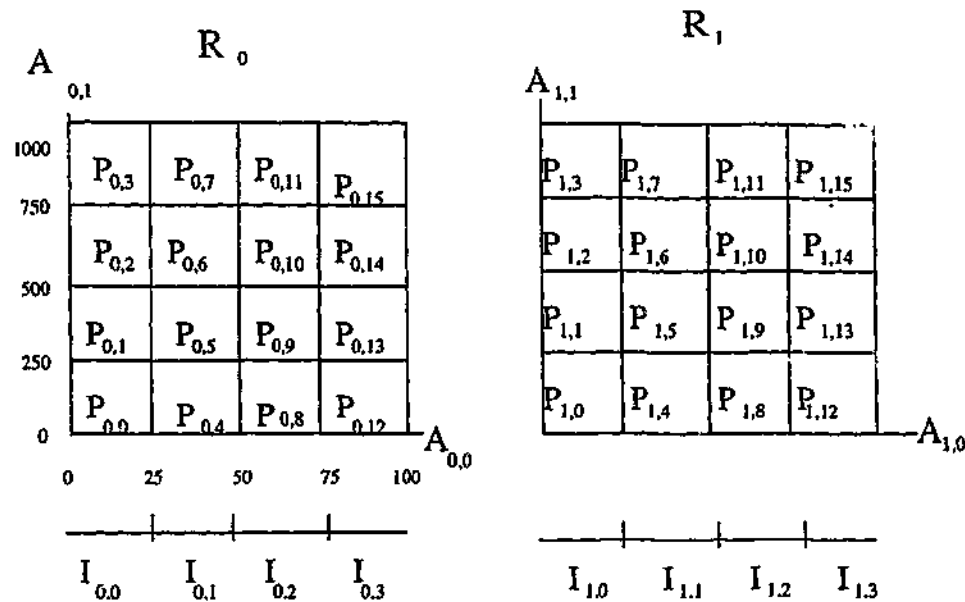


Figure 6.1:  $p_{0,0}$  and  $P_{1,0}$  are join-compatible while  $P_{0,0}$  and  $P_{1,12}$  are not.

### 6.2.1 The selection-module

The selection-module exploits the partitioning and clustering properties of the BANG or other multidimensional file structures in selecting the next set of join-compatible partitions. For example, assume the following query,  $q_1$ , which uses  $R_0$  and  $R_1$  of Figure 6.1 as input relations.

```

SELECT  $A_{0,0}$ ,  $A_{0,1}$ ,  $A_{1,1}$ 
FROM  $R_0$ ,  $R_1$ 
WHERE  $A_{0,0} = A_{1,0}$ 

```

( $q_1$ )

In processing  $q_1$ , for example, there is no point of matching  $P_{0,3}$  and  $P_{1,12}$  for join. They don't share tuples which can satisfy the join-condition because their join-attribute edges don't overlap. By *don't overlap* we mean they don't share any common join-attribute values. The join-attribute edge of  $P_{0,3}$  covers values between 0 and 25 and that of  $P_{1,12}$  covers values between

75 and 100. But  $P_{0,3}$  and  $P_{1,0}$  are join-compatible partitions, because their join-attribute edge overlap. Both cover values between 0 and 25.  $P_{0,3}$  is also join-compatible to  $P_{1,1}$ ,  $P_{1,2}$  and  $P_{1,3}$ . In fact each of  $P_{0,0}$ ,  $P_{0,1}$ ,  $P_{0,2}$  and  $P_{0,3}$  are join-compatible to  $P_{1,0}$ ,  $P_{1,1}$ ,  $P_{1,2}$  and  $P_{1,3}$ . So the selection-module identifies such join-compatible partitions and passes them to the matching-module. The matching-module matches the tuples of these partitions for join. Once the join of the current join-compatible set is completed, the algorithm again starts selecting the next join-compatible set. In case the of  $q_1$ , the next join-compatible set contain partitions  $P_{0,4}$ ,  $P_{0,5}$ ,  $P_{0,6}$  and  $P_{0,7}$  with  $P_{1,4}$ ,  $P_{1,5}$ ,  $P_{1,6}$  and  $P_{1,7}$ .

Before the start of the two modules, the join algorithms computes the number of join-compatible sets. This is done by logically dividing the join-attribute domain into a number of equal *intervals*. Then partitions are mapped into these intervals. Partitions mapping to the same interval form a set called a *wave*. The selection-module uses intervals and waves to select the next join-compatible partitions. Intervals and waves are discussed in detail in the following subsection.

### Intervals

As mentioned in the last paragraph, the join-attribute domain is logically dividing into a number of equal *intervals* before performing selection or matching. The size of each interval is equal to the size of join-attribute edge of the smallest partition in the relation. In case of  $q_1$ , the domain of the join-attribute is partitioned into 4 equal intervals as shown in Figure 6.1. This is because the size of the smallest join-attribute edge in the relation is a fourth

of the join-attribute domain size, which is 100.

For time being let's assume that the number of intervals in the input relations are equal, and each partition be within one interval. Later in this chapter we will remove these restrictions.

Intervals are labeled. Intervals corresponding to  $R_i$  are labeled as  $I_{i,0}, I_{i,1}, I_{i,2}$  and so on. Attribute values in  $I_{i,j+1}$  are higher than those in  $I_{i,j}$ . For example, in Figure 6.1 the intervals of  $R_0$  are labeled as  $I_{0,0}, I_{0,1}, I_{0,2}$ , and  $I_{0,3}$ .  $I_{0,0}$  covers attribute values between 0 and 24 inclusive,  $I_{0,1}$  covers values between 25 and 49 inclusive,  $I_{0,2}$  covers values between 50 and 74 inclusive, and  $I_{0,3}$  covers values between 75 and 99 inclusive.

Partitions whose join-attribute edge overlapping the same interval are put in the same join-compatible set. This makes the number of intervals to be equal to the number of join-compatible sets.

The number of intervals can be computed from the choice vector. For example in a choice vector of size 10 elements, let the second, fifth and seventh elements belong to an arbitrary attribute  $A_{i,j}$ . The second element of the choice vector splits  $D_{i,j}$ , the domain of  $A_{i,j}$ , into 2 equal edges. The fifth element further splits an edge, which was created by the second element, into two equal halves. The seventh element further splits an edge, which was created by the fifth element, into two equal halves. In other words, the second, fifth and seventh elements of the choice vector result in edges which are  $\frac{1}{2}, \frac{1}{4}$  and  $\frac{1}{8}$  the size of  $D_{i,j}$  respectively. Therefore if in a choice vector, the number of elements which correspond to a join-attribute  $A_{i,j}$  is  $d_{i,j}$ , then the number of intervals along  $D_{i,j}$  is  $2^{d_{i,j}}$ .

## Waves

As mentioned previously, partitions of a relation whose join-attribute edge overlapping the same interval belongs to the same join-compatible set. For example, partitions overlapping  $I_{i,k}$  are put in one set and partitions overlapping  $I_{i,k+1}$  are put in a separate set. We call such a set a *wave*.

Waves are labeled as  $W_{i,0}, W_{i,1}, W_{i,2}$  and so on. There is one-to-one mapping between waves and intervals.  $W_{i,k}$  contains only partitions overlapping  $I_{i,k}$ . This makes the number of waves to be equal to the number of intervals.

## Relations with equal number of waves

Waves which contain join-compatible partitions are called *join-compatible waves*. If the number of waves in two arbitrary input relations,  $R_i$  and  $R_j$  is the same, then each  $W_{i,k}$  is only join-compatible to  $W_{j,k}$ . For example, when using  $q_1$ , 4 waves are created in  $R_0$ , and 4 waves in  $R_1$  of Figure 6.1.

Waves created in relation  $R_0$  are:

$W_{0,0}$ , which consists of partitions  $P_{0,0}, P_{0,1}, P_{0,2}$  and  $P_{0,3}$ ,

$W_{0,1}$ , which consists of partitions  $P_{0,4}, P_{0,5}, P_{0,6}$  and  $P_{0,7}$ ,

$W_{0,2}$ , which consists of partitions  $P_{0,8}, P_{0,9}, P_{0,10}$  and  $P_{0,11}$ , and

$W_{0,3}$ , which consists of partitions  $P_{0,12}, P_{0,13}, P_{0,14}$  and  $P_{0,15}$ ,

And waves created in partitions  $R_1$  are:

$W_{1,0}$ , which consists of partitions  $P_{1,0}, P_{1,1}, P_{1,2}$  and  $P_{1,3}$ ,

$W_{1,1}$ , which consists of partitions  $P_{1,4}, P_{1,5}, P_{1,6}$  and  $P_{1,7}$ ,

$W_{1,2}$ , which consists of partitions  $P_{1,8}, P_{1,9}, P_{1,10}$  and  $P_{1,11}$ , and

$W_{1,3}$ , which consists of partitions  $P_{1,12}, P_{1,13}, P_{1,14}$  and  $P_{1,15}$ .



$W_{0,0}$ ,  $W_{0,1}$ ,  $W_{0,2}$ , and  $W_{0,3}$  are join-compatible to  $W_{1,0}$ ,  $W_{1,1}$ ,  $W_{1,2}$ , and  $W_{1,3}$  respectively.

All waves are not created in the selection-module of the same cycle. For example,  $W_{i,0}$  and  $W_{j,0}$  are created in the selection-module of the first cycle,  $W_{i,1}$  and  $W_{j,1}$  are created in the selection-module of the second cycle and so on.

### Relations with unequal number of waves

At the beginning of this chapter we discussed that the number of intervals created along the domain values of  $A_{i,j}$ ,  $D_{i,j}$ , is  $2^{d_{i,j}}$ . The ratio of the number of waves created in one relation to the number of waves created in another relation is always  $1 : 2^n$ , where  $n$  is an integer number.

For example, let us join  $R_2$  and  $R_3$  of Figure 6.2 using  $A_{2,1} = A_{3,1}$  as the join-condition. The number of intervals corresponding to  $R_{2,1}$  is  $2^1$  and that of  $A_{3,1}$  is  $2^2$ . The number of waves of  $R_2$  to that of  $R_3$  is  $\frac{2^2}{2^1} = 2^1$ , which means that the size of  $I_{2,i}$  is twice that of  $I_{3,j}$ . In other words, each wave of  $R_2$  has two join-compatible waves of  $R_3$ .  $W_{2,0}$  must join with  $W_{3,0}$  and  $W_{3,1}$ . Similarly  $W_{2,0}$  must join with  $W_{3,2}$  and  $W_{3,3}$ . Each  $W_{2,i}$  must join with  $W_{3,i+2}$  and  $W_{3,i+3}$ . In fact if the the number of waves in  $R_i$  is  $m$  times that of  $R_j$ , then each  $W_{i,k}$  must join with  $W_{j,k+m}$ ,  $W_{j,k+m+1}$ ,  $W_{j,k+m+2}$ ,  $\dots$ ,  $W_{j,k+m+m-1}$ . From now on let  $W_{j,k}$  represent all the join-compatible waves of  $W_{i,k}$ .

### Embedded waves

The join-attribute edge of a partition can span more than one interval. Such a partition can become a member of more than one wave. For example,  $P_{4,0}$

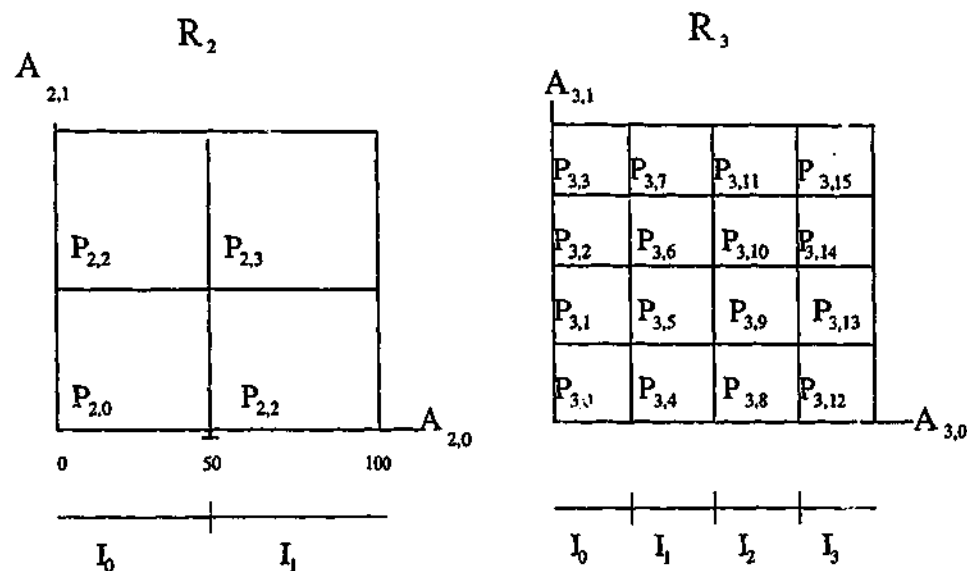


Figure 6.2: For each interval of  $R_2$  there are two intervals of  $R_3$ .

of Figure 6.3 is such a partition. It spans  $I_{4,0}$  and  $I_{4,1}$ . Therefore it is a member of  $W_{4,0}$  and  $W_{4,1}$ .

Some times all the members of a one wave are also members of another wave. This happens when some partitions of a wave span more than one interval. For example, all the members of  $W_{5,1}$  of Figure 6.4, are also members of  $W_{5,0}$ . If all members of  $W_{i,k}$  are also members of  $W_{i,j}$  for any  $j < k$  We say,  $W_{i,k}$  is *embedded* in  $W_{i,j}$ . All the join-compatible waves of the embedded wave are also join-compatible of the embedding one. For example, when we join relations  $R_6$  and  $R_7$  of Figure 6.5, using  $A_6 = A_7$  as a join-condition,  $W_{6,0}$ , which embeds  $W_{6,1}$ , is join-compatible to  $W_{7,0}$  and  $W_{7,1}$ .

### Mapping partions to waves

A partition can be mapped to particular wave number using its partition-number and its corresponding choice vector. The following steps show the

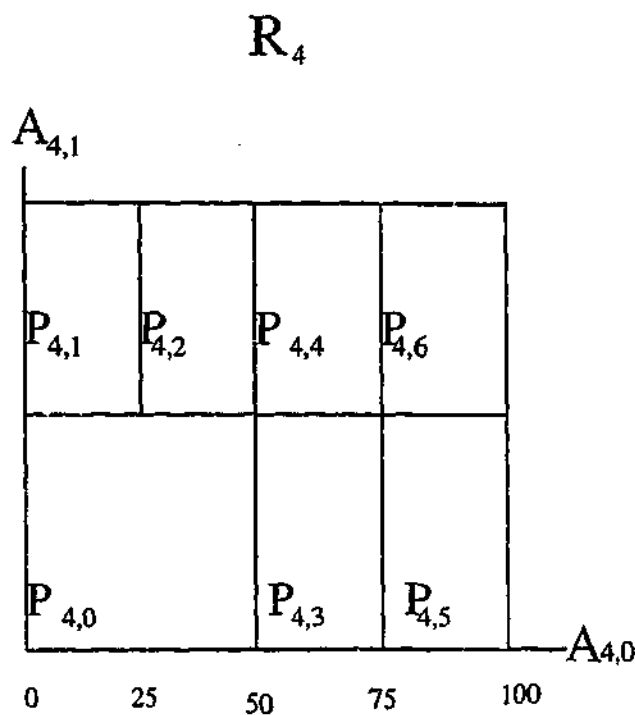


Figure 6.3:  $P_{4,0}$  spans  $I_{4,0}$  and  $I_{4,1}$ .

mapping of an arbitrary partition,  $P_{i,j}$ , into a wave-number.

1. Convert the partition-number of  $p_{i,j}$  into its equivalent binary number, say  $b_1$ .
2. If the number of bits in  $b_1$  is less than the size of the choice vector, prepend zero bits to  $b_1$ .
3. Extract all bits values in  $b_1$  corresponding to the the join-attribute and form another binary number  $b_2$ .
4. Inverse  $b_2$  and form another binary number  $b_3$ . In this step the most significant bit in  $b_2$  becomes the least significant bit in  $b_3$  and so on.
5. The wave number of  $P_{i,j}$  is the decimal equivalent of  $b_3$ .

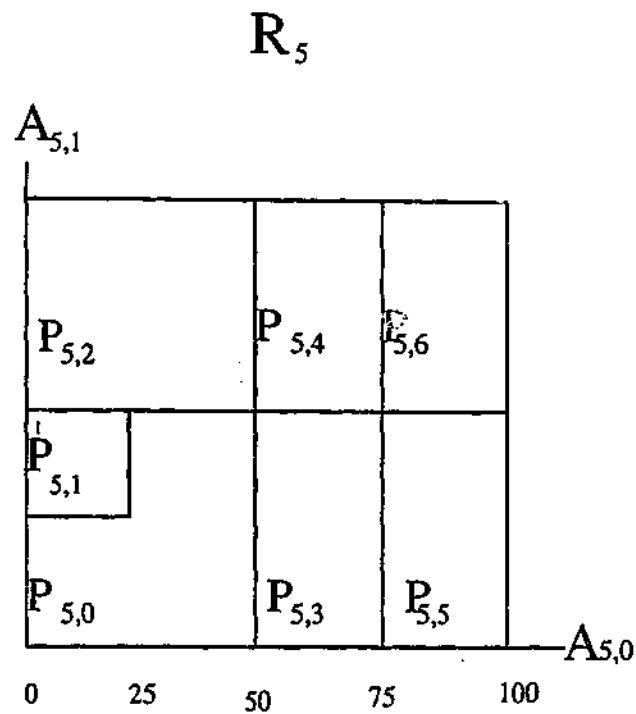


Figure 6.4:  $W_{5,1}$  is embedded in  $W_{5,0}$ .

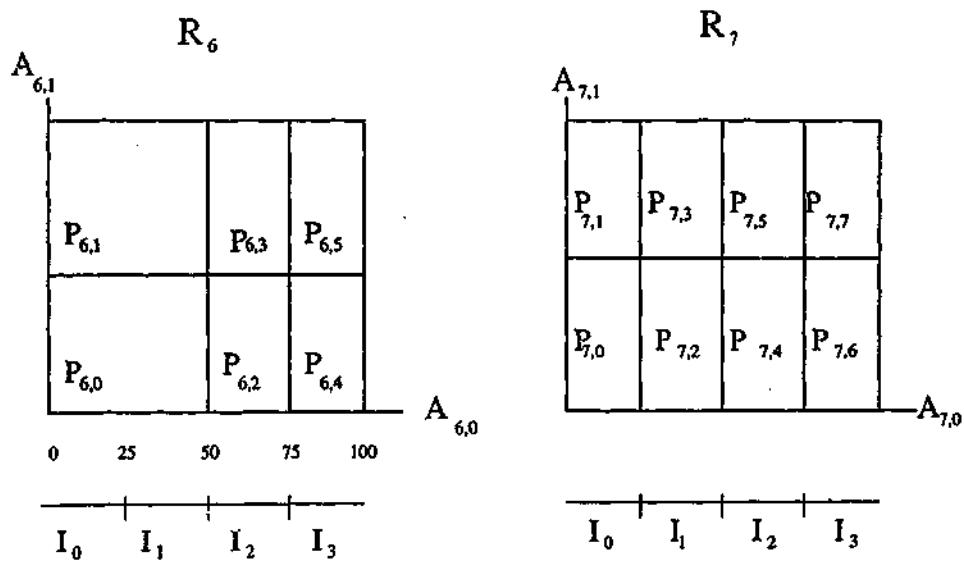


Figure 6.5:  $W_{6,1}$  is join-compatible to  $W_{7,0}$  and  $W_{7,1}$ .

For example, take an arbitrary relation  $R_i$  which has four attributes,  $A_{i,0}$ ,  $A_{i,1}$ ,  $A_{i,2}$  and  $A_{i,3}$  and a choice vector of size 6. Let  $A_{i,0}$  be the join-attribute and 001023 be the choice vector. By following the above mentioned steps, the wave-number of a partition with a partition-number of 13 can be computed as follows:

1. The binary equivalent of 13 is  $1101_b$  or  $b_1$  is  $1101_b$ .
2. The relation has a choice vector of size 6, so we prepend two zero bits to  $b_1$ , which makes  $b_1$   $001101_b$ .
3. In the choice vector, the bits corresponding to the join-attribute are bits 2, 4 and 5 (where bit 0 is the right most element) and their values are 1, 1 and 1 respectively. Therefore,  $b_2$  is  $111_b$ .
4. Inverting  $b_2$  ( $111_b$ ) results in  $b_3$ , which is  $111_b$ .
5. The wave number is the decimal equivalent of  $b_3$ , which is 7.

### Multiple join-attributes

When the number of join-attributes in a relation is more than one, the number of intervals (waves) created is a multiple of all the individual join-attribute intervals. For example, let  $A_{1,i}$  and  $A_{1,j}$  be two join-attributes. Also let  $d_{1,i}$  and  $d_{1,j}$  be their number of elements in the corresponding choice vector respectively. Then the number of waves created are  $2^{d_{1,i}} \times 2^{d_{1,j}}$ .

The steps of mapping a partition-number to a wave-number is the same as that of a single join-attribute. The only difference is that we use bit positions of multiple join-attributes instead of a single attribute. For example, if in a

partition-number, each bit-position corresponding to a join-attribute has a value of 0, then the partition belongs to  $W_{i,0}$ .

### 6.2.2 The matching-module

After the selection-module of a cycle is completed, the matching-module of the same cycle starts. Up to now, we have identified the data partitions of both waves but we haven't actually read any of them yet. It is in the matching-module where the data partitions of the current join-compatible waves are actually read and joined. The matching-module is performed in nested loop. In the outer loop, the partitions of one of the two waves are read and their tuples are put into a hash table. In the inner loop, the partitions of the other wave are read and their tuples probed into the hash table for join with the tuples of the outer loop. Let us call the wave processed in the outer loop of the matching-module the *outer-wave* and the one processed in the inner loop the *inner-wave*. A simplified matching-module algorithm is shown below. In the algorithm, *hash-table.put* is a function which hashes a record and puts it into a hash table, the *hash-table.probe* functions hashes a record and probes it into the same hash table to match it for join, the *join* function joins the matching records and the *READ* function reads a partition from disk.

The following is the matching algorithm:

```
MODULE matching(outer-wave, inner-wave)
BEGIN
  FOR partition IN outer-wave
  BEGIN
    READ(partition);
    FOR EACH record IN partition
      hash-table.put(record);
    END
  FOR EACH partition IN inner-wave
  BEGIN
    READ(partition);
    FOR EACH record IN partition
    BEGIN
      hash-table.probe(record);
      join;
    END
  END
END
```

The matching algorithm

### 6.3 Optimizing join query processing

The cost of a join query is minimised if:

1. pages which don't contribute to the join result are not accessed,
2. each page which contribute to the join result is accessed once and
3. only records which can satisfy the join condition are matched.

The proposed join algorithm is based on the partitioning of the input relations into join compatible waves before tuples are matched for join. Minimising the size of the join compatible waves minimise the number of tuples which has to be matched for join. To reduce the number of times a page is accessed, the size of one of each join compatible waves must be less than the buffer size. The relationship of a buffer size and a wave size is explained in detail in the next subsection.

The number of waves and the number of partitions per wave is affected by the choice vector. Optimal choice vectors tend to reduce the number of partitions per wave and this increases the chance of a wave to fit into the available buffer. The relationship of choice vectors and waves is explained in the subsection 6.3.2.

### 6.3.1 Buffer size vs wave size

As mentioned in section 2.1, the cost of a query is mainly measured by the number of disk accesses required to answer the query. The number of disk accesses required is significantly affected by the buffer size. Smaller buffer can result in more disk accesses than a larger one. For example, let the number of partitions of the outer-wave be 24 and the number of partitions of the inner-wave be 7. Let the buffer size be 10 blocks. Let 8 of the 10 blocks be allocated for the outer-wave, 1 block be allocated for inner-wave and the remaining 1 block be allocated for the result. Since we allocated 8 blocks for the outer-wave we can only hash 8 partitions at a time. We can read all of them in 3, which is  $\lceil \frac{24}{8} \rceil$ , loops. In the first loop we will read the first 8 partitions and hash them. Then in the same loop we will read the



seven partitions of the inner-wave, one at a time (since we have allocated one block for the wave) and join them with the 8 partitions of the outer-wave. Then we repeat the same process for the next 8 partitions of the outer-wave with the same 7 partitions of the inner-wave and so on. At the end of the third loop, the join of the two waves is complete and  $24 + \left\lceil \frac{24}{8} \right\rceil \times 7 = 45$  disk accesses are performed. 24 accesses are performed reading the partitions of the outer-wave, which is done in 3 loops, and  $\left\lceil \frac{24}{8} \right\rceil \times 7 = 21$  disk accesses are performed reading the partitions of the inner-wave. Let the number of partitions in  $W_{i,k}$  be denoted by  $|W_{i,k}|$ .

The cost (in terms of disk accesses) of joining arbitrary wave  $W_{i,k}$  and its join-compatible wave,  $W_{j,k}$ , using a buffer size of  $B$  is,  $|W_{i,k}| + \left\lceil \frac{|W_{i,k}|}{B} \right\rceil \times |W_{j,k}|$ .

The cost of processing join-compatible waves can be reduced if the wave with the smallest number of partitions is used as the outer-wave. If the buffer size is big enough to accommodate all the outer-wave partitions, each partition of the inner-wave will be read only once. For instance in our previous example if the wave with 7 partitions was done the first wave and the other with 24 partitions as the inner-wave, the cost would have been  $7 + \left\lceil \frac{7}{8} \right\rceil \times 24$ , which is 31 disk access only.

### 6.3.2 Wave size and choice vector

Choice vectors significantly affect the cost of a join query. For example, assume the join of two relations  $R_8$  and  $R_9$  of Figure 6.6 with  $A_{8,0} = A_{9,0}$  as a join-condition. Let us assume that the first-wave always belongs to  $A_{8,0}$  and the second-wave to  $A_{9,0}$ . Let the buffer size allocated for the first-wave be 2 and for the second-wave be 1 and for the result be 1. Then the cost

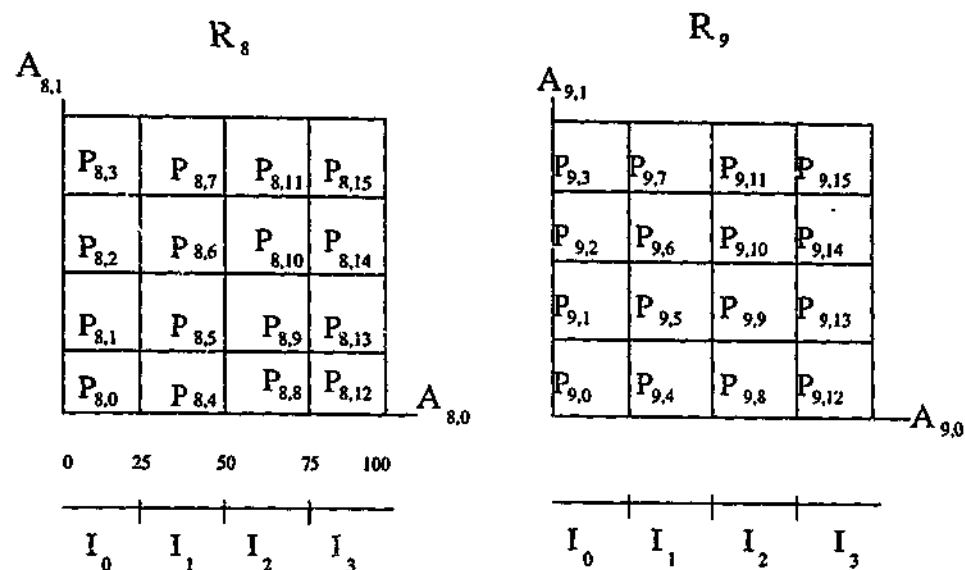


Figure 6.6: Waves with number of partitions higher than the buffer size result in higher join query cost.

of each join-compatible wave is as follows:  $4 + \left\lceil \frac{4}{2} \right\rceil \times 4 = 12$ . Since we have 4 join-compatible waves, the join of  $R_8$  and  $R_9$  will cost  $4 \times 12 = 48$  disk accesses. But if  $R_9$  was partitioned (using a different choice vector) as shown in Figure 6.7 the cost will be 32. The join using  $R_8$  of Figure 6.7 instead of  $R_8$  of Figure 6.6 costs less because the size of its corresponding waves is smaller. So allocating more elements of a choice vector to join-attributes results in smaller size waves. The problem now is, given a number of join queries and their probabilities, to find optimal choice vectors which results in minimal average cost. Finding optimal choice vectors for arbitrary query distribution is NP-hard [94]. Hence we will use heuristic algorithms to find optimal or near optimal choice vectors.

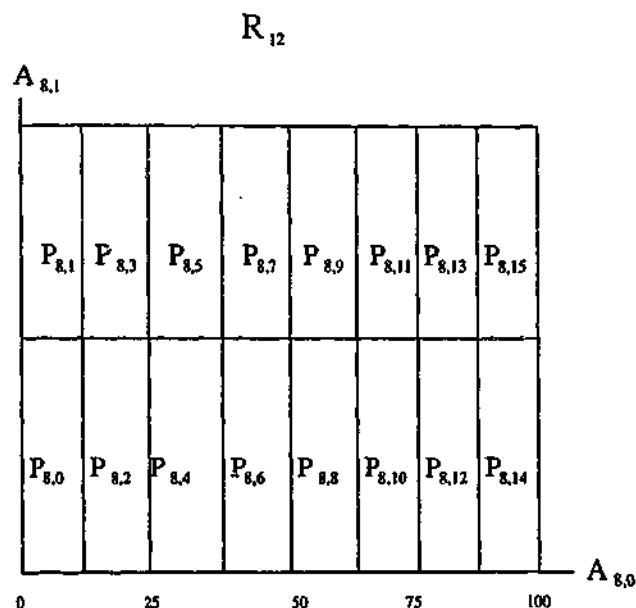


Figure 6.7: Reducing the number of partitions per wave so that they fit in the available buffer reduces cost.

### 6.3.3 Heuristic algorithms and Cost functions

The heuristic algorithm that we used to find optimized choice vectors was simulated annealing. Simulated annealing was extensively discussed in section 2.6.3. Simulated annealing uses cost functions which are dependent on the problem on hand. In our case, the problem is given a set of join queries and their probabilities, is to find optimized choice vectors which minimize the average query cost of the set. The rest of this section discusses the cost functions we used.

Assume that  $R_i$  and  $R_j$  are two arbitrary input relations. In the proposed join algorithm, the partitions of two relations is grouped into waves and then the join compatible waves are joined. So the cost of joining the two relations is equivalent to the sum of the sub join costs. The average join query cost is:

$$C_Q = \sum_q C_q \times p_q \quad (6.1)$$

where  $p_q$  is the probability of  $q$  and  $Q$  is a set of join queries.  $C_q$  is the cost of a single query. The cost of joining two join compatible waves is:

$$C_{W_k} = \min(|W_{i,k}|, |W_{j,k}|) + \max(|W_{i,k}|, |W_{j,k}|) \left\lceil \frac{\min(|W_{i,k}|, |W_{j,k}|)}{B} \right\rceil \quad (6.2)$$

Where  $B$  is the buffer size.

Let the choice vector elements of  $R_i$  be  $d_i$  out of which  $d_{i*}$  belong to the join attribute. Similarly let the choice vector elements of  $R_j$  be  $d_j$  and that of its join attribute be  $d_{j*}$ . The number of waves created in  $R_i$  and  $R_j$  is  $2^{d_i}$  and  $2^{d_j}$  respectively. So on the average each wave of  $R_i$  will contain  $2^{d_i - d_{i*}}$  and each wave of  $R_j$  will contain  $2^{d_j - d_{j*}}$ . Let  $\delta_i = d_i - d_{i*}$  and  $\delta_j = d_j - d_{j*}$ . Hence Equation 6.2 can be rewritten as:

$$C_{W_k} = \min(2^{\delta_i}, 2^{\delta_j}) + \max(2^{\delta_i}, 2^{\delta_j}) \left\lceil \frac{\min(2^{\delta_i}, 2^{\delta_j})}{B} \right\rceil \quad (6.3)$$

The cost of a query is equal to the cost of its join compatible joins and is represented as:

$$C_q = \sum_{k=0}^{2^{\min(\delta_i, \delta_j)} - 1} C_{W_k} \quad (6.4)$$

By combining Equations 6.1, 6.4 and 6.3 we end up with:

$$C_Q = \sum_q p_q \sum_{k=0}^{2^{\min(\delta_i, \delta_j)} - 1} \min(2^{\delta_i}, 2^{\delta_j}) + \max(2^{\delta_i}, 2^{\delta_j}) \left\lceil \frac{\min(2^{\delta_i}, 2^{\delta_j})}{B} \right\rceil \quad (6.5)$$

So by using simulated annealing together with Equation 6.5 we will find the optimized choice vectors.

## 6.4 Results and analysis

In this section we present the results of experiments comparing the performance of the optimised and cyclic choice vectors.

The first set of results shows the performance of the optimised and the cyclic choice vectors on different data and query distributions. The second set of results shows the effect of the number of attributes on the performance of both choice vectors. The third, fourth and fifth sets show the effect of the file size, page size and buffer size on the performance of the cyclic and optimised choice vectors.

Query distributions change over time. A choice vector optimised for one query distribution may not perform as well if the query distribution changes. A solution, based on simulated annealing and Equation 6.5, is called *stable* if a slight change in the query distribution doesn't affect the optimality of the solution. The last set of results demonstrate the stability of the optimised choice vector.

### 6.4.1 Environment

We implemented a BANG file with our extension of using a choice vector during partition splitting. In each experiment we used randomly generated queries and assigned each of them a randomly generated probability. Unless specified, we used a page size of 1024 bytes, four integer attributes per record and one million randomly generated records per relation (BANG file). We ran all our experiments on a SPARC station 20.

The data distributions used were uniform, clustered regions, a linear correlation, and a non-linear correlation function (a sine wave). Examples of these are shown in Figure 3.9. We refer to them as uniform, clustered, linear and sinusoidal, respectively.

In the experiments, four sets of query distributions were used. Query distributions in each set were generated randomly using a fixed set of seeds. The seeds used for one set were different from that of the other. In this thesis, these four sets of query distributions are referred to as  $\Theta_1$ ,  $\Theta_2$ ,  $\Theta_3$  and  $\Theta_4$ .

### 6.4.2 Effect of data and query distributions

The effect of using the optimised and cyclic choice vectors on the average query cost using different data and query distributions is shown in Tables 6.1 to 6.4. The first column in each of these tables shows the query distribution used. The second and the fourth columns correspond to the cyclic choice vector and show the cost in number of disk accesses and in lapse time, respectively. Similarly, the third and fifth columns show the costs corresponding

Query Distribution	Disc accesses		Time (sec)		Improvement in	
	Cyclic	Optimised	Cyclic	Optimised	Disk access	Time
$\Theta_1$	154634	60093	1438	589	2.57	2.44
$\Theta_2$	151735	61208	1382	588	2.48	2.35
$\Theta_3$	160048	60128	1483	594	2.66	2.50
$\Theta_4$	162986	58960	1464	554	2.76	2.64

Table 6.1: Average query cost for a uniform data distribution.

Query Distribution	Disc accesses		Time (sec)		Improvement in	
	Cyclic	Optimised	Cyclic	Optimised	Disk access	Time
$\Theta_1$	153329	62549	1397	572	2.45	2.41
$\Theta_2$	150170	60740	1381	573	2.47	2.41
$\Theta_3$	159020	60575	1351	529	2.62	2.55
$\Theta_4$	161434	58211	1402	517	2.77	2.71

Table 6.2: Average query cost for a clustered data distribution.

to the optimised choice vector. The improvement in the number of disk page accesses and time taken when using the optimised choice vector rather than the cyclic choice vector is shown in the final two columns.

In all the experiments performed the optimised choice vector performed better than the cyclic choice vector. The improvement is lower when both input relations have skewed data distributions. The more skewed the data

Query Distribution	Disc accesses		Time (sec)		Improvement in	
	Cyclic	Optimised	Cyclic	Optimised	Disk access	Time
$\Theta_1$	51684	50073	477	468	1.03	1.02
$\Theta_2$	51462	49133	480	466	1.05	1.03
$\Theta_3$	61156	59429	558	552	1.03	1.01
$\Theta_4$	51756	48356	472	441	1.07	1.04

Table 6.3: Average query cost for a sinusoidal data distribution.

Query Distribution	Disc accesses		Time (sec)		Improvement in	
	Cylic	Optimised	Cyclic	Optimised	Disk access	Time
$\Theta_1$	52213	49131	482	459	1.06	1.05
$\Theta_2$	52040	48759	485	458	1.07	1.06
$\Theta_3$	55847	51415	490	449	1.09	1.09
$\Theta_4$	52477	48177	474	439	1.09	1.08

Table 6.4: Average query cost for a linear data distribution.

distribution is, the more are the elements of the choice vector. More elements in the choice vector causes more number of waves hence less number of partition per wave which can fit into the available memory.

### 6.4.3 Number of attributes

As the number of attributes increases, the number of attributes that are specified in few or no queries (*nonsignificant* attributes) is likely to increase. As the ratio of the nonsignificant to significant attributes increases, the performance improvement of the optimised choice vector over the cyclic choice vector decreases. This is because when the cyclic choice vector is used, peer-splitting based on the non significant attributes increases. As a result, the performance improvement achieved by using the optimised choice vector instead of the cyclic choice vector increases.

Table 6.5 shows experimental results obtained using BANG files with different numbers of attributes for different query distributions. The first column of the table shows the number of attributes in each BANG file. The second column shows the query distribution used. The third column shows the average number of disk page accesses required when the cyclic



Number of Attributes	Query distribution	Number of Disk accesses		Gain = Cyclic/Optimised
		Cyclic	Optimal	
2	$\Theta_1$	23589	22572	1.05
2	$\Theta_2$	23649	22542	1.05
2	$\Theta_3$	23473	22591	1.04
2	$\Theta_4$	23434	22566	1.04
3	$\Theta_1$	66949	41478	1.61
3	$\Theta_2$	66948	41432	1.62
3	$\Theta_3$	66936	41398	1.62
3	$\Theta_4$	66939	41487	1.61
4	$\Theta_1$	154634	60093	2.57
4	$\Theta_2$	151735	61208	2.48
4	$\Theta_3$	160048	60128	2.66
4	$\Theta_4$	162986	58960	2.76
8	$\Theta_1$	524492	73400	7.15
8	$\Theta_2$	524465	73290	7.16
8	$\Theta_3$	524573	73310	7.16
8	$\Theta_4$	524512	73237	7.16

Table 6.5: Effect of the number of attributes on the average query cost.

choice vector was used. The fourth column shows the average number of disk page accesses required when the optimised choice vector was used. The last column shows the improvement achieved using the optimised choice vector rather than the cyclic choice vector. The table shows that as the number of attributes increases the improvement increases, as we expect.

#### 6.4.4 File size

To study the effect of the file size on the performance of the optimised and cyclic choice vectors, experiments with files ranging in size from 4 Mbytes to 40 Mbytes were performed. The experiments were repeated using the uniform, clustered, sinusoidal and linear data distributions. The results are

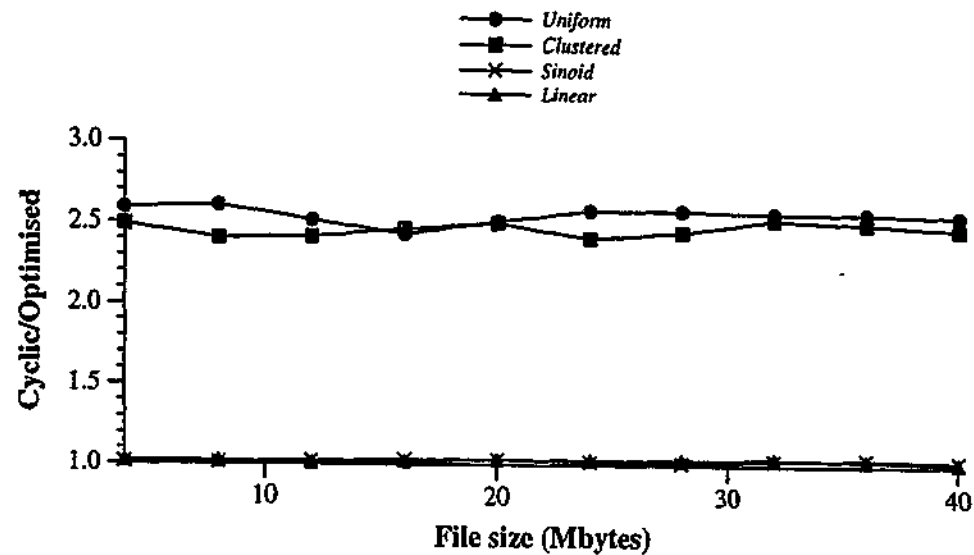


Figure 6.8: Effect of file size on relative performance.

shown in Figure 6.8. The vertical axis of Figure 6.8 represents the average query cost ratio  $\frac{\text{cyclic}}{\text{optimised}}$  and the horizontal axis represents the file size in Mbytes.

In all the experiments the optimised choice vector consistently performed better than the cyclic choice vector, as can be seen in Figure 6.8. The improvement remains nearly the same as the file size is increased, which shows that the file size has no effect on the performance of the proposed join algorithm.

#### 6.4.5 Page size

Experiments were conducted to study the effect of the page size on the performance of the optimised and cyclic choice vectors. Page sizes between 1 and 64 kbytes were used. The experiments were repeated using the uniform, clustered, sinusoidal and linear data distributions. The results are shown in

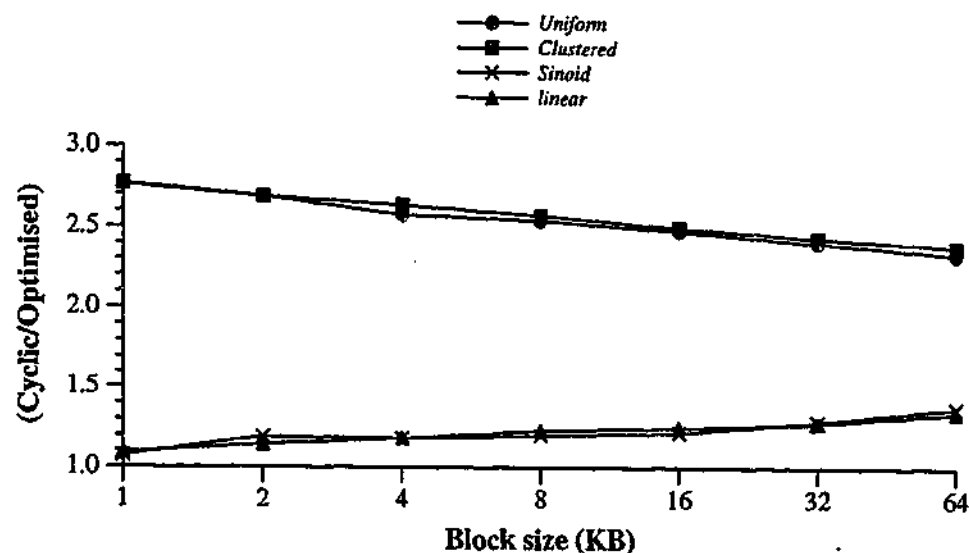


Figure 6.9: Effect of page size on performance.

Figure 6.9.

As can be seen in Figure 6.9, the optimised choice vector performs better than the cyclic choice vector for all page sizes. The results show that the performance improvement is greater with smaller page sizes when the data distribution is uniform. This is because smaller pages result in higher number of pages and page splits, so the improvement gained by using a better splitting policy is greater. When the data distribution is skewed, the improvement in cost is better when larger pages sizes are used. This is because larger pages decrease the size of the choice vector hence the number of splits based on the non significant attributes.

#### 6.4.6 Buffer size

The size of the buffer that is available for the join operation significantly affects the cost of join. If the number of partitions in each of two join

compatible waves is smaller than the available buffer size then each partition in those waves is read only once. But if each join compatible wave has partitions too big to fit into the available buffer then the partitions of one of the waves will be accessed once but that of the other will be read several times.

For example, assume  $W_{i,j}$  and  $W_{k,j}$  are two join compatible waves and  $B$  is the size of the available buffer. Let us also assume that  $W_{i,j}$  has less number of partitions than  $W_{k,j}$ . If  $|W_{i,j}| < B$ , then each partition of those waves will be accessed once. But if  $|W_{i,j}| \geq B$ , then the partitions of  $W_{k,j}$  will be accessed several times ( $\frac{|W_{i,j}|}{B}$  on the average) thus increasing the cost of the join operation. The cost is higher if the available buffer size is smaller.

The experimental results in Table 6.6 shows the relationship between the buffer size and the cost of the join operation. Column one of the table shows the ratio of the buffer size to the size of the smaller relation. Column two shows the query distribution used to perform the joins. Columns three and four show the cost of the join operations when using the cyclic choice vector and the optimised choice vector respectively. The last column shows the gain that is achieved by using the optimised choice vector instead of the cyclic choice vector. It is computed by dividing the cost obtained by using the cyclic choice vector by that of the optimised choice vector. As can be seen from the table, as the buffer size increases the cost decreases.

#### 6.4.7 Stability

Query distributions can change over time. A choice vector optimised for a given query distribution may perform worse than the cyclic choice vector if

Size ratio (buffer/relation)	Query distribution	Number of disk accesses		Gain = (Cyclic/Optimised)
		Cyclic	Optimised	
0.001	$\Theta_1$	1286212	270619	4.75
0.005	$\Theta_1$	281041	75878	3.70
0.01	$\Theta_1$	154634	60093	2.57
0.05	$\Theta_1$	64042	47474	1.35
0.1	$\Theta_1$	45893	45117	1.02
0.15	$\Theta_1$	45482	44873	1.01
0.2	$\Theta_1$	45235	44873	1.01

Table 6.6: Effect of buffer size on query cost.

the query distribution changes significantly. In order to study the stability of our optimised choice vectors, experiments were done to determine the change in performance when the query distribution changes. We state that each query distribution is changed by  $x\%$  if each query probability,  $p$ , is randomly changed to be in the range  $p \times (1 \pm \frac{x}{100})$  prior to the whole query distribution being normalised.

Figures 6.10 to 6.13 show how the average query cost is affected when the probability of each query changes by up to 80%. In each figure, three average query cost ratios are shown, using dotted, dashed and solid lines. The dotted line ("Cyclic/New") corresponds to comparing the average query cost of a BANG file built using a cyclic choice vector with that of a BANG file built using a choice vector optimised for the new, changed, query distribution. The dashed line ("Old/New") correspond to a BANG file which was built using an optimised choice vector determined by using the original query distribution. The solid line ("New/New") corresponds to BANG files built using an optimised choice vector determined using the changed probability distribution.

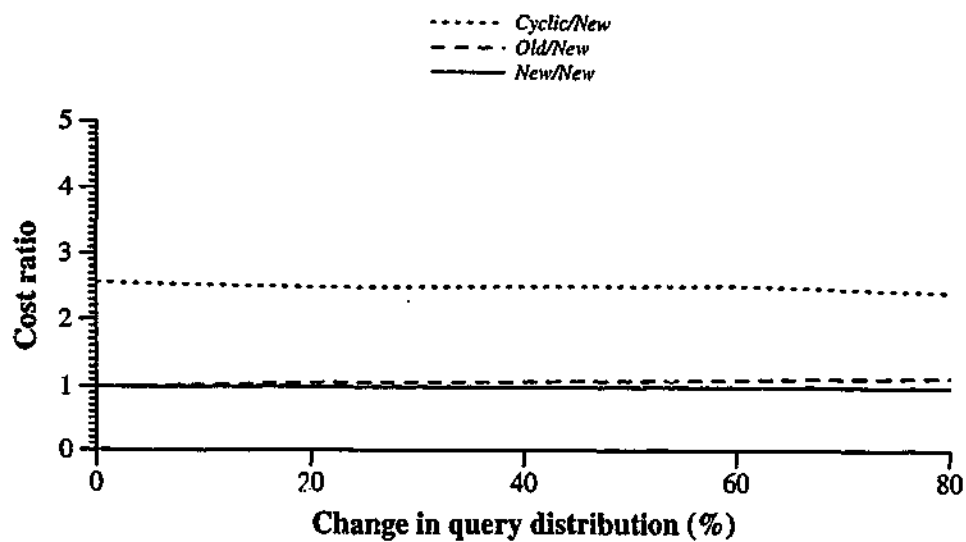


Figure 6.10: Stability of the optimised choice vector using  $\Theta_1$  and the uniform data distribution.

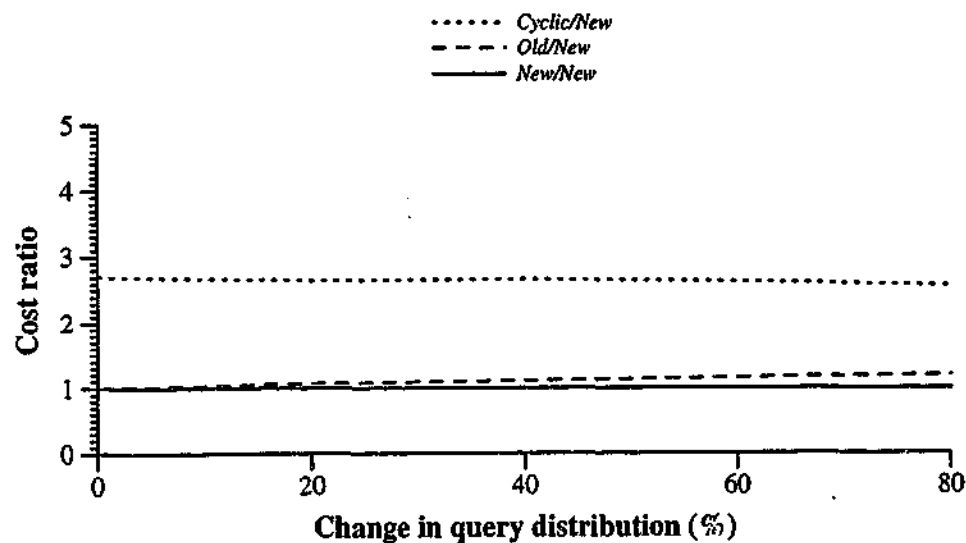


Figure 6.11: Stability of the optimised choice vector using  $\Theta_1$  and the clustered data distribution.

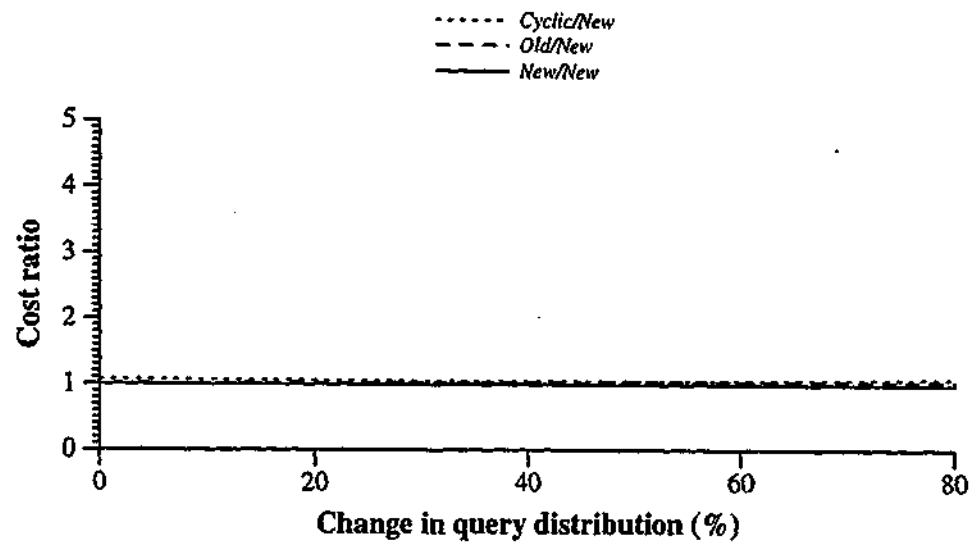


Figure 6.12: Stability of the optimised choice vector using  $\Theta_1$  and the sinusoidal data distribution.

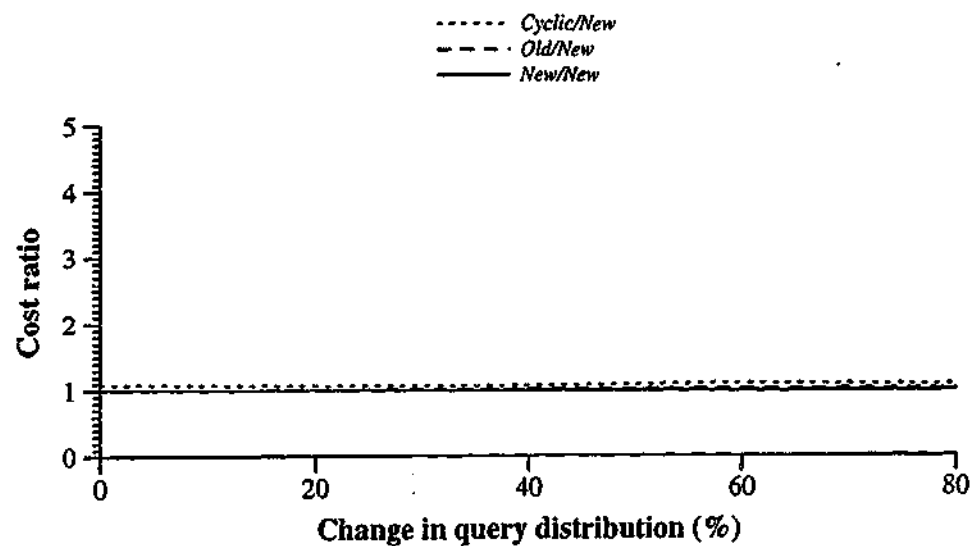


Figure 6.13: Stability of the optimised choice vector using  $\Theta_1$  and the linear data distribution.

Our results show that the performance of the optimised choice vector of the original query distribution is almost as good as that of the changed query distribution even when the distribution was changed by 80%. As a result, we can conclude that an optimised BANG file needs to be reorganised rarely, only when the query distribution changes drastically.

## 6.5 Conclusion

Our study shows that given a probability distribution of join queries, an efficient physical database design can be created by using simulated annealing and Equation 6.5. Unlike previous approaches, our approach is not limited to a uniform data distribution or to independently specified attributes, and the precise nature of any non-uniformity does not need to be known. We avoid these limitations by using a file structure which distributes records evenly amongst disk pages even when the data distribution is highly non-uniform. For our experiments, we used the BANG file.

When compared to the cyclic choice vector, our results show that the optimised choice vector produces more efficient physical database designs, reducing the average query cost. For example, in one of our experiments in which a BANG file of eight attributes was used, the optimised choice vector resulted in an improvement of 716% over the cyclic choice vector.

The improvement gained by using an optimised choice vector instead of the cyclic choice vector increases as the number of attributes increases. This is because as the number of attributes increase, the likelihood of dividing the domain space using attributes which do not occur frequently in queries



is higher when the cyclic choice vector is used. This results in an inefficient physical database design. For example, in the experiments that we performed, the improvement in performance was greater when there were eight attributes in the relation than when there were two, three or four attributes. Similarly, as the ratio of attributes which occur frequently in queries to attributes which do not increases, the improvement in the performance of the optimised choice vector over the cyclic choice vector decreases.

We found that the optimised choice vector consistently performs better than the cyclic choice vector across a wide range of file sizes, page sizes and buffer sizes.

There is no need to rearrange the optimised choice vector whenever the query distribution changes by a small amount. Our experiments show that the optimised choice vector built for the current query distribution will remain almost as good as the optimised choice vector built for a variation on the current query distribution even when the query distribution changes by 80%. There was a degradation of less than 5% in performance.

We conclude that if the query distribution is known and a file structure which evenly distributes records amongst disk pages is used regardless of the data distribution, an optimised choice vector produces an efficient physical database design. To our knowledge, this is the most practical method of storing multidimensional data in order to best exploit a known query distribution. We therefore recommend that such structures be incorporated into new generation database systems.

## Chapter 7

# Optimizing other relational operations

### 7.1 Introduction

In the previous chapters we discussed new techniques of optimising the organisation of multidimensional data in order to minimise the cost of partial match, range and join queries. In this chapter we will discuss new techniques of optimising the organisation of multidimensional data in order to minimise the cost the other standard basic operations such as selection, projection, join, intersection, union difference and division.

The selection operation includes the exact match, partial match and range queries all of which were explained in the previous chapters. Hence, the selection operation will be discussed briefly.

Projection requires each record to be read once. It cannot be optimised if it is the only operation to be performed and duplicates are to remain in the output relation.

The implementation of intersection, union difference and division is similar to that of the join operation. Thus, the same basic approach can be taken in their efficient implementation. There are a number of possible join implementations which can be used with multidimensional data. This chapter will first discuss these join implementations and then it will discuss how some of these implementations can be used for intersection, union difference and division.

This chapter consists of 10 sections. The next two sections, sections 7.2 and 7.3, discuss the implementation of selection and projection operations respectively. The different algorithms of the join operation are covered in section 7.4. The implementations of the intersection, union, difference and division operations are explained in sections 7.5 to 7.8 respectively. Duplicate removals and aggregation is discussed in section 7.9. The last section is the conclusion of this chapter.

## 7.2 Selection

The selection operation retrieves records from a file or a database which satisfy the queries condition. There are three types of selection queries, namely, exact match, partial match and range query.

1. In an *exact match query* records to be retrieved are described by specifying all the fields of the record.

2. In a *partial match query* records to be retrieved are described by specifying a subset of its fields.
3. In a *range query* the records to be retrieved are described by specifying a range of values to a subset of the fields.

In an exact match query, a search string, which is a string of binary values, is constructed from the values of the specified attributes. This is done by assigning the bit positions in the choice vector which correspond to the specified attribute according to the specified value. Once the search string is constructed the search for the record starts from the root. Each root entry is matched with the search string. The matching is done by comparing the bits of the search string to that of the partition number of the entry. If the partition level of the entry is  $d$ , and, the first  $d$  bits of the partition number match that of the search string, then this entry is a candidate to hold the required record. From all the candidate entries, the one with the highest partition level is the one which can hold the search record. Then the search descends to the next lower directory level using the chosen entry. The same process (as that of the root) is again repeated on the entries of the chosen page. The search then descends to the next lower level and so on until the data page which may contain the required record is retrieved.

A search string is also constructed when processing partial match queries. In the search string, the bits which correspond to the specified attributes are set accordingly and those bits which correspond to the unspecified attributes are left as don't care. In a partial match query, searching for the required records starts from the root. All the entries of the root page are matched

with the search string. All the entries matching the bits of the specified attribute in the search string are chosen. The search for the required records then descends to the next lower directory level using all the entries chosen from the root page. The same process is repeated in this level again. The search then descends to the next lower level and so on until all the data pages which may contain the required records are searched. The main difference in processing a partial match query and an exact match query is that, in an exact match query an entry which directly encloses the required record is chosen, while in partial match query all the entries which directly or indirectly enclose the required records are chosen.

In a range query, the construction of the search string is nearly the same as that of the partial match query. As was explained in Section 5.2, a range query can be envisioned as a subspace called *query space* within the domain space of a relation. The search string represents the query space. An answer to a range query is the retrieval of all the records in the query space. Searching for the required records starts from the root. Root entries matching the search string are candidates for the next search. If there are two candidate entries,  $A$  and  $B$ , where  $A$  encloses  $B$  and  $B$  encloses the query area, then  $A$  is dropped as a candidate. This is because  $A$  will never contain the required records.

As was discussed in the beginning of this chapter, selection consists of partial match query and range query. Hence, the techniques used, in chapters 4 and 5, to find optimised choice vectors for partial match query and range query can be used to minimise the cost of selection.

### 7.3 Projection

If we think of a relation as a table, then the select operation selects some rows from the table while discarding the other rows. The project operation, on the other hand, selects certain columns from the table and discards the other columns. If we are interested in certain attributes of a relation, we use the project operation to "project" the relation over the selected attribute list. If the attribute list includes only nonkey attributes of a relation, then it is probable that duplicate records may appear in the result. The result of the project operation is a set of records and hence a valid relation. Providing no duplicate removal is required, the number of records in the output relation is the same as that of the input relation.

The project operation is usually performed as a part of another operation. Because it requires full scan of the input relation, its implementation is the most straightforward of all the relational operations providing no duplicate removal is required. The removal of duplicate records from the result query is a problem common to this operation and many of the other relational operations. Consequently, we discuss it separately, in Section 7.9.

### 7.4 Join

The join operation is one of the database operations which is used to combine records from two or more relations based on a condition known as *join-condition*. Records of the input relations are combined when they satisfy the specified condition. The result of a join operation is a relation which has some or all of the attributes of the input relations.

The analysis and implementation of the join operation for uni-dimensional data has been an active area of research. There is a comprehensive survey paper on this active research area done by Mishra and Eich in [89]. The existing join implementations for a uni-dimensional data can also be used for multidimensional data after the multidimensional data is optimally organised using choice vectors. At the moment, there are three main types of join algorithms, namely, nested loop, sort-merge and hash join. This section contains overview of each of these algorithms, providing the foundation upon which our variation is based.

#### 7.4.1 Nested loop

The nested loop algorithm is the simplest of the join algorithms. In nested loop one of the relations being joined is designated as *inner relation*, and the other one is designated as the *outer relation*. It works in the following way. For each record in the outer relation, all records of the inner relation are read and compared with the record from the outer relation. Whenever the compared records satisfy the join condition, they are concatenated and placed in the output buffer. The outer relation is typically the smaller of the two relations.

In practice, more than one record of the outer relation is read before the inner relation is scanned. For example, Blasgen and Eswaran [13] held many records of the outer relation in memory as possible and read one record at a time from the inner relation.

A similar algorithm has been suggested on the disk block level. For example, if the size of the memory is  $B$  blocks,  $B - 2$  blocks of the outer

relation are read at a time, and the inner relation is scanned one block at a time. One block is reserved for the result records. This algorithm is often called the nested block algorithm.

Several optimisations can be applied to the above algorithm. Two important ones are, the use of a *hash table*, and *rocking*. Instead of comparing every record in the inner relation with every record of the outer relation, the records of the outer relation can be inserted into a hash table using some attributes to form the hash key. The records of the inner relation are then used to probe the hash table, searching for records to join with. This significantly reduces the number of comparisons required.

A further step towards efficiency consists of *rocking* [67] as suggested by Kim. Rocking is used when the outer relation is larger than its memory buffer. On the first pass through the inner relation, the inner relation is read from disk. On subsequent passes, part of the inner relation will already be in the memory, from the previous pass. This part need not be reread from disk. The name, rocking, derives from the observation that one implementation of this is to read the inner relation forwards and backwards on alternate passes. Thus the beginning and the end of the relation is only read on alternate passes.

In most operating system it is much more efficient to read a file forward than backwards. Under these circumstances, a better implementation of rocking is to read the file in a circular manner. Each pass should start by processing the records already in memory. It should then start reading from the end of the last part of the file read during the previous pass and read to the end of the file. It should then go back to the start of the file and read to



the start of the first block of the file which was in memory at the beginning of the pass. The same number of blocks are read as in Kim's scheme, but the total time taken to read it will be shorter because the file was always read in the forwards direction [51].

### 7.4.2 Sort-merge

The sort-merge join is done in two phases, namely, a sorting phase and a merging phase. In the sorting phase, each relation is physically sorted in its respective attributes and in the merging phase, both relations are scanned in the order of the join attributes, and records satisfying the join condition are merged to form a single relation. Whenever a record from the first relation matches a record from the second relation, the records are concatenated and placed in the output relation.

In the sorting phase, depending on the size of available memory, a number of sorted partitions are created. Replacement sections, as described by Knuth [70], can be used to generate the initial sorted partitions. In the merging phase, corresponding sorted partitions are scanned and records with the matching join attributes are joined. By first sorting both relations, the merging phase is performed in linear time in the size of the relations.

If the relations are presorted, this algorithm has a major advantage over the other join algorithms because each relation is scanned only once. Further if the number of the records in the output relation is low in comparison to the number of records in either input relations, then the number of records to be compared are considerably lower than that in the nested loop join algorithm. Blasgen and Eswaran has shown that this algorithm is most efficient in a

uniprocessor system [13]. Blasgen and Eswaran and another researcher, Su, in [132] suggested that in the absence of indexes and knowledge about the selectivity, and if there is no basis for choosing a particular join algorithm, then the sort-merge algorithm is often found to be the best choice.

### 7.4.3 Hash joins

In a hash join, the join attribute values of each record in the first relation is hashed and then put in a hash table according to the hash value. Similarly the join attribute values of each record in the second relation is hashed and is probed the same hash table for join [24]. The hash join tries to take the advantage of nested loop and sort-merge join algorithms. It takes the advantage of the fact that the nested loop algorithm only requires a single scan of the input relations if one of the two relations can be completely contained in memory. They aim to partition the relations so that this is possible. Also it takes the advantage of merge-sort in comparing only records which can possibly satisfy the join condition [14, 45]. A large number of join algorithms using hashing has been proposed and we will briefly discuss some of them in the following subsections.

#### GRACE hash join

GRACE hash join consists of two phases, namely, the *partitioning phase* and the *matching phase* [68].

During partitioning phase each relation is split into equal number of partitions. This is done by reading each record of the input relations and applying

hash function to its join attributes. The results of applying the hash function are used to form a hash key for each record. The hash key is used to determine which output partition each record is placed in. The same hash function must be used to partition each relation, producing the same number of partitions,  $P$ , from each relation. If two records must be joined they will have the same hash keys, and, therefore, will be in the corresponding partition of each relation. If the smaller partition of a corresponding pair of partitions is larger than main memory the pair of partitions are themselves partitioned into pairs of smaller partitions. This process continues until at least one partition in each pair can be contained in memory.

In the matching phase of the GRACE hash join algorithm, the nested loop algorithm is applied to each pair of partitions. In each case, the outer relation is read and its records are inserted into a hash table. Then the inner relation is scanned and the hash table is probed to join the records. Records satisfying the join condition are concatenated and placed in the output buffer.

### Hybrid hash join

The hybrid hash join algorithm is to a large extent similar to the GRACE hash join [24, 128]. The difference lies in the fact that the hybrid hash join algorithm does not write out all partitions to disk. It starts the join process on the first pair of partitions while the second relation is being partitioned.

Instead of writing out each partition to the disk as it is created, the hybrid hash join keeps one partition in main memory (in a hash table) while writing out all the others to the disk. When the second relation is partitioned, records which hash into the partition which corresponds to the one in the hash table,

are joined with the records of the first relation by probing the hash table.

Keeping one of the partitions in the memory during the partitioning phase of the first relation, minimizes the I/O activity to the extent of not having to write the partition to disk and then read it back once the partitioning phase is complete. This is particularly advantageous when the size of each partition is quite large.

A number of hybrid hash join variations have been proposed. Their primary aim has been to overcome the problems of uneven distribution of data which can result in large differences in the sizes of the partitions of a relation [96, 112, 144].

#### 7.4.4 The proposed join algorithm

Our version of join implementation was extensively discussed in the previous chapter. The proposed join implementation is a variation of the hash join algorithm. Hence, in this subsection we will explain how it is related to the other hash joins.

As was discussed in Section 7.4.3, a hybrid join algorithms has 2 phases, namely, the partitioning phase and the matching phase. The main difference between the various hybrid joins and ours is mainly in the partitioning phase. We know that in the BANG file as in all other multidimensional file structures, the data is already partitioned. So our join algorithms exploits the existing partitions of such file structures to skip or minimize the partitioning phase.

Our algorithm starts by mapping the domains of all the join attributes into one domain. This is done by assuming the bit positions in the choice

vector which correspond to the join attributes to belong to one imaginary attribute. For example, if the number of join attributes is 3 and each one of them has 2 choice vector bits, then the number of choice vector bits of the imaginary attribute will be 6. Let us call these bits  $\alpha$  bits. If the sizes of the  $\alpha$  bits in the joining relations is different, then the minimum of all the sizes is considered to be the size of the  $\alpha$  bits. If the size of the  $\alpha$  bits is  $n$ , then the imaginary domain is partitioned into  $2^n$  intervals. These intervals are labeled as  $0, 1, \dots, 2^n - 1$ .

A partition intersects an interval if the values of its  $\alpha$  bits in its partition number is equal to the interval label. For example, if the size of the  $\alpha$  bits is 6, and the  $\alpha$  bits in the partition-number of a partition  $P$  have values of 0, 0, 0, 0, 1, 0, 1, then  $P$  intersects interval 5, which is 0000101 in binary. Let us call the set of partitions from  $R_i$  which intersect interval  $j$  as  $W_{i,j}$ . For example, the set of  $R_1$  partitions intersecting interval 0 is  $W_{1,0}$ .

Once the size and the positions of the  $\alpha$  bits are known, our algorithm searches both joining relations,  $R_1$  and  $R_2$ , for members of  $W_{1,0}$  and  $W_{2,0}$ . If the number of partitions in  $W_{1,0}$  are less than those in  $W_{2,0}$ , records in  $W_{1,0}$  are placed in a hash table and those in  $W_{2,0}$  probe the hash tables for join. But, if the number of partitions in  $W_{2,0}$  are less than those in  $W_{1,0}$ , records in  $W_{2,0}$  are placed in a hash table and the records in  $W_{1,0}$  probe the hash tables for join. Once the join of  $W_{1,0}$  and  $W_{2,0}$  is done, a similar process is repeated for the join of  $W_{1,1}$  and  $W_{2,1}$ , then for  $W_{1,2}$  and  $W_{2,2}$  and so on till the join of  $W_{1,2^n-1}$  and  $W_{2,2^n-1}$ .

Each set of partitions doesn't have to be processed in a separate loop. If the smallest partition in a set spans many intervals, then all the partitions

which intersect those intervals can be processed in one loop. Hence the number of loops can be much less than the number of intervals.

If a set contains partitions of different sizes, records in a large partition may be processed in different loops. Some records of such a partition may not be put in a hash table but instead be placed in partitions, according to their hash values, which will be processed in the coming loops.

## 7.5 Intersection

An intersection is an operation which takes two relations as input, and results in a third relation which includes records which are in both input relations.

The implementation of the intersection operation is similar to that of the join. The primary difference (other than the result of the operation) is that, with the intersection operation all the choice vector bits can be considered as  $\alpha$  bits. But if the size of the  $\alpha$  bits is equal to the size of the choice vector, each interval on the average will be intersected by one page. This means that one page from each relation will be read and processed in each loop of this algorithm. But processing one page at a time will substantially increase the number of directory traversals. So to minimise the traversal of directory pages, the number of intervals must be reduced. Reducing the number of intervals will increase the average number of data pages intersecting an interval. So to reduce the number of intervals, the number of  $\alpha$  bits can also be reduced. The number of  $\alpha$  bits must be reduced until the number of pages intersecting an interval comes close to the available memory.

Without loss of generality let us assume that the current intersecting

sets of partitions to be  $W_{i,j}$  and  $W_{k,j}$  and the number of partitions in  $W_{i,j}$  to be less than that of  $W_{k,j}$ . The implementation of the join operation of section 7.4.4, works by reading records of the  $W_{i,j}$  matching certain hash values. The intersection operation can be implemented in the same way provided that all of the records of the  $W_{i,j}$  matching the given hash value are in memory at once. During the pass over the  $W_{k,j}$ , those records in the  $W_{i,j}$  which match (intersect with) those in the  $W_{k,j}$  should be marked. After all the records in the  $W_{k,j}$  have been read, the marked records of the  $W_{i,j}$  can be written out. This also ensures that no duplicate records are written.

## 7.6 Union

A union is an operation which takes two relations and results in a third relation which includes all records that are in either or in both input relations. Duplicates must also be removed. However, unlike intersection operation, the answer to the union are not present in one relation. To remove duplicates without performing passes over the current sets of partitions, we must have all the records from both sets matching certain hash values in memory simultaneously. If this condition is met, the algorithm is simply a modification of the join algorithm which performs the union operation instead of join in the loop. The number of partition in  $W_{i,j}$  and  $W_{k,j}$  must not be larger than the memory buffer allocated to them. This can be done by choosing a reasonable size of  $\alpha$  bits.

The output partitions must be processed to remove the duplicates, and the duplicate removal is described in Section 7.9.

## 7.7 Difference

The difference operation, which is denoted as  $R_1 - R_2$ , is an operation which takes two input relations,  $R_1$  and  $R_2$ , and results in a third relation which includes all the records that are in  $R_1$  but not in  $R_2$ .

The initial location of the output records of the difference is the first relation,  $R_1$ . This is different from the union operation, in which the result records come from both relations. It is also different from the intersection operation, in that the result records are found within one specific relation. The difference operation is not commutative, and requires that duplicate records be removed from the output. To avoid performing a final pass over the output file to remove duplicates, each partition of the first relation must be held in memory at one time. Thus the partitions of first relation must be placed in the hash table, even if they are more than those of the second relation.

The difference operation can be implemented using the same algorithm as the join, given in Section 7.4.4, except that the partitions of the first relation must be placed into the hash table and those of the second one must probe the hash table later. If the number of partitions in the current set of the first relation is more than the allocated memory, duplicate records will have to be removed from the output partitions after it has been produced, as described in Section 7.9.

The main part of the difference algorithm can be implemented in a manner similar to the intersection operation, described in Section 7.5. The only change that is required is that instead of writing out the marked records, the



Subject code	Subject name
CS001	Databases
CS002	Operating systems

Table 7.1: SUBJECTS table

Subject ID	Subject code
STU001	CS002
STU002	CS001
STU003	CS001
STU003	CS002
STU004	CS001
STU005	CS002
STU006	CS001
STU007	CS001
STU007	CS002
STU008	CS002
STU009	CS002

Table 7.2: STUDENTS-SUBJECTS table

unmarked records should be written out.

## 7.8 Division

The division process is best illustrated by considering the division of a relation with two columns by a relation with single column. As an example assume the division of the STUDENTS-SUBJECTS relation, shown in Table 7.2, by SUBJECTS relation which is shown in Table 7.1.

The result relation, the quotient, will be a relation with one column which is the Student ID. containing the ids of the students who took all the subjects in the SUBJECTS table. The rows in the quotient relation will consist of

Student ID
STU003
STU007

Table 7.3: Answer

students ids of the students who took all the subjects in the SUBJECTS table. With our sample data, the new relation will consist of STU003 and STU007 as shown in Table 7.3.

Our new division algorithm, on relations having multidimensional data, starts by organising the data with optimised choice vector. The process of finding the optimised choice vector is the same as that of join operation and was discussed in Chapter 6. Like the join, the division operations consists of two phases, namely, *the selection phase* and *the hashing phase*.

The selection phase is exactly the same as that of the join operation and was explained in Section 6.2.1. The hashing phase has the following features.

1. Two hash tables, one for the divisor and the other one for the quotient are created.
2. Each divisor is assigned a unique sequence number.
3. For each quotient candidate, a bit map is kept. The bit map contains a bit for each divisor, indexed by the sequence number.
4. When a quotient candidate is found, the bit corresponding to the divisor is set in the bit map of the quotient candidate.
5. The final quotient consists of all quotient candidates which have all the bits set in their bit map.

## 7.9 Duplicate removal and aggregation

The removal of duplicate records is implicit in most relational operations. If an operation is of the type select-operation-project, and the attributes on which the operation is performed are not included in the records after the projection, the output could include duplicate records which must be removed.

We eliminate duplicate records from a relation by arranging the records into partitions, such that each partition is smaller than the size of main memory. The partitions can then be read into main memory, the duplicates removed, and the remaining records written out. The first step is the normal process of partitioning a relation. Memory which is not used during the execution of relation operation can be used to create the initial part of the index at little additional cost.

Aggregate functions like SUM, AVG, MAX, MIN etc., often group records together and produce some computed output. Therefore aggregate functions can be implemented using the same basic algorithm as duplicate removals.

## 7.10 Conclusion

In this chapter we covered new techniques of optimising the organisation of multidimensional data in order to minimise the cost of the standard relational operations such as selection, projection, join, intersection, union difference and division.

The selection operation is either exact match, partial match or range query. The cost of exact match query is determined by the depth of the

BANG file so no optimal organisation of the multidimensional data is required. Finding optimised choice vectors which minimise the cost of the partial match and range queries were covered in Chapters 4 and 5 respectively, and a brief summary of them is in this chapter for completeness.

Projection requires each record to be read once. It cannot be optimised if it is the only operation to be performed and duplicates are to remain in the output relation.

Duplicate records are eliminated from a relation by arranging its records into partitions, such that each partition is smaller than the size of main memory. The partitions can then be read into main memory and the duplicates removed.

Aggregate functions often group records together and produce some computed output. Thus they can be implemented using the same basic algorithm as duplicate removals.

The implementation of intersection, union difference and division is similar to that of the join operation. Thus, the same basic approach, as the one discussed in Chapter 6, can be taken in their efficient implementation.

## Chapter 8

# Conclusions and Future work

### 8.1 Conclusions

Effective and efficient management of a large volume of data is critical in modern and future computer applications. As the size and speed of computer systems has increased, so has the amount of data which has to be manipulated. While the CPU speed is still doubling nearly every eighteen months, the performance of the secondary storage devices is not increasing on the same rate. That is why the cost of answering a query is mainly measured by the number of disk accesses performed to retrieve the records described by the query. Therefore the efficiency with which these devices are used continues to be very important. This thesis has addressed the issues associated with improving the efficiency with which these devices are used. The improvement is achieved by way of organising the data based on the distribution of queries.

Minimizing the cost of uni-dimensional access methods has been extensively studied, hence the aim of these thesis is to find techniques of optimally organising multidimensional data. The lack of order that preserves spatial proximity of records in uni-dimensional access methods makes them much easier to design than multidimensional access methods. This is because there is no total ordering of objects in two or higher dimensional space that completely preserves spatial proximity.

Optimally organising multidimensional data is NP-hard. To circumvent the problem, in this thesis, we use heuristic solutions, that is, we look for total orders that preserve spatial proximity at least to a great extent. Our aim is that objects located close to each other in the original space should likely be stored close together on the disk. This could contribute substantially in minimizing the number of disk accesses per query. Existing solutions are limited to uniform data distribution or to optimising either a partial match query or range query. The solutions in this thesis include skewed data distributions and all types of relational queries.

For each type of relational operation a new algorithm was proposed. Also, for each type of relational query, a cost model was proposed in order to come up with cost functions that are used in association with the heuristic algorithms. The proposed cost models are more accurate than the existing cost models [17, 51, 136]. Unlike the existing cost models, the proposed cost models doesn't ignore the cost associated with directory pages.

We found that the proposed organisation of multidimensional data consistently performs better than the standard organisation across a wide range of file sizes, page sizes, and buffer sizes. In one of our experiments, the gain

in performance was 3617%. The improvement in performance was greater for relations containing a larger number of attributes. Further more, the proposed data organisation is not that sensitive to minor changes in the query distribution. In more than 90% of our experiments, a change in the query distribution of up to 20% has a minimal impact on the performance (less than 5% degradation). Often the degradation is not significant (less than 20%) even when the original query distribution is changed by 80%.

In range queries, the relative size of the query-space also affects the performance of the proposed solution. For a query space size which is either equal to the whole domain-space or is a point query-space, both the proposed data organisation and the standard data organisation perform the same. However, the proposed solution performs better than the standard one when the query-space size is between these two extremes.

To our knowledge, this is the most practical method of storing multidimensional data in order to best exploit a known query distribution. We therefore recommend that such structures be incorporated into new generation database systems.

## 8.2 Future work

There are a number of open problems resulting from the work in this thesis. The heuristic algorithms that are used in this thesis to generate the optimised organisation of multidimensional data are minimal marginal increase and simulated annealing. Minimal marginal increase is a greedy algorithm and is not guaranteed to find the optimal solution. The time taken by simu-

lated annealing to come up with the optimal solution may not be an option for some applications. So there is a scope of finding heuristic algorithms that are fast, dynamic and those which result in the optimal organisation of multidimensional data.

A lot of the solutions that we proposed in this thesis can be done in parallel. For example, in the proposed join algorithm, the join compatible waves can be processed in parallel. Also, in recent years a lot of researches came up with parallel algorithms to implement relational database operations [12, 80, 82, 92, 93, 118, 123, 129, 138]. These algorithms can be enhanced to exploit the proposed multidimensional data organisation. This is another area that can be investigated.

The full advantage of processing queries in parallel can not be achieved unless the data is striped across multiple disks. The optimal way of striping skewed multidimensional data in order to speed up query processing is an area which has never been touched before.

In section 6.4.6 we showed that how the buffer size affects the cost of a join query. Also, Evan et. al. in [53] show that the way the available buffer is split between the two input relations significantly affects the cost of the join operation. The optimal way of dividing the available memory between the two input relations of the join operation to further optimise the proposed join algorithms is another area to be looked at.

In multidimensional files, the way transactions are controlled and the way locks are administered are not yet investigated. Also the way constraints are handled, specially foreign key constraint, needs further investigation.



## Bibliography

- [1] E. Aarts and J. Korst. *Simulated annealing and Boltzmann Machines*. Wiley, 1989.
- [2] D. J. Abel and J. L. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Comput. Vis.*, 24:1-13, 1983.
- [3] A. V. Aho and J. D. Ullman. Optimal partial-match retrieval when fields are independently specified. *ACM Transactions on Database Systems*, 4(2):168-179, June 1979.
- [4] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1(3):173-189, 1972.
- [5] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta Inf.*, pages 1-21, 1977.
- [6] B. Becker, P. FrancLosa, S. Gschwind, T. Ohler, F. Thiem, and P. Widmayer. Enclosing many boxes by an optimal pair of boxes. In *Proceedings of STACS92*, pages 475-486, 1992. A. Finkel and M. Jantzen, Eds., LNCS 525, Springer-Verlag, Berlin/Heidelberg/New York.

- [7] L. Becker. *A new algorithm and a cost model for join processing with the grid file*. PhD thesis, Universitiit-Gesamthochschule, Siegen, Germany, 1992.
- [8] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Sfege. The r\*-tree: An efficient and robust access method for points and rectangles. In *ACM SIGMOD International Conference on Management of Data*, pages 322-331, 1990.
- [9] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509-517, 1975.
- [10] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397-409, 1979.
- [11] S. Berchtold, D. Keim, and H.-P. Kriegel. The x-tree: An index structure for high-dimensional data. In *The 22nd International Conference on Very Large Data Bases*, pages 168-179, 1996.
- [12] D. Bitton, H. Boral, D. J. DeWitt, and W. K. Wilkinson. Parallel algorithms for the execution of relational database operations. *ACM Transactions on Database Systems*, 8(3):324-353, September 1983.
- [13] M. W. Blasgen and K. P. Eswaran. Storage and access in relational database. *IBM Systems*, 16(4):105-115, 1977.
- [14] K. Bratbergesen. Hashing methods and relational algebra operations. In *Proceedings of the 10th VLDB Conference*, pages 323-333, August 1984.

- [15] W. A. Burkhard. Interpolation-based index maintenance. *BIT*, 23:274-294, 1983.
- [16] W. A. Burkhard. Index maintenance for non-uniform record distribution. In *The Third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 173-180, 1984.
- [17] C. Y. Chen, C. C. Chang, and R. C. Lee. Optimal MMI file systems for orthogonal range queries. *Information Systems*, 18:37-54, 1993.
- [18] L. Chen, R. Drach, M. Keating, S. Louis, D. Rotem, and A. Shoshani. Access to multidimensional datasets on tertiary storage systems. *Inf Syst.*, 20(2):155-183, 1995.
- [19] S. Christodoulakis. Estimating block transfer and join size. In *Proceedings of SIGMOD. CODASYL, 1971. Database Task Group Report.*, pages 105-115, 1985.
- [20] D. Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11(2):121-138, 1979.
- [21] H. Dang and D. Abramson. Cooling schedules for simulated annealing based on scheduling algorithms. In *Proceedings of the 17th Annual Computer Science Conference*, pages 541-550, Christchurch, New Zealand, January 1994.
- [22] B. P. Desai. Performance of a composite attribute and join index. *IEEE Trans. on Software Engineering*, SE-15(2):143-152, February 1989.

- [23] D. J. DeWitt and J. Gary. Parallel database systems: the future of database processing or passing fad. *SIGMOD rec.*, 19(4):104-112, December 1990.
- [24] D. J. DeWitt, R. H. Katz, L. D. Olken, R. Shapiro, Stonebraker, and Wood D. Implementation technique for main memory database systems. pages 1-8, Boston, Massachusetts, USA, June 1984.
- [25] R. Elmasri and S. B. Navathe. *Fundamentals of database systems*. Benjamin/Cummings Publishing Co., Redwood City, California, 1994.
- [26] M. Ester, J. Kohlhammer, and H. Kriegel. T-dc-tree: A fully dynamic index structure for data warehouses. In *icde*, pages 379-388, 2000.
- [27] G. Evangelidis, D. Lomet, and B. Salzberg. The hb-tree: A modified hb-tree supporting concurrency, recovery and node consolidation. In *The 21st International Conference on Very Large Data Bases*, pages 551-561, 1995.
- [28] G. Evangelidis. *The hB-tree: A concurrent and recoverable multi-attribute index structure*. PhD thesis, Northeastern University, Boston, MA., 1994.
- [29] R. Fagin, J. Nievergelt, N. Pippenger, and R. Strong. Extendible hashing: A fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315-344, 1979.
- [30] C. Faloutsos. Multiattribute hashing using gray-codes. In *The ACM SIGMOD International Conference on Management of data*, pages 227-238, 1986.

- [31] G. Faloutsos. Gray-codes for partial match and range queries. *IEEE Trans. Softw. Eng.*, 14:1381-1393, 1988.
- [32] G. Faloutsos and Y. Rong. Dot: A spatial access method using fractals. In *The Seventh IEEE International Conference on Data Engineering*, pages 152-159, 1991.
- [33] G. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *The Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 247-252, 1989.
- [34] R. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval of composite keys. *Acta Inf.*, 4(1):1-9, 1974.
- [35] S. Finkelstein. Physical database design for relational databases. *ACM Transactions on Database Systems*, 13(1):91-128, March 1988.
- [36] M. Freeston. The bang file: A new kind of grid file. In U. Dayal and I. Traiger, editors, *The ACM SIGMOD International Conference on Management of Data*, pages 260-269, San Francisco, California, USA, May 1987.
- [37] M. Freeston. Advances in the design of the bang file. In *In the proceeding of the Third International Conference on Foundations of Data Organization and Algorithms*, pages 322-338, 1989.
- [38] M. Freeston. On the complexity of bv-tree updates. In *The CDB97 and CP'96 Workshop on Constraint Databases and their Application*, V. Gaede, A. Brodsky, O. Gunther, D. Srivastava, V. Vianu,

and M. Wallace, Eds., pages 282-293, LNCS 1191, Springer-Verlag, Berlin/Heidelberg/New York, 1997.

- [39] H. Fuchs, G. D. Abram, and F. D. Grant. Near real-time shaded display of rigid objects. *Computer Graph*, 17(3):65-72, 1983.
- [40] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Computer Graph*, 14(3), 1980.
- [41] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(1):170-230, June 1998.
- [42] A. K. Garg and C. C. Gotlieb. Order preserving key transformation. *ACM Trans. Database Syst.*, 11(2):213-234, 1986.
- [43] F. Glover. Tabu search: a tutorial. *Interface*, 20(4):74-94, 1990.
- [44] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley, Reading, Massachusetts, USA, 1989.
- [45] J. R. Goodman. An investigation of multiprocessor structures and algorithms for database management. Technical Report UCB/ERL, M81/33, University of California, Berkly, 1981.
- [46] P. Goyal, H. Li, E. Regener, and F. Sadri. Scheduling of page fetches in join operations using bc-trees. In *In Proceedings of Conference of Data Engineering*, pages 304-310, 1988.
- [47] O. Gunther. The cell tree: An object-oriented index structure for geometric databases. In *The Fifth IEEE International Conference on Data Engineering*, pages 598-605, 1989.

- [48] O. Gunther and A. Buchmann. Research issues in spatial databases. *SIGMOD Rec.*, 19(4):61-68, 1990.
- [49] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on the Management of Data*, pages 47-54, Boston, 1984.
- [50] L. Harada, M. Nakano, M. Kitsuregawa, and M. Takagi. Query processing method for multi-attribute clustered relations. In *Proceedings of the 16th VLDB Conference*, pages 59-70, Brisbane, Australia, August 1990.
- [51] E. P. Harris. *Towards optimal storage design for efficient query processing in relational database systems*. PhD thesis, The University of Melbourne, Melbourne, Australia, 1994.
- [52] E. P. Harris and K. Ramamohanarao. Optimal dynamic multi-attribute hashing for range queries. *BIT*, 33(4):561-579, 1993.
- [53] E. P. Harris and K. Ramamohanarao. Optimal clustering of relations to improve sorting and partitioning for joins. *The Computer Journal*, 40(7):416-434, 1997.
- [54] E. P. Harris and K. Ramamohanarao. Generalising minimal marginal increase to cluster records in multi-dimensional data files. In John Roddick, editor, *Database Systems '1999, Proceedings of the 10th Australasian Database Conference*, pages 129-140, Auckland, New Zealand, January 1999. Springer.

- [55] A. Henrich, H.-W. Six, and P. Widmayer. The lsd tree: Spatial access to multidimensional point and non-point objects. In *The 15th International Conference on Very Large Data Bases*, pages 45-53, 1989.
- [56] K. Hinrichs. Implementation of the grid file: Design concepts and experience. *BIT*, 25:569-592, 1985.
- [57] A. Hutflesz, H.-W. Six, and P. Widmayer. Globally order preserving multidimensional linear hashing. In *The Fourth IEEE International Conference on Data Engineering*, pages 572-579, 1988.
- [58] A. Hutflesz, H.-W. Six, and P. Widmayer. Twin grid files: Space optimizing access schemes. In *The ACM SIGMOD International Conference on Management of Data*, pages 183-190, 1988.
- [59] L. Ingber and B. Rosen. Genetic algorithms and very fast simulated annealing: a comparison. *Mathematical and Computer Modelling*, 16(11):87-100, 1992.
- [60] Y. E. Ioannidis and Y. C. Kang. Randomized algorithms for optimizing large join queries. In *Proceedings of the 1990 ACM SIGMOD International Conference on the Management of Data*, pages 312-321, Atlantic city, New Jersey, USA, May 1990.
- [61] H. V. Jagadish. Linear clustering of objects with multiple attributes. In *The ACM SIGMOD International Conference on Management of Data*, pages 332-342, 1990.
- [62] H. V. Jagadish. On indexing line segments. In *The Sixteenth International Conference on Very Large Data Bases*, pages 614-625, 1990.



- [63] H. V. Jagadish. Spatial search with polyhedra. In *The Sixth IEEE International Conference on Data Engineering*, pages 311-319, 1990.
- [64] I. Kamel and C. Faloutsos. Parallel r-trees. In *The ACM SIGMOD International Conference on Management of Data*, pages 195-204, 1992.
- [65] I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *The Twentieth International Conference on Very Large Data Bases*, pages 600-509, 1994.
- [66] A. Kemper and M. Wallrath. An analysis of geometric modeling in database systems. *ACM Computing Surveys*, 19(1):47-91, 1987.
- [67] W. Kim. A new way to compute the product and join of relations. In *Proceedings of the 1980 ACM SIGMOD International Conference on the Management of Data*, pages 179-187, 1980.
- [68] M. Kitsuregawa, M. Nakayama, and M. Takagi. The effect of bucket size tuning in the dynamic hybrid grace hash join methods. In *Proceedings of the 15th VLDB Conference*, pages 257-266, Amestardam, The Netherlands, August 1989.
- [69] G. D. Knott. Hashing functions. *Comput. J.*, 18(3):265-278, 1975.
- [70] D. E. Knuth. *Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, USA, 1973. Volume 3 of The Art of Computer Programming.
- [71] H.-P. Kriegel and B. Seeger. Multidimensional order preserving linear hashing with partial expansions. In *In Proceedings of the Interna-*

*tional Conference on Database Theory, LNCS 243, Springer-Verlag, Berlin/Heidelberg/NewYork, 1986.*

- [72] H.-P. Kriegel and B. Seeger. Multidimensional quantile hashing is very efficient for non-uniform record distributions. In *In Proceedings of the Third IEEE International Conference on Data Engineering*, pages 10-17, 1987.
- [73] H.-P. Kriegel and B. Seeger. Multidimensional quantile hashing is very efficient for non-uniform distributions. *Inf. Sci.*, 48:99-117, 1989.
- [74] M. Kriegel, H.-P. and Schiwietz, R. Schneider, and B. Seeger. Performance comparison of point and spatial access methods. pages 89-114, 1990.
- [75] H.-P. Kriegel and B. Seeger. Plop-hashing: A grid file without directory. In *The Fourth IEEE International Conference on Data Engineering*, pages 369-376, 1988.
- [76] A. Kumar. G-tree: A new data structure for organizing multidimensional data. *IEEE Trans. Knowl. Data Eng.*, 6(2):341-347, 1994.
- [77] R. S. G. Lancelotte, P. Valduriez, and M. Zait. On effectiveness of optimisation search strategies for parallel execution spaces. In *Proceedings of the 19th VLDB Conference*, pages 493-504, Dublin, Ireland, August 1993.
- [78] P. A. Larson. Linear hashing with partial expansions. In *Proceedings of the 6th VLDB Conference*, pages 224-232, 1980.

- [79] J.-H. Lee, Y.-K. Lee, K.-Y. Whang, and I.-Y. Song. A region splitting strategy for physical database design of multidimensional file organizations. In *Proceedings of the 23rd VLDB Conference*, pages 416-425, Athens, Greece, August 1997.
- [80] J. Li, D. Rotem, and J. Srivastava. Algorithms for loading parallel grid files. In *Proceedings of the 1993 ACM SIGMOD International Conference on the Management of Data*, pages 347-356, Washington, DC, USA, May 1993.
- [81] W. Litwin. Linear hashing: A new tool for file and table addressing. In *The Sixth International Conference on Very Large Data Bases*, pages 212-223, Monterial, Canada, August 1980.
- [82] W. Litwin and M.-A. Neimat. Distributed linear hashing. Technical memo HPL-DTD-92-7, Hewlett Packard, 7 1992.
- [83] J. W. Lloyd. Optimal partial-match retrieval. *BIT*, 20:406-413, 1980.
- [34] J. W. Lloyd and Ramamoharao K. Partial-match retrieval for dynamic files. *BIT*, 22:156-168, 1982.
- [85] D. B. Lomet and B. Salzberg. The hb-tree: A robust multiattribute search structure. In *In Proceedings of the Fifth IEEE International Conference on Data Engineering*, pages 296-304, 1989.
- [86] D. B. Lomet and B. Salzberg. The hb-tree. a multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):38-71, 1990.

- [87] H. Lu and B.-C. Ooi. Spatial indexing: Past and future. *IEEE Data Eng. Bull.*, 16(3):16-21, 1993.
- [88] Z. Michalewicz. *Genetic algorithms + data structures = evolution programs*. Springer-Verlag, 1992.
- [89] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63-113, March 1992.
- [90] S. Mohammed, E. P. Harris, and K. Ramamohanarao. Efficient partial-match retrieval for skewed data distributions. In John Roddick, editor, *Database Systems '1999, Proceedings of the 10th Australasian Database Conference*, pages 37-48, Auckland, New Zealand, January 1999. Springer.
- [91] S. Mohammed, E. P. Harris, and K. Ramamohanarao. Efficient range query retrieval for non-uniform data distributions, January 2000.
- [92] S. Mohammed and B. Srinivasan. A novel parallel algorithms for grid files. In *IEEE, 3rd International Conf. on High Performance Computing*, pages 31-40, Trivandrum, India, December 1996.
- [93] S. Mohammed and B. Srinivasan. Efficient parallel join algorithms for multidimensional files. In *Robotics, Vision and Parallel Processing for Industrial Automation Conf.*, pages 141-151, Ipoh, Malaysia, November 1997.
- [94] S. Moran. On the complexity of designing optimal partial-match retrieval systems. *ACM Transactions on Database Systems*, 8(4):543-551, December 1983.

- [95] G. Morton. A computer oriented geodetic database and a new technique in file sequencing. *IBM Ltd.*, 1966.
- [96] M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-partitioned join method using dynamic destaging strategy. In *Proceedings of the 15th VLDB Conference*, pages 468-478, Los Angeles, California, USA, August 1988.
- [97] J. Nievergelt, H. Hinterberger, and K. Sevcik. The grid file: An adaptable, symmetric multikey file structure. In *The Third ECI Conference*, A. Duijvestijn and P. Lockemann, Eds., LNCS 123, Springer-Verlag, Berlin/Heidelberg/New York, pages 236-251, 1981.
- [98] J. Nievergelt, H. Hinterberger, and K. C. Seycik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38-71, 1984.
- [99] K. J. Nurmela. Constructing combinatorial designs by local search. Department of Computer Science, Digital Systems Laboratory A-27, Helsinki University, Finland, November 1993.
- [100] E. Omiecinski. Heuristics for join processing using non clustered index. *IEEE Trans. on Software Engineering*, 15(1):18-25, January 1989.
- [101] B. C. Ooi, K. J. Medonell, , and R. Sacks-davis. Spatial kd-tree: An indexing mechanism for spatial databases. In *The IEEE Computer Software and Applications Conference*, pages 433-438, 1987.
- [102] P. Oosterom. *Reactive-data structures for geographic information systems*. PhD thesis, University of Leiden, The Netherlands, 1990.

- [103] J. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *The Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 181-190, 1984.
- [104] J. A. Orenstein. A dynamic hash file for random and sequential access. In *Proceedings of the 9th VLDB Conference*, pages 132-141, Florence, Italy, November 1983.
- [105] Y. Oshawa and M. Sakauchi. A new tree type data structure with homogeneous node suitable for a very large spatial database. In *The Sixth IEEE International Conference on Data Engineering*, pages 296-303, 1990.
- [106] E. J. Otoo. Symmetric dynamic index maintenance scheme. In *The International Conference on Foundations of Data Organization*, Plenum, New York, pages 283-296, 1985.
- [107] E. J. Otoo. Balanced multidimensional extendible hash tree. In *Proceeding of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 100-113, 1986.
- [108] M. Ouksel. The interpolation based grid file. In *The Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 20-27, 1985.
- [109] M. Ouksel and P. Scheuermann. Storage mappings for multidimensional linear dynamic hashing. In *The Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 90-105, 1982.

- [110] M. A. Ouksel and O. Mayer. A robust and efficient spatial data structure. *Acta Informatica*, 29:335-373, 1992.
- [111] E. A. Ozkarahan and M. Ouksel. Dynamic and order preserving data partitioning for database machines. In *Proceedings of the 11th VLDB Conference*, pages 90-105, 1983.
- [112] H. Pang, M. J. Carey, and M. Livny. Partially preemptive hash joins. In *Proceedings of the 1993 ACM SIGMOD International Conference on the Management of Data*, pages 59-68, Washington, DC, USA, November 1993.
- [113] W. Perrizo, J. Y Lin, and W. Hoffman. Algorithms for distributed query processing in broadcast local area network. *IEEE Trans. on Knowledge and Data Engineering*, 1(2):215-225, June 1989.
- [114] K. Ramamohanarao and J. W. Lloyd. Dynamic hashing schemes. *The Computer Journal*, 25:478-485, 1982.
- [115] K. Ramamohanarao and R. Sacks-Davis. Recursive linear hashing. *ACM Transactions on Database Systems*, 8(9):369-391, September 1984.
- [116] K. Ramamohanarao, J. Shepherd, and R. Sacks-Davis. Multi-attribute hashing with multiple file copies for high performance partial-match retrieval. *BIT*, 30:404-423, 1990.
- [117] M. Regnier. Analysis of the grid file algorithms. *BIT*, 25:335-357, 1985.

- [118] J. P. Richardson, H. Lu, and K. Mikkilineni. Design and evaluation of parallel pipelined join algorithms. pages 399-409, Sanfrancisco, California, May 1987.
- [119] J. T. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD International Conference on the Management of Data*, pages 10-18, 1981.
- [120] J. B. Rosenberg. Geographical data structures compared: A study of data structures supporting region queries. *IEEE Trans. on Computer-aided Design*, CAD-4(1):53-67, January 1985.
- [121] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed r-trees. In *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 17-31, 1985.
- [122] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA., 1990.
- [123] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared nothing multi-processor environment. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, pages 110-121, Portland, Oregon, 1989.
- [124] R. Schneider and H.-P. Kriegel. The tr\*-tree: A new representation of polygonal objects supporting spatial queries and operations. In



*In Proceedings of the Seventh Workshop on Computational Geometry, LNCS 553, Springer-Verlag, Berlin/Heidelberg/New York, pages 249-264, 1992.*

- [125] B. Seeger. Performance comparison of segment access methods implemented on top of the buddy-tree. In *In Advances in Spatial Databases, O. Gunther and H. Schek, Eds., LNCS 525, Springer-Verlag, Berlin/Heidelberg/New York, pages 277-296, 1991.*
- [126] T. Sellis, N. Roussopoulos, and C. Faloutsos. The r+-tree. a dynamic index for multidimensional objects. In *In Proceedings of the Thirteenth International Conference on Very Large Data Bases, pages 507-518, 1987.*
- [127] K. Sevcik and N. Koudas. Filter trees for managing spatial data over a range of size granularities. In *In Proceedings of the 22th International Conference on Very Large Data Bases, pages 16-27, Bombay, India, 1996.*
- [128] L. D. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3):239-264, September 1986.
- [129] A. Shatdal and J. F. Naughton. Using shared virtual memory for parallel join processing. In *Proceedings of the 1993 ACM SIGMOD International Conference on the Management of Data, pages 119-128, Washington, DC, USA, May 1993.*

- [130] S. Shekhar and D.-R. Liu. Ccam: A connectivity-clustered access method for aggregate queries on transportation networks: A summary of results. In *Proceedings of the Eleventh IEEE International Conference on Data Engineering*, pages 410-419, 1995.
- [131] R. E. Smith, D. E. Goldberg, and J. A. Earickson. Sga-c: A c-language implementation of a simple genetic algorithm. Technical Report 91002, The Clearinghouse for Genetic Algorithms, Department of Engineering Mechanics, The University of Alabama, Tuscaloosa, Alabama, USA, May 1991.
- [132] S. Y. W. Su. *Database Computers: Principles, Architectures, and Techniques*. McGraw-hill, New York, 1988.
- [133] A. Swami. Optimization of large join queries: combining heuristics and combinatorial techniques. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, pages 367-376, Portland, Oregon, USA, June 1989.
- [134] M. Tamminen. The extendible cell method for closest point problems. *BIT*, 22:27-41, 1982.
- [135] J. A. Thom and L. Ramamohanarao, K. Naish. A superjoin algorithm for deductive databases. In *Proceedings of the 12th VLDB Conference*, pages 189-196, Koyoto, Japan, August 1986.
- [136] J. D. Ullman. *Principles of database and knowledge-base systems*, volume 1. Computer Science Press, Rockville, Maryland, USA, 1988.

- [137] J. D. Ullman. *Principles of database and knowledge-base systems*, volume 2. Computer Science Press, Rockville, Maryland, USA, 1989.
- [138] C. B. Walton, A. G. Dale, and R. M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proceedings of the 17th VLDB Conference*, pages 537-548, Barcelona, Spain, September 1991.
- [139] K.-Y. Whang and Krishnamurthy R. Multilevel grid files. Technical report, IBM Thomson J. Research Center, November 1985. IBM Research Report RC 11516.
- [140] K.-Y. Whang and Krishnamurthy R. The multilevel grid file — a dynamic hierarchical multidimensional file structure. In *International Symposium on Database Systems for Advanced Applications*, pages 449-459, Tokyo, Japan, April 1991.
- [141] M. White. N-trees: Large ordered indexes for multidimensional space. Technical report, Statistical Research Division, US Bureau of the Census, 1981. Application Mathematics Research Staff.
- [142] H. Yoo and S. Lafortune. An intelligent search method for query optimisation by semi-joins. *IEEE Trans. on Knowledge and Data Engineering*, 1(2):226-237, June 1989.
- [143] C.T. Yu and et al. Adaptive record clustering. *ACM Transactions on Database Systems*, 10(2):180-204, June 1985.

- [144] H. Zeller and J. Gray. An adaptive hash join algorithms for multiuser environments. In *Proceedings of the 16th VLDB Conference*, pages 186-197, Brisbane, Australia, August 1990.