Multi-Dimensional Bitmap Indices for Optimising Data Access within Object Oriented Databases at CERN

eingereicht von:
Mag. Kurt Stockinger

Dissertation

zur Erlangung des akademischen Grades
Doctor rerum socialium oeconomicarumque
(Dr. rer. soc. oec.)
Doktor der Sozial- und Wirtschaftswissenschaften

Fakultät für Wirtschaftswissenschaften und Informatik
Universität Wien

Erstgutachter: ao.Univ.-Prof. Dr. Erich Schikuta
Zweitgutachter: Univ.-Prof. Dr. Dr. Gerald Quirchmayr
CERN Betreuer: Dr. Dirk Düllmann

Genf, im November 2001

To Sabinka

Contents

1 Introduction **9**
- 1.1 Multi-Petabyte Data Challenge at CERN . . . 10
- 1.2 HEP Data Model . . . 10
- 1.3 Physics Analysis . . . 11
- 1.4 Contribution of the Thesis . . . 12

2 "Conventional" Multi-dimensional Access Methods **14**
- 2.1 Introduction . . . 14
- 2.2 Problem Definition . . . 14
- 2.3 Basic Data Structures . . . 15
- 2.4 Point Access Methods (PAMs) . . . 16
  - 2.4.1 Multi-Dimensional Hashing . . . 16
  - 2.4.2 Hierarchical Access Methods . . . 17
- 2.5 Spatial Access Methods (SAMs) . . . 17
  - 2.5.1 Overlapping Regions . . . 18
  - 2.5.2 Clipping . . . 21
  - 2.5.3 Transformation . . . 22
  - 2.5.4 Multiple Layers . . . 24
- 2.6 Pyramid-tree . . . 24
  - 2.6.1 Treatment of Data Space . . . 25
  - 2.6.2 Index Creation and Query Processing . . . 26

3 Bitmap Indices **28**
- 3.1 Introduction . . . 28
- 3.2 Simple Bitmap Indices . . . 28
  - 3.2.1 Space Complexity . . . 29
  - 3.2.2 Time Complexity . . . 30
  - 3.2.3 Pros and Cons of Simple Bitmap Indices . . . 30
- 3.3 Equality, Range, Interval Encoding . . . 31
- 3.4 Range-Based Indices . . . 31
- 3.5 Encoded Bitmap Indices . . . 33
- 3.6 Miscellaneous Techniques . . . 34
- 3.7 Bitmap Compression . . . 34

3.7.1 LZ Encoding . . . 35
3.7.2 ExpGol Encoding . . . 35
3.7.3 Byte-Aligned Bitmap Codes . . . 36

4 Current Access Methods for HEP Queries **39**
4.1 Introduction . . . 39
4.2 Implementation Issues . . . 40
4.3 Performance Analysis . . . 41
4.4 Conclusions . . . 43

5 Bitmap Indices for Scientific Data **44**
5.1 Introduction . . . 44
5.2 Bitmap Indices for HEP . . . 44
5.3 Implementation on Objectivity/DB . . . 45
5.4 Brief Justification of the Bitmap Index Approach . . . 46
5.4.1 Cost Model for Equality Encoded Bitmap Indices . . . 47
5.4.2 Size of the Index . . . 48
5.4.3 I/O Complexity of the Index . . . 49
5.4.4 Maximal Page I/O Costs for Index Evaluation Phase . . . 49
5.4.5 Page I/O Costs for Candidate Check Phase . . . 50
5.4.6 Total I/O Costs . . . 50
5.5 Analytical Results . . . 53
5.5.1 Equality Encoding on Generic Tags . . . 53
5.5.2 Equality Encoding on Sliced Tags . . . 55
5.5.3 Comparison - Equality Encoding Generic Tags vs. Sliced Tags . . . 57
5.6 Partitioned Range Encoding . . . 57
5.7 Conclusions . . . 60

6 Generic Range Encoding **61**
6.1 Introduction . . . 61
6.2 Example: Range Encoding for Non-Discrete Attribute Values . . . 61
6.3 Generic Range Encoding - A Novel Algorithm . . . 63
6.4 Cost Model for `GenericRangeEncoded` Bitmap Indices . . . 68
6.4.1 I/O Complexity of the Index . . . 68
6.4.2 Page I/O Costs for Index Evaluation Phase . . . 69
6.4.3 Page I/O-Costs for Candidate Check Phase . . . 69
6.5 Analytical Results . . . 71
6.6 Experimental Results . . . 74
6.7 Analytical vs. Experimental Results . . . 77
6.8 Conclusions . . . 79

7 Advanced Features **80**
7.1 Introduction . . . 80
7.2 Adaptive Index . . . 80
7.3 Binning Strategies . . . 81
7.4 Bitmap Compression on Uniformly Distributed Data . . . 81
7.4.1 Equality Encoded Bitmap Index . . . 81
7.4.2 Range Encoded Bitmap Index . . . 83
7.4.3 Compressed Equality Encoding vs. Verbatim Range Encoding . . . 84
7.5 Bitmap Compression on Non-Uniformly Distributed Data . . . 86
7.6 Conclusions . . . 90

8 Applications **93**
8.1 Introduction . . . 93
8.2 Extended Cost Model . . . 93
8.2.1 Clustering . . . 93
8.2.2 Correlation . . . 94
8.3 Query Parser . . . 94
8.4 High Energy Physics . . . 95
8.4.1 Data Distribution . . . 95
8.4.2 Sample Queries . . . 96
8.4.3 Comparison with the Cost Model . . . 99
8.4.4 Bitmap Compression . . . 100
8.5 Astronomy . . . 101
8.5.1 Data Preparation . . . 102
8.5.2 "Typical" Astronomy Sample Queries . . . 103
8.5.3 Analysis of the Results . . . 105
8.6 Conclusions . . . 107

9 Outlook - Query Optimisation in a Grid Environment **108**
9.1 Introduction . . . 108
9.2 The EU DataGrid . . . 108
9.3 Optimisation Opportunities for Analysis . . . 108
9.4 Grid Query Optimisation for Analysis within a Typical Grid Architecture . . . 111
9.4.1 The Local Application Layer . . . 112
9.4.2 The Grid Application Layer . . . 112
9.4.3 The Collective Services Layer . . . 113
9.4.4 The Task of Grid Query Optimisation within a Typical Architecture . . . 114
9.5 Interaction of Services for a Particular HEP Use Case . . . 115
9.6 Conclusions and Future Work . . . 117

10 Conclusions **118**

11 Acknowledgements **120**

List of Figures

1.1 Particle traces after event collision. 12

2.1 R-tree. 19
2.2 R*-tree. 21
2.3 X-tree. 22
2.4 Various shapes of the X-tree in different dimensions. 22
2.5 R+-tree. 23
2.6 Hilbert Curves of order 1, 2 and 3. 24
2.7 Hilbert R-tree. 25
2.8 Space partitioning method of the Pyramid-tree. 26
2.9 Key values for the Pyramid-tree. 26
2.10 Query processing with the Pyramid-tree. 27

3.1 Algorithm `RangeEval` and `RangeEval-Opt`. 33
3.2 Hufmann encoded bitmap index. 34
3.3 Compression ratios of various bitmap compression algorithms. Bit desity (x-axis) vs. compression ratio (y-axis). 37
3.4 Time for compression and uncompression of various bitmap compression algorithms. 37
3.5 Compression algorithm with best performance on Boolean operations. 38

4.1 Response times for selecting attributes based on various different tag implementations. a) 81,060 tags with 302 attributes b) 1,000,000 tags with 25 attributes. 42
4.2 Access patterns of disk head movements for reading parallel streams a) without prefetch optimsation b) with prefetch optimisation. 42

5.1 One sided-range query on a range encoded bitmap index. 45
5.2 Architectural overview of the bitmap index on top of Objectivity/DB. 46
5.3 Variable query selectivities for two sided-range queries. 48
5.4 Access granularity of a database in terms of pages rather than objects. 50
5.5 Generic tag - I/O costs for optimal number of bins for 50% attribute selectivity (worst case) compared to sequential scan. 54
5.6 Generic tag - I/O costs for optimal number of bins for 50% attribute selectivity (worst case) are split into index-I/O and candidate-I/O. 55

5.7 Generic tag - I/O costs for multiple attribute selectivities compared to sequential scan. . . . 56
5.8 Sliced tag - I/O costs for optimal number of bins for 50% attribute selectivity (worst case) compared to sequential scan. . . . 57
5.9 Sliced tag - I/O costs for optimal number of bins for 50% attribute selectivity (worst case) are split into index-I/O and candidate-I/O. . . . 58
5.10 Sliced tag - I/O costs for multiple attribute selectivities compared to sequential scan. . . . 59
5.11 Partitioned Equality Encoding (EQ) vs. Partitioned Range Encoding (Range). 60

6.1 One-sided range query on a range encoded bitmap index. . . . 62
6.2 Candidate check in multi-dimensional space. For each attribute the candidates are checked separately. . . . 65
6.3 Analytical results: Page I/O for queries over multiple dimensions with various selectivities. 10 bins. . . . 72
6.4 Analytical results: Page I/O for queries over multiple dimensions with various selectivities. 100 bins. . . . 73
6.5 Analytical results: Page I/O for queries over multiple dimensions with various selectivities. 1,000 bins. . . . 74
6.6 Analytical results: Page I/O for queries over multiple dimensions with various selectivities. 1,000,000 bins. . . . 75
6.7 Experimental results: Page I/O and response for queries over multiple dimensions with various selectivities - one to three dimensions. . . . 77
6.8 Experimental results: Page I/O and response for queries over multiple dimensions with various selectivities - five to 25 dimensions. . . . 78

7.1 Query response time of verbatim vs. compressed equality encoded bitmap index - 100 bins. . . . 82
7.2 Query response time of verbatim vs. compressed equality encoded bitmap index - 1000 bins. . . . 83
7.3 Compression ratio of range encoded bitmap index with 100 bins. . . . 84
7.4 Query response time for compressed equality encoding vs. verbatim range encoding. . . . 85
7.5 Query response time for compressed equality encoding vs. verbatim range encoding. . . . 86
7.6 Range encoded bitmap index based on data following an exponential distribution. 87
7.7 Response time for verbatim vs. compressed bitmap indices. Queries include the "<"-operator. . . . 88
7.8 Response time for verbatim vs. compressed bitmap indices. Queries include the ">"-operator. . . . 89
7.9 Response time for compressed bitmap indices. Queries including "<"-operator vs. queries including ">"-operator. . . . 90
7.10 Response time for queries based on compressed bitmap index with 100 bins vs. 400 bins. Queries include the "<"-operator. . . . 91

7.11 Response time for verbatim vs. compressed bitmap indices on "tier2". 92

8.1 Effect of equi-width binning for the attributes `jet1E` and `jet2E`. 96
8.2 Effect of equi-width binning for the attributes `jet1Theta` and `jet1Phi`. 97
8.3 Compression ratios for various attributes based on range encoded bitmap index with 100 bins.The compression algorithm is two-sided byte-aligned bitmap compression. 101
8.4 Response times for queries `Q8` to `Q13` based on verbatim vs. compressed bitmap indices. 102
8.5 Response times for 10 dimensional queries based on verbatim vs. compressed bitmap indices. 103
8.6 NGC 5792, a highly inclined spiral galaxy. 104
8.7 Effect of equi-width binning for the attributes `ra` and `dec`. 105
8.8 Effect of equi-width binning for the attributes `r`, `i` and `g`. 106

9.1 Workpackages within the EU DataGrid. 109
9.2 Grid Architecture from the point of view of "Grid Query Optimisation". 111

List of Tables

1.1 Data Model for High Energy Physics. 11

3.1 Bitmap Indices for Stock Trading. 29
3.2 a) Projection Index π_A and b) Equality Encoding E^i. 31
3.3 a) Projection Index π_A, b) Range Encoding R^i and c) Interval Encoding I^i. . . 32

5.1 Sequential scan vs. bitmap index. 47
5.2 Parameters of the cost model. 51

6.1 Parameters for algorithm `GenericRangeEval`. 63
6.2 Parameters of the cost model. 69
6.3 Effect of number of bins on the number of candidate objects for one dimension. 70
6.4 Response time for sequential scan over 10^6 objects with various number of attributes (dimensions) - attribute-wise clustering. 76

8.1 Response time for sequential scan over 1,401,020 objects with various number of attributes (dimensions) - attribute-wise clustering. 95
8.2 Attribute ranges of queried attributes. 98
8.3 Number of pages access and query response time for 13 sample queries. 99
8.4 Number of pages access and query response time for 13 sample queries. 100
8.5 Response time for sequential scan over 6,182,527 objects with various number of attributes (dimensions) - attribute-wise clustering. 104

Chapter 1

Introduction

Over the last three decades many different index data structures were proposed to optimise the access to database management systems. Some of these are optimised for one-dimensional queries such as the B+tree [3] whereas others are optimised for multi-dimensional queries such as the Pyramid-tree [5] or bitmap indices [50]. Recently, especially bitmap indices have become popular access methods for data warehouse applications and decision support systems with large amounts of read-mostly data.

The basic idea of a bitmap index is to store one vector of bits per distinct attribute value (e.g. possible attribute values are *colours*). Each bit of the value is mapped to a record. The associated bit is set if and only if the record's value fulfils the property in focus (e.g. the respective value of the record is equal to *red*).

Bitmaps indices efficiently support complex, multi-dimensional queries. These data structures are also implemented in commercial database management systems such as Oracle, Sybase or Informics. All these implementations are optimised for typical business applications which are characterised by discrete attribute values. However, scientific data which is mostly characterised by non-discrete attribute values, cannot be handled efficiently by these kind of data structures.

In the literature on multi-dimensional index data structures one will often encounter the words "curse of dimensionality". According to [8] this expression refers to the exponential growth of hypervolume as a function of dimensionality. In the fields of multi-dimensional access methods these words refer to the degeneration of conventional access methods in multi-dimensional search spaces. In short, it is argued that in many cases the sequential scan over the base data is more efficient than an indexed query. The main task of this Ph.D. is to design, analyse and implement a novel access method based on bitmap indices for speeding up multi-dimensional queries. We thus want to demonstrate analytically and experimentally that our access method shows good performance behaviour in multi-dimensional search spaces and significantly outperforms the sequential scan.

1.1 Multi-Petabyte Data Challenge at CERN

This Ph.D. thesis was performed at CERN, the European Organization for Nuclear Research [12] in Geneva, Switzerland which is commonly known as the birthplace of the web [26]. However, the main goal of CERN is to study the fundamental structure of matter and the interaction of forces. In particular, sub-atomic particles are accelerated to nearly the speed of light and then collided. Such collisions are called *events* and are measured at time intervals of only 25 nanoseconds in four different particle detectors of the Large Hadron Collider (LHC) - CERN's "next generation" accelerator which starts data taking in 2006. According to [42] each of the 4 main experiments will produce around 1 Petabyte of data a year over a life span of about two decades. This data will be analysed by some 5,000 physicists around the world.

Since CERN experiments are collaborations of over a thousand physicists from many different universities and institutes, the experiments' data is not only stored locally at CERN but is distributed world wide in so-called Regional Centres (RCs), in national institutes and universities. This complex distributed computing infrastructure is a typical example of a Grid environment as defined in [20]. In order to handle the huge data challenge of managing Petabytes of data distributed around the globe, CERN has established the DataGrid Project [18] which is a multi-disciplinary project containing three different application areas, namely Bioinformatics, Earth Observation and High Energy Physics.

1.2 HEP Data Model

In the HEP community, a single data set is a unit of related data objects and is called an *event* which in turn includes different types of hierarchical data. Initially, data which is produced by the detectors is filtered by dedicated hardware and software and only "interesting" data is stored. This data is called *raw data*. A complete set of the raw data will be stored in large storage devices (probably tapes and disk pools) at CERN. Other copies of the raw data, will be maintained at Tier 1 sites (of which there should be around 5 around the world). The size of a typical raw data event is 1 MB.

In the next step a reconstruction function is used to bring more structure into the raw data. The result of the reconstruction process is so-called *reconstructed data* which can further be split into *Event Summary Data (ESD)* and *Analysis Object Data (AOD)*. This data contains information about particle tracks in the detector, energy values and other physics data. Similar to the raw data, also this data is replicated over multiple distributed sites. However, the object hierarchy is much more complex than the one for raw data and can reach up 1,000 inherited classes.

The reconstructed objects are often too large or not specific enough for physicists to do "day-to-day" analysis on them directly. Thus, the whole collaboration (i.e. experiment) will perform collective production of "summary tables" containing values for the most important attributes (or aggregations of them) from all of the events. The tables contain links to objects within AOD, and/or ESD, and/or RAW data. Such summary tables are referred to as *tag data*, and are stored on local data resources. Typical tables will contain 1 KB worth of data for each of the 10^9 events produced each year. These so-called *collaboration tags* will be produced 25 times per year. In addition, there may be around 25 groups of physicists per experiment who

Table 1.1: Data Model for High Energy Physics.

Data type	Size/event	Size/year
Raw Data (RAW)	1 MB	1 PB
Event Summary Data (ESD)	200 KB	200 TB
Analysis Object Data (AOD)	10 KB	10 TB
Tag Data (TAG)	1 KB	1 TB

produce tag data on a subset of all events, so-called *group tags*. Tag data can be also created on demand, in order to test new algorithms for tag data production.

Table 1.1 lists the four general data types, their storage amount per unit and the expected storage amount per year for a typical HEP experiment like CMS [16]. The experiment is supposed to run and to produce data for about 100 days a year over a life span of 15 to 20 years.

1.3 Physics Analysis

A typical physics analysis job starts by selecting a large initial collection of data sets which fulfil certain physics properties. These events are mostly independent from each other which means that the physics result yielded by processing the collection of events (data sets) is independent of the sequence of processing each event. In other words, the processing order of the events could be changed according to some optimisation technique. In an analysis job a physicist applies some "cut predicates" (queries) on the data and thereby reduces the number of events in the event collection. The most common cuts can be regarded as multi-dimensional range queries where the potential search space consists of hundreds of independent dimensions. Queries typically cover 10 to 100 dimensions and are expressed by query predicates, for example:

(Energy $>$ 100.5) AND (pT1 $<$ 83.7) AND (pT2 $<$ 92.6)

where `Energy`, `pT1` and are `pT2` experiment specific physics parameters. An example of a detector image which traces the paths of particles after the collision is given in Figure 1.1.

In a typical analysis effort, the number of resulting events, i.e. the found set, is iteratively reduced by applying cuts with smaller ranges or by adding more attributes to the cuts which results in a higher dimensional query. The results yielded by the cuts are mostly stored in histograms and plotted afterwards for analysing the physics properties. During other typical analysis efforts, arbitrary mathematical expressions of attributes can be used during the selection. A typical example looks like:

sin(pT1) $>$ 0.7 OR sin(pT1) $>$ 0.3

Currently, data access methods in HEP are based on sequentially scanning a search space up to several hundred of dimensions.

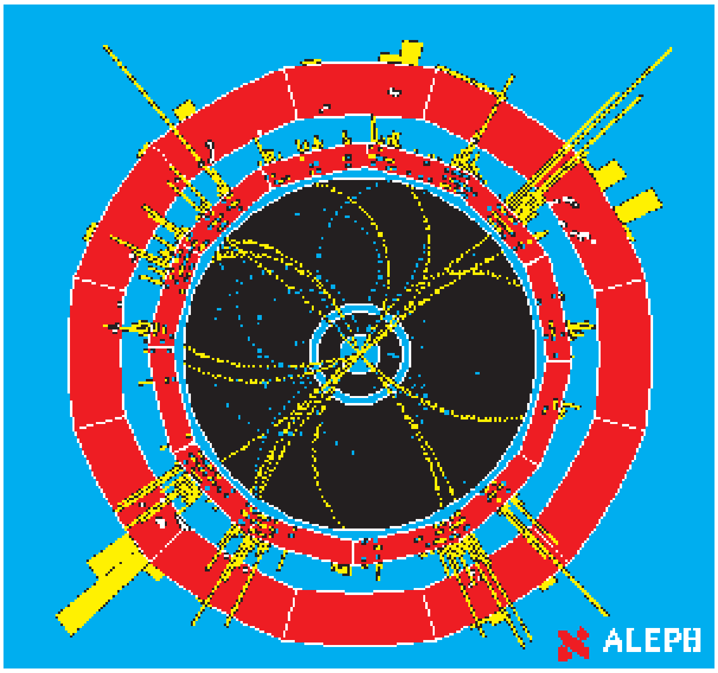

Figure 1.1: Particle traces after event collision.

1.4 Contribution of the Thesis

In Chapter 2 we give an overview of the main trends in database access methods of the last three decades of research. We define the basic terminology we use throughout the thesis and categorise the index data structures according to *hash-based* and *interval-based structures*. We also discuss point access and spatial access methods with respect to high-dimensional search spaces.

Trends in the state of the art of bitmaps indices which are more recent multi-dimensional index data structures mainly applied in data warehouse applications are presented in Chapter 3. We discuss different bitmap design strategies for optimising various query types. We also review various bitmap compression techniques for reducing the size of the index and also improving the query performance characterstics.

Chapter 4 is dedicated to an evaluation of some of the current access methods in High Energy Physics (HEP) for end-user analysis. We give a performance analysis of sequentially scanning physics data clustered in two different ways on top of an object oriented database management system. The two different clustering approaches are called *generic tag* and *sliced tag*. The results of this analysis serve as the main comparison for evaluating multi-dimensional access methods.

In Chapter 5 we report on the desgin and implementation of bitmap indices for scientific data which is characterised by non-discrete attribute values, in particular mostly floating points. We start with a brief justification that bitmap indices can improve the query response time of typical physics analysis queries. We will introduce a novel cost model for studying analytically the performance of so-called *equality encoded* bitmap indices and discuss the optimal index size for various kinds of queries.

In Chapter 6 we propose a novel bitmap index algorithm called `GenericRangeEval` for processing multi-dimensional range queries against scientific data. We also extend the cost model introduced in Chapter 5 to predict the performance of this index data structure. Both analytically and experimentally we show that for up to 25 dimensional range queries our proposed bitmap index can significantly speed up the query responds time when compared to the sequential scan.

Various optimisation possibilities for bitmap indices are discussed in Chapter 7. In particular, we elaborate on different binning strategies and different query plans. We also evaluate experimentally the impact of bitmap compression for equality encoded and range encoded bitmap indices based on scientific data.

In Chapter 8 we demonstrate that by using this kind of access method we can significantly improve the performance of typical queries of two different application areas, namely High Energy Physics and Astronomy.

Chapter 9 is dedicated to an outlook of *Grid Query Optimisation* where we discuss various access optimisation opportunities within a wider range of HEP data. In particular, we elaborate on optimisation opportunies for possible physics analysis on data which is replicated all over the world.

Concluding remarks and a summary of the results of this thesis are given in Chapter 10.

Chapter 2

"Conventional" Multi-dimensional Access Methods

2.1 Introduction

In this chapter we first give a definition about the terminology for index data structures and then outline the main trends of data access methods in database management systems. Rather than giving a detailed discussion on every single algorithm, we will focus our attention on the most important data structures and give a short impression why some of these are still regarded as "near optimal" due to 30 years of intensive database research. One example of these data structures is the B-tree which was developed in 1972 by Bayer et al. [3] and is still regarded as one of the most universal algorithms. This will become clear when we take a closer look at Berchtold's Pyramid-tree [5], a structure for indexing high-dimensional data where the basic idea is based on a simple B-tree.

2.2 Problem Definition

Before we start describing different kinds of index data structures, we define the terminology we will use throughout the thesis.

Basically, we can distinguish between two classes of queries, namely:

- *Exact match queries* of the form $A = v$ where A refers to the attribute and v to a specific attribute value, e.g. income = "30,000"
- *Range queries* of the form $v_1 \leq A \leq v_2$, e.g. $30{,}000 \leq$ income $\leq 40{,}000$.

Throughout the thesis we will analyse access methods for improving the response time of so-called *ad-hoc queries*, i.e. interactive queries, over a multi-dimensional search space which is also called *universe*. In physics terminology a query is mostly referred to as a *cut*.

For most of our evaluations the *selectivity* of a query has an impact on the performance of the data structure. We define the selectivity as the number of objects fulfilling the query constraint (result set) divided by the total number of objects.

For multi-dimensional queries we also talk about *attribute query selectivity*. In this case we refer to the selectivity of a particular attribute. Consider the following two dimensional query with a search space in the range [0;100] and uniformly distributed data values: $a_0 \leq 30$ AND $a_1 \leq 20$. In this case, the attribute query selectivity of a_0 and a_1 is 30% and 20% respectively. The *total query selectivity* is 6% (0.3 * 0.2 = 0.06).

Especially during the discussing of so-called "conventional index data structures" we will often use the term *bucket* which refers to data points that are organised on one disk page.

2.3 Basic Data Structures

According to [43] we can use two different classifications for index data structures, namely:

- Hash-based data structures and
- Interval-based data structures.

Typical hash-based data structures are:

- Linear Hashing [46]
- Extendible Hashing [21]

Linear hashing divides the universe into binary intervals where an interval corresponds to a bucket. If the capacity of a bucket is reached, the bucket is split and a new entry is created in an *overflow page* [46]. *Extendible Hashing* maintains no overflow pages but a central directory. Each bucket (cell) has an entry in a directory. Once the capacity of the directory is reached, all cells are split which means that the directory doubles in size.

Some of the most basic so-called interval-based structures are:

- B-tree [3]
- k-d-tree [4]
- Quadtree [57]

All of these data structures organise the data in a hierarchical way. A *B-tree* is a balanced tree with a height of $log(n)$ where n is the number of nodes in the tree. The search operation on a B-tree is analogous to a search on a binary tree with an order of $O(log(n))$.

The *k-d-tree* is a binary search tree that represents a recursive subdivision of the universe into subspaces by means of $(d-1)$ dimensional hyperplanes. One of the disadvantages of the k-d-tree is that the structure is sensitive to the order in which that data values are inserted and the data points are scattered all over the tree. Thus, variants of the k-d-tree use a split strategy that each hyperplane contains the same number of elements. However, for certain distributions no hyperplane can be found that splits the data points evenly [25].

The *Quadtree* is closely related to the k-d-tree and also decomposes the universe by means of iso-oriented hyperplanes. However, the Quadtrees are no binary trees any more. The interval-shaped partitions do not have to be of equal size and thus this tree is not necessarily well balanced.

Many of the more recent index data structures are based on these approaches. Thus, a good understanding of these basic techniques is vital for elaborating on more complicated multi-dimensional index data structures.

2.4 Point Access Methods (PAMs)

In the previous section we were discussing data structures which are mainly designed for main memory applications (apart from the B-tree) and, thus, do not bear in mind secondary or even tertiary storage devices. However, since database applications become more complex and the data volume much bigger, for instance at CERN we are dealing with data volumes up to several Petabytes, data structures must bear in mind both secondary storage and the underlying operating system.

Point Access Methods can be categorised based on following strategies:

- Multi-dimensional hashing
- Hierarchical access methods

As we mentioned already before, we will only discuss a few point access methods and have a closer look at them. Lets us start with typical data structures based on multi-dimensional hashing.

2.4.1 Multi-Dimensional Hashing

The Grid File

The *Grid File* [45] is a typical data structure that is based on hashing. In short, a d-dimensional orthogonal non-regular grid makes up the universe of the data. The cells, which are yielded by putting a grid over all data, may have different shapes and sizes. These cells are associated with data buckets that in turn reside on one disk page. The grid itself is kept in main memory to guarantee that the data is found with two disk accesses at most.

Data is retrieved according to the following two steps. First, the data is located in the cells by means of scales. Second, if the data does not reside in main memory, it must be fetched with a second disk access. The main disadvantage and, thus, a driving force for further research is the super linear directory growth for non-uniformly distributed data.

Some variations of the Grid File are:

- EXCELL (Extendible Cell) [70]
- Two-Level Grid File [30]
- Twin Grid File [36]

EXCELL decomposes the universe in a regular way such that all grid cells are of equal size. The *Two-Level Grid File* uses a second grid file to manage the grid directory, the same is true for the *Twin Grid File*. However, the first one uses a hierarchical approach whereas the latter one uses a somewhat more balanced approach. A clustering approach based on grid files is presented in [59].

2.4.2 Hierarchical Access Methods

We will now discuss a few of the most important hierarchical point access methods such as:

- k-d-B tree [56]
- LSD-tree [28]
- Buddy-tree [62]
- hB-tree [47]
- BV-tree [23]

The *k-d-B tree* partitions the universe like the adaptive k-d-tree with mutually disjoint regions. In addition it has the advantage of a B-tree, namely it is well balanced and thus guarantees the same access time for all data points. However, this structure does not guarantee minimal space utilisation due to the *forced split policy* [25].

The *LSD-tree* (Local Split Decision) partitions the universe like the adaptive k-d-tree with the advantage that it adapts well to non-uniformly distributed data values.

The *Buddy Tree* can be regarded as an *hybrid approach* since it combines a dynamic hashing approach with a tree structured directory. In order to avoid the disadvantage of the Grid-file, the universe is partitioned into two parts of equal sizes with iso-oriented hyperplanes.

The *hB-tree* (holey brick tree) uses the k-d-tree to organise the space. However, splitting is based on multiple attributes and so called "cascading" splits are avoided.

Finally, the *BV-tree* is an attempt to solve the d-dimensional B-tree problem which can be regarded as a generalisation of the B-tree for higher dimensions. For further details, we refer to the respective literature.

2.5 Spatial Access Methods (SAMs)

In this section we discuss data structures which handle objects with spatial extension with complex structures that might be dynamic and large. According to [25] SAMs are modified PAMs which can be defined according to four categories:

- Overlapping Regions (object bounding)
- Clipping (object duplication, no overlap)
- Transformation (object mapping to higher dimensional space)
- Multiple layers (special case of overlapping regions)

2.5.1 Overlapping Regions

R-tree

Let us start our discussion with the type *Overlapping Regions* where we will in particular focus our attention on one the most studied and modified trees, namely the R-tree which was first published by Guttman in 1984 [27].

The motivation for the development of the R-tree was the lack of flexibility of conventional data structures (hash tables, B-trees,...) in terms of multi-dimensional applications and spatial objects. Other structures like the Quadtree and the k-d-tree do not take paging of secondary memory into account whereas the k-d-B-tree bears this in mind but is most efficient for point data [27].

Thus, the need for the development of an index structure which represents spatial data objects by intervals in several dimensions was prevailing and regarded as an alternative to Grid Files that handle non-point data by mapping each object to a point in a higher-dimensional space [45].

The R-tree is a completely dynamic index structure for n-dimensional spatial objects analogous to a B+-tree. What is more, insertions and deletions can be intermixed with queries. However, the major difference is the representation of the nodes and the organisation of splits. Let us start our discussion with the representation of leaf nodes first.

Leaf nodes contain index record entries of the form $(I, tuple-identifier)$ where *tuple identifier* refers to a tuple in the database and I is an n-dimensional rectangle containing the spatial objects it represents. In contrast, non-leaf nodes contain entries of the form $(I, child-pointer)$ where *child-pointer* is the address of another node in the tree and I covers all rectangles in the lower node's entries [27]. Each node corresponds to a disk page if the structure is disk resident (persistent).

The R-tree uses rectangles for organising the universe. Thus, the data space can consist of many different multi-dimensional geometrical shapes which can be more efficiently organised and approximated by a simple shape such as a bounding box which is true for R-trees. We will later see that other index structures use different shapes for organising the data space, for example, by means of spheres.

In short, the most important property of this simple approximation is that a complex object is represented by a limited number of bytes [10]. It is clear that due to this approximation some data gets lost. However, the most important geometric properties like

- the location of the object and
- the extension of the object in each axis

are preserved.

The main features of an R-tree are as follows [27]:

- The root has at least two children unless it is a leaf.
- Every non-leaf node contains between m and M index records (entries, children) unless it is the root whereas M is the maximum number of entries that will fit into a node and

m is defined as $m \leq \frac{M}{2}$ and specifies the minimum number of node entries. What is more, this index record (I, tuple-identifier) is the smallest record that spatially contains the n-dimensional data object.

- Every leaf node contains between m and M children unless it is the root.
- All leaves appear on the same level.

The height of an R-tree tree containing N index records is given in the worst case by: HeightR-tree=$ceil(log(mN))$ with a worst case space utilisation of space-utilisationR-tree = $\frac{m}{M}$.

A typical R-tree structure and the geometric form it represents is shown in Figure 2.1.

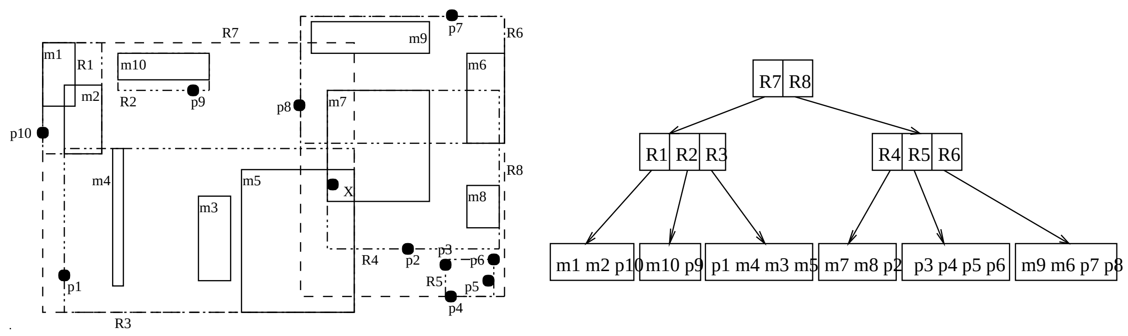

Figure 2.1: R-tree.

The only important feature we want to highlight here is that overlapping regions are allowed in R-trees. However, the main difference to the B+-trees is the organisation of space and the treatment of splits which is based on d-dimensional minimum bounding boxes MBBs. The main idea is that nodes should be split in such a way that the possibility of these splitted nodes to be split again in the near future is minimised [27]. Minimising the total area of the two covering rectangles after the split can yield this effect.

Finally, we will give an example of a query, which illustrates one typical feature of an R-tree, namely the effect of overlapping regions. By looking at Figure 2.1 which illustrates an R-tree, we see that for retrieving the point `X` (which lies within the rectangles `m5` and `m7`) from our universe, two path accesses are required. In particular, following paths are yielded:

- $R8- > R4- > m7$
- $R7- > R3- > m5$

R*-tree

According the Beckmann, Kriegel et al. [10] the heuristic optimisation of the area of enclosing rectangles in each inner node of an R-tree can be improved by a combined approach. In particular, the R*-tree incorporates a combined optimisation of area, margin and overlap of each enclosing rectangle in the directory. The main advantage of the R*-tree over Guttman's R-tree is that both, point and spatial data can be more efficiently retrieved in a high-dimensional universe.

[10] propose to minimise the margin or overlap of the minimum bounding rectangles in the R-tree.

Further considerations and, thus, driving forces for the creation of the R*-tree are as follows:

- Why not optimise storage utilisation?
- Why not minimise the margin or overlap and to optimise the storage utilisation?

The main problem of the R-tree is that it is only based on the optimisation of one parameter, namely the size of the minimum bounding box (MBB). However, since the data rectangles in the universe may have different sizes and shapes in addition to a dynamic change of the size of directory rectangles, the solution, which is yielded by an R-tree, may only be sub-optimal.

In order to overcome a further problem of the R-tree, namely that the directory rectangles which are chosen with respect to the MBB is no longer suitable to a good retrieval performance in the new situation, the R*-tree achieves dynamic reorganisation and, thus, dynamic adaption of the directory rectangle by means of *forced reinsert* [10]. In short, rather than constantly inserting new data points and splitting the nodes afterwards, some nodes are taken out of a particular rectangle and inserted in the tree where they suit best. Thus, the whole tree adapts to the newly inserted data much better.

Experiments show that if a threshold value of p=30% of M is reached (where p refers to the number of points in a particular rectangle), this method of forced reinsert should be applied [10].

To sum up, the R*-tree has following features [10]:

- Forced reinsert changes the entries between neighbouring nodes and thus decreases the overlap, which also improves the storage utilisation.
- Fewer splits occur due to a dynamic restructuring process of the whole tree.
- Due to reinsertion of outer rectangles of a node, the ideal shape of the directory rectangles is quadratic.

A typical R*-tree is depicted in Figure 2.2.

X-tree

One motivation for the development of the X-tree (eXtended Tree) [9] was that the R*-tree, which was one of the most powerful index structures at that time, did not perform well for dimensions greater than five. What is more, due to the fact that overlapping bounding rectangles are allowed, these overlaps may increase even faster with the increasing number of dimensions. According to experiments of Berchtold et al. these overlaps may reach up to 90% if the number of dimensions exceeds five, which yields quite bad performance values.

The X-tree tries to overcome these obvious drawbacks of an R*-tree for high-dimensional space and introduces the concept of *supernodes*. This new method can be regarded as internal

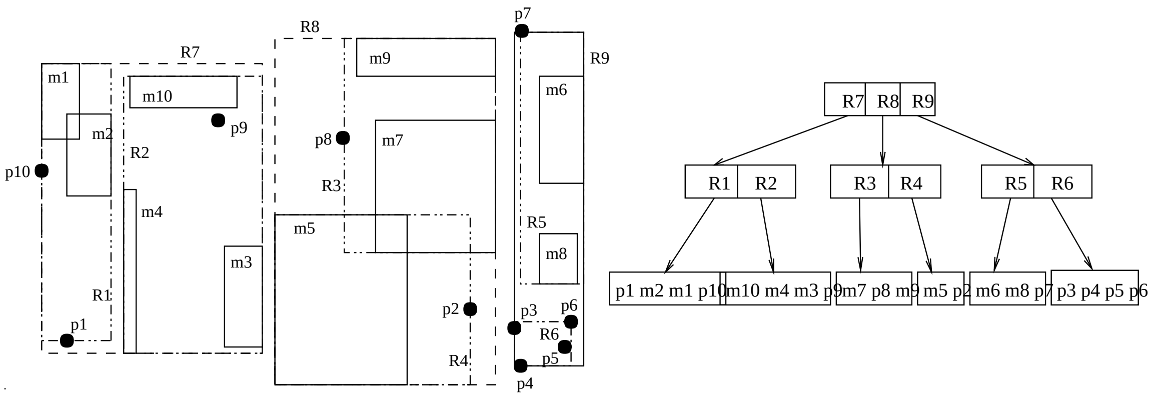

Figure 2.2: R*-tree.

nodes that expand as the number of dimensions increases and consequently reduces the space of overlaps by a certain node split algorithm. In particular, directory nodes are extended over the usual block size, which in the worst case can be regarded as a linear scan.

We can distinguish between two special cases of the X-tree [9]:

- None of the directory nodes (internal nodes) is a supernode.
- The directory consists of only one large supernode (root).

In the first case, the X-tree can be regarded as a fully hierarchical index structure, which is very similar to an R-tree. This case is true for low-dimensional and non-overlapping data. The second case occurs for high-dimensional data and, thus, yields a performance which is equivalent to a linear directory scan which in turn is much more favourable than scanning data in a random fashion on disk and, thus, causing a high number of disk arm movements.

A typical X-tree is depicted in Figure 2.3. The shape for different number of dimensions is given in Figure 2.4.

2.5.2 Clipping

A further technique of SAM is called *Clipping*. The main difference to *Overlapping Regions* is that no overlaps in the bounding rectangles are allowed. In other words, all bucket regions are mutually disjoint.

R+-Tee

The R+-tree [64] is a variant of Guttman's R-tree and does not allow overlapping regions. This can be achieved by allowing partitions to split rectangles so that this technique results in zero overlap among intermediate node entries. However, avoiding overlap is only achieved at the expense of space, which increases the height of the tree. Due to the fewer number of access paths in comparison to an R-tree, the R+-tree yields better search performance for point queries [64]. Exact match queries in R+-trees correspond to single-path tree traversals

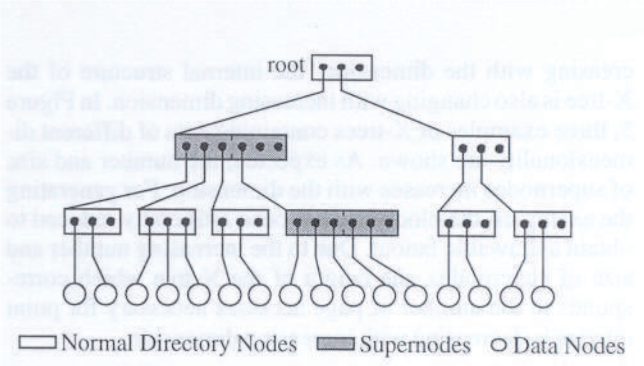

Figure 2.3: X-tree.

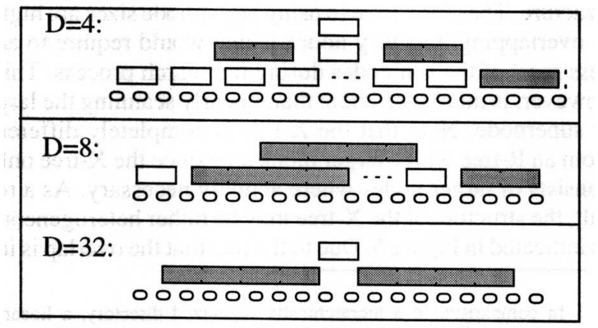

Figure 2.4: Various shapes of the X-tree in different dimensions.

from the root to one of the leaves. Range queries, on the other hand, lead to the traversal of multiple paths in both the R+-tree and the R-tree. A typical R+-tree is depicted in Figure 2.5.

2.5.3 Transformation

Let us explain this category of spatial access methods by means of *Space Filling Curves*, [22] which are used for representing extended objects by a list of grid cells or a list of one-dimensional intervals. In short, space filling curves try to store points, which are close in space, i.e. logical order, also close on disk, i.e. physical order. To put it in other words, these methods try to produce a physical order to points, which are very close in terms of logical order.The main idea is to visit all points in a grid without crossing itself.

Hilbert R-tree

The basic idea of the Hilbert R-tree [41] is to make use of the *deferred splitting approach* used in R-trees. In particular, an ordering of tree nodes is used which groups similar rectangles

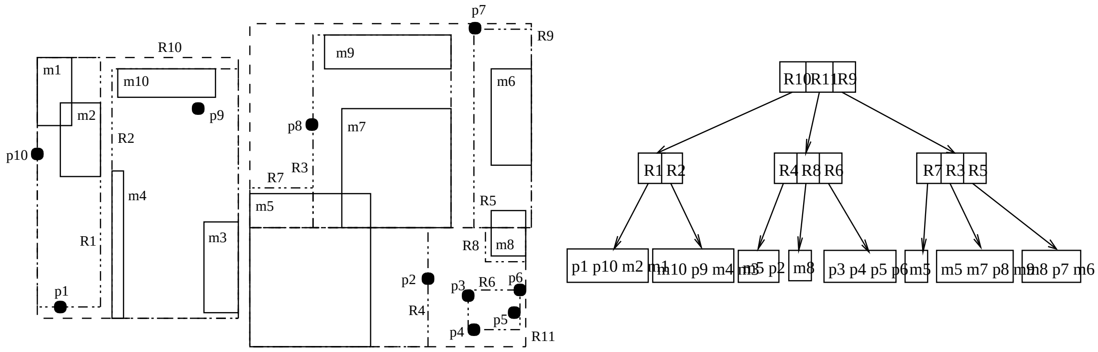

Figure 2.5: R+-tree.

together and thus minimises the area and the perimeter of the resulting minimum bounding rectangles (MBRs).

The authors of [41] claim that the Hilbert R-tree achieves higher space utilisation than the R*-tree proposed in [9] because latter does not allow to control space utilisation. In contrast, the Hilbert R-tree allows to adjust the splitting policy, e.g. 2-to-3 or 3-to-4 etc., which can result in 100% space utilisation, however, the average case is about 70%.

The performance of R-trees depends on the algorithms, which cluster the data rectangles to nodes. The Hilbert R-tree uses the Hilbert Curve for clustering data.

Let us first provide a brief introduction to the Hilbert Curve before we start our discussion on the Hilbert R-tree.

Figure 2.6 shows a typical Hilbert Curve on a 2x2 grid, denoted by H1. It also shows different orders of the Hilbert Curves, which can be generalised for higher dimensionalities. In this case, the order tends to infinity where the result is a fractal. To sum it up, this kind of space-filling curve imposes a linear ordering on the grid points according to a fractal. For more details, we refer to [22].

We can now specify the main characteristics of a Hilbert R-tree [41]:

- behaves like an R-tree
- supports deferred splitting on insertion by means of the Hilbert value of the inserted data rectangle as the primary key

What is more, for every node n of the tree, the

- MBR and
- Largest Hilbert Value (LHV) of the data rectangles that belong to the subtree with root n are stored.

We see that similar to the R*-tree, overlapping regions are allowed. However, even non-leaf nodes contain entries about the LHVs. In our example we see that every node keeps track of the LHV and the MBR defined by XL, YL, XH, and YH, i.e. the coordinates of the MBR.

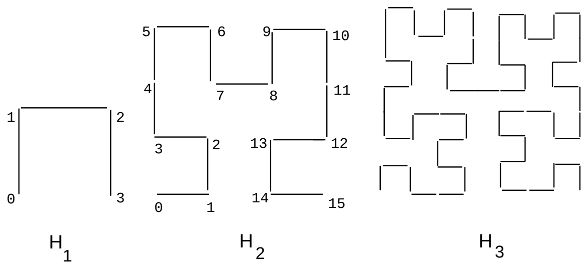

Figure 2.6: Hilbert Curves of order 1, 2 and 3.

One of the advantages of this splitting strategy over the R*-tree is that a shallower tree and a higher fanout are yielded due to a better packing mechanism of the tree [41].

2.5.4 Multiple Layers

The multiple layer technique can be regarded as a variant of the overlapping region approach, because data regions of different layers may overlap. The characteristics of this method can be summarised as follows [25]:

- Layers are organised in a hierarchical way.
- Each layer partitions the universe in a different way.
- Data regions with a layer are disjoint.
- Data regions do not adapt to the spatial extensions of the corresponding data objects.

To give only some examples of the multiple layer technique, we want to mention the *Multi-Layer Grid File* [65] and the *R-File* [37] and refer the reader again to the original literature for a detailed discussion.

2.6 Pyramid-tree

The motivation for the development of the Pyramid-tree was the sub-optimal split strategy of index structures studied so far. This is especially true for high-dimensional data. What is more, random page accesses for performing queries against a multi-dimensional universe mostly causes many disk arm movements and yields the typical I/O bottleneck.

The Pyramid-tree tries to overcome these drawbacks by a new splitting strategy, which is optimised for high-dimensional data. The basic idea is to transform the d-dimensional data points into 1-dimensional values [5] and then store and access the data in a way, which we know from conventional B+-trees [17] and thus "inherit" the positive features of this index data structure, namely:

- height-balancing due to good splitting technique

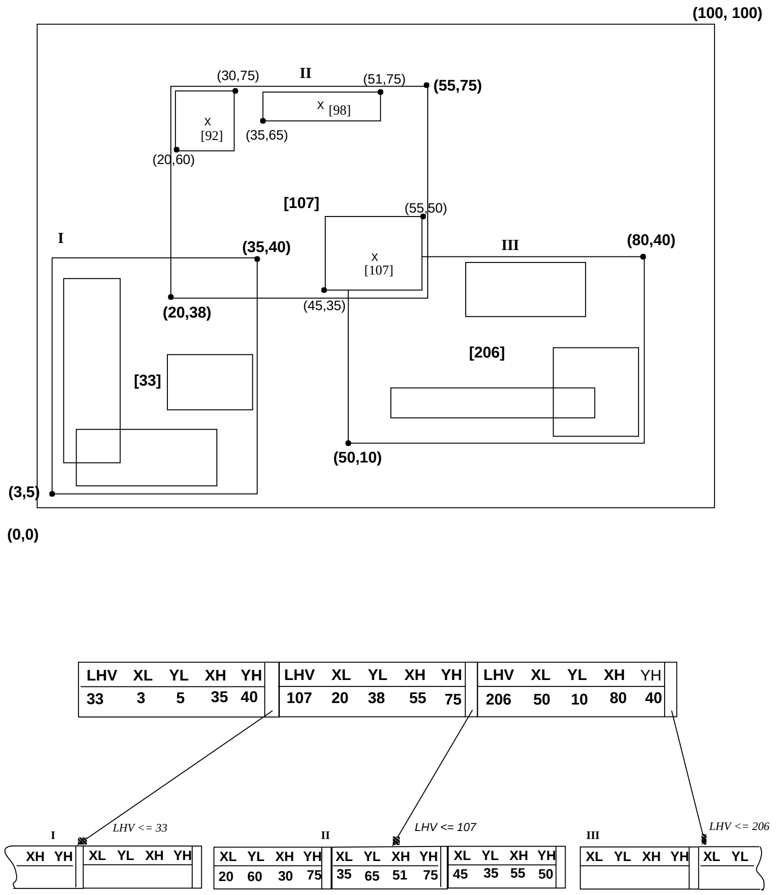

Figure 2.7: Hilbert R-tree.

- good transaction times and bulk loading effects

In what follows, we will give a detailed discussion on the technique of this index structure for high-dimensional data. We will also give an example of how to build such a structure.

2.6.1 Treatment of Data Space

First, the data space is divided in $2d$ pyramids. We will assume a d=2-dimensional data space which results in 4 pyramids that make up the whole universe.

Second, each pyramid is divided into partitions or layers, which are parallel to the base line of the pyramid. These layers in turn correspond to one data page of the resulting B+-tree. The partitioning method is depicted in Figure 2.8.

On partitioning the data space in different layers of pyramids, we can now discuss how the key values for the B+-tree are maintained. Basically, each key value can be regarded as a point in high-dimensional space specified by:

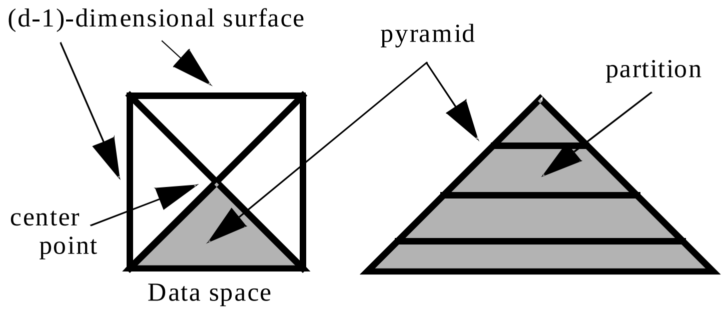

Figure 2.8: Space partitioning method of the Pyramid-tree.

- pyramid number $p_0, ..., p_{2d}$ where d refers to the number of dimensions
- height h_v of the value within a pyramid

For example, the pyramid value p_1 and the height h_v define value v in Figure 2.9.

$$h_v = \left|0.5 - v_{i \text{ MOD } d}\right|$$

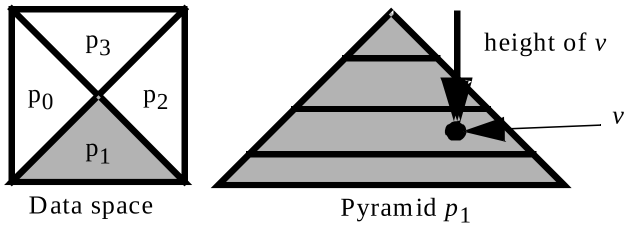

Figure 2.9: Key values for the Pyramid-tree.

2.6.2 Index Creation and Query Processing

Let us now take a look at how data are retrieved In particular we only like to retrieve data, which lies in a certain range. Thus, this range has to be interpreted by the lower and upper height value, h_{low} and h_{high}, respectively. In order to keep the retrieval process as general as possible, the universe is normalised to $[0,1]^d$. In other words, the scope of each dimension of our search space lies between 0 and 1.

In our example we see that every query is described by a query rectangle and then the intervals $[h_{low}, h_{high}]$ must be evaluated for each pyramid which is intersected by the query rectangle.

In general, data from a Pyramid-tree are retrieved in two steps:

- Step 1: determine the affected pyramids and the corresponding heights of the values

- Step 2: select the qualified attributes from the result of step 1

Let us now elaborate on these two basic steps and, thus, explain the characteristics of this index data structure.

Step 1 is exactly what we were discussing so far. Step 2 then actually retrieves the data by means of the Pyramid-tree. Consequently, all the qualified values, which are yielded by step 1, i.e. the values, which are within the scope of the intersecting rectangle, must be checked against the initial query because step 2 yields more data than actually needed by the query.

For example, take a look at the rectangle of pyramid p_1 in our next figure. Here we see that the right most corner lies in the range of the pyramid p_2 but is not "needed" by our query. Thus, this "surplus" data must be sieved out which is done in step 2.

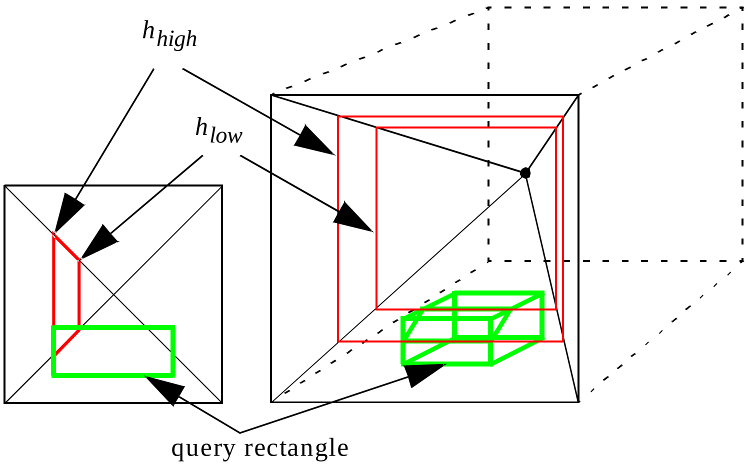

Figure 2.10: Query processing with the Pyramid-tree.

Chapter 3

Bitmap Indices

3.1 Introduction

Bitmap Indics were first proposed by O'Neil in the Model 204 DBMS [50] although they were already used in the 60s. This kind of index data structure is mostly used for typical OLAP and data warehouse applications [13], which are mainly characterised by complex query types and read-mostly environments. Bitmap indices reflect these requirements and optimise the query performance. However, bitmap indices are not optimised for typical transaction operations such as insert, delete or update.

In this chapter we give a detailed survey of various bitmap index techniques and discuss the advantages and disadvantages with respect to the more conventional index data structures presented in the previous chapter.

3.2 Simple Bitmap Indices

We motivate our discussion on bitmap indices with a simple example about stock trading [75] where the advantage of bitmap indices can be demonstrated very easily. Table 3.2 shows various stocks traded at different stock exchanges. We first merely concentrate our attention on the last column.

We can see that stocks are traded at two different stock exchanges, namely at NASDAQ and at NYSE. More formally, the attribute *Exchange* has two distinct attribute values. Furthermore, we see that our stock example comprises 12 different stocks which are uniquely identified by their record ID given in the first column.

Stocks and their corresponding trading places can be represented by the following simple bitmaps:

```
NASDAQ:  (1 0 0 0 0 1 0 0 0 1 1 1)
NYSE:    (0 1 1 1 1 0 1 1 1 0 0 0)
```

Since we have two distinct attribute values for the stock exchange, we need two bitmaps, which in turn consist of 12 bit values since our example comprises 12 different stocks.

Record ID	Ticker Symbol	Trading Volume	Closing Price	Exchange
1	AAPL	4,575,000	36.625	NASDAQ
2	ABF	64,200	24.500	NYSE
3	AET	369,000	72.625	NYSE
4	CPQ	8,968,800	51.375	NYSE
5	DEC	4,461,100	49.750	NYSE
6	DELL	2,714,400	89.750	NASDAQ
7	HWP	3,009,300	90.250	NYSE
8	IBM	7,657,700	92.500	NYSE
9	IFMX	3,493,600	33.000	NYSE
10	INTC	17,694,400	65.500	NASDAQ
11	LGNT	2,600	47.250	NASDAQ
12	MSFT	18,288,600	91.125	NASDAQ

Table 3.1: Bitmap Indices for Stock Trading.

For example, the first bit of the bitmap of NYSE is set to 0 because the first stock is not traded at NYSE. However, the next four stocks are traded at NYSE and, thus, the bits 2, 3, 4 and 5 are set to 1. In general, a bit is set to 1 if the stock is traded at the particular stock exchange, and it is set to 0 otherwise. We, thus, have a straightforward way of describing the stock exchange by means of bitmaps.

How do we retrieve data from such a bitmap index? We, therefore, make a simple modification of our example and assume that some stocks are traded at both stock exchanges which could result in the following two bitmaps:

```
NASDAQ:  (1 0 1 1 0 1 0 0 0 1 1 1)
NYSE:    (0 1 1 1 1 0 1 1 1 0 1 0)
```

We see that the 3rd, 4th and 11th bit are set in both bitmaps. This means that in our example these stocks are traded at both stock exchanges. In order to retrieve this information from our database, we simply AND both bitmaps together, i.e. we perform a bitwise Boolean AND-operation between the two bitmaps.

```
NASDAQ:  (1 0 1 1 0 1 0 0 0 1 1 1)
NYSE:    (0 1 1 1 1 0 1 1 1 0 1 0)  AND
         (0 0 1 1 0 0 0 0 0 0 1 0)
```

Since the 3rd, the 4th and the 11th bits of the resulting bitmap are set to 1, we know that these stocks are traded at both stock exchanges.

3.2.1 Space Complexity

On giving these simple examples, we will now discuss the space and time complexities for building simple bitmap indices and compare them to a B+-tree. However, we discuss the time complexity for querying bitmap indices in Chapter 6.

Let T be a (database) table and let $|T|$ be the cardinality of T, i.e. the number of distinct tuples in T. Thus, the space complexity in terms of bytes for building a simple bitmap index on an attribute A of the table T is given as:

$$size_{bitmap} = \frac{|T||A|}{8} \tag{3.1}$$

where $|A|$ corresponds to the cardinality of attribute A, i.e. number of distinct values of attribute A.

The space complexity in bytes for a B+-tree is given by:

$$size_{b-tree} = \frac{1.44p|T|}{M} \tag{3.2}$$

where p is the page size and M the degree of the B+-tree, i.e. the maximum number of elements in one data bucket. When we assume a page size p of 4 KB and a bucket size M of 512, then a bitmap on A is more space efficient than B+-tree if if $|A| < 93$. In general, for low cardinality attributes the bitmap index is more space efficient than the B+-tree.

3.2.2 Time Complexity

The time complexity for building a bitmap index in big O notation is given by (worst case):

$$O(|T||A|) \tag{3.3}$$

In contrast, the worst case for building a B+-tree is given by:

$$O(|T|log\frac{M}{2}|A|) + O(|T|log_2\frac{p}{4}) \tag{3.4}$$

where p is the page size and 4 the size of the tuple ID. Term 1 refers to the cost of traversing the tree from root to leaf nodes and term 2 refers to the cost of inserting tuple-IDs into the corresponding leaf nodes. Let us make following simple considerations. If $|T|$ is very large and $|A|$ is very small, then the time complexity of building B+-trees is larger than for building a bitmap index.

3.2.3 Pros and Cons of Simple Bitmap Indices

The main advantage of bitmap indices is that logical operations are very well supported by hardware and, thus, the operations are executed quite fast. What is more, the cost for constructing bitmap indices as well as the processing costs are very low.

However, simple bitmap indices are only efficient for attributes with a low number of distinct values. In other words, if the cardinality of the indexed attribute is low, a low number of bitmaps is required and, thus, the space complexity for such an index structure is low. For high cardinality attributes the space complexity of the simple bitmap index is considerably higher than for conventional index data structures.

3.3 Equality, Range, Interval Encoding

A detailed discussion on designing bitmap indices based on different encoding schemes is presented in [14] and [15]. In particular, space and time complexities for so-called *equality encoded* (simple bitmap index), *range encoded* and *interval encoded* bitmap indices are evaluated. Equality encoding (Table. 3.3 (b)) can be regarded as the most fundamental method that consists of $|A|$ bitmaps (bitmap vectors) where $|A|$ is the cardinality of the attribute to be indexed on. This type of index is optimal for exact match queries of the form $Q_e : v = a_i$.

	π_A(R)	E^9	E^8	E^7	E^6	E^5	E^4	E^3	E^2	E^1	E^0
1	4	0	0	0	0	0	1	0	0	0	0
2	3	0	0	0	0	0	0	1	0	0	0
3	4	0	0	0	0	0	1	0	0	0	0
4	5	0	0	0	0	1	0	0	0	0	0
5	7	0	0	1	0	0	0	0	0	0	0
6	8	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0	0	0	1
9	4	0	0	0	0	0	1	0	0	0	0
10	8	0	1	0	0	0	0	0	0	0	0
11	6	0	0	0	1	0	0	0	0	0	0
12	7	0	0	1	0	0	0	0	0	0	0

Table 3.2: a) Projection Index π_A and b) Equality Encoding E^i.

One sided-range queries like $Q_{1r} : v_1$op a_i where $op \in \{<, \leq, >, \geq\}$ show the best performance characteristics with range encoded bitmap indices (Table 3.3 (b)), which only consist of $|A| - 1$ bitmap vectors. Finally, interval encoding (Table 3.3 (c)) consists of $\frac{|A|}{2}$ bitmap vectors only and is optimal for two-sided range queries $Q_{2r} : v_1$op a_i op v_2 where $op \in \{<, \leq, >, \geq\}$.

Table 3.3 and 3.3 depict these different encoding techniques for the same set of attribute values. According to the terminology of [51, 52] the values in Table 3.3 (a) are referred to as *projection index* whereas the other methods are called *bit sliced* indices.

A first algorithm called `RangeEval` for evluating queries based on range encoded bitmap indices was presented by [51]. Later [14] proposed a new algorithm called `RangeEval-Opt` hat reduces the number of bitmap operations by about 50% and requires one less bitmap scan for range queries. Both algorithms are depicted in Figure 3.1. We will go into more detail about these algorithms in Chapter 6.

3.4 Range-Based Indices

One of the major problems of simple bitmap indices, namely handling of large cardinality domains, is solved in [75] by range-based indices. A bitmap vector is used to represent an attribute range instead of a distinct value. The entire ranges are partitioned into equally spaced

	π_A(R)	R^8	R^7	R^6	R^5	R^4	R^3	R^2	R^1	R^0	I^4	I^3	I^2	I^1	I^0
1	4	1	1	1	1	1	0	0	0	0	1	1	1	1	1
2	3	1	1	1	1	1	1	0	0	0	0	1	1	1	1
3	4	1	1	1	1	1	0	0	0	0	1	1	1	1	1
4	5	1	1	1	1	0	0	0	0	0	1	1	1	1	0
5	7	1	1	0	0	0	0	0	0	0	1	1	0	0	0
6	8	1	0	0	0	0	0	0	0	0	1	0	0	0	0
7	0	1	1	1	1	1	1	1	1	1	0	0	0	0	1
8	0	1	1	1	1	1	1	1	1	1	0	0	0	0	1
9	4	1	1	1	1	1	0	0	0	0	1	1	1	1	1
10	8	1	0	0	0	0	0	0	0	0	1	0	0	0	0
11	6	1	1	1	0	0	0	0	0	0	1	1	1	0	0
12	7	1	1	0	0	0	0	0	0	0	1	1	0	0	0

Table 3.3: a) Projection Index π_A, b) Range Encoding R^i and c) Interval Encoding I^i.

buckets. However, range-based indices require additional query processing time to examine the details of all the records in the matched buckets. A detailed analysis and a possible solution to the problem of the additional overhead for retrieving data from disk ("sieving out" the matching attribute values), was still left an open issue.

We will now take a look at how to partition the attribute *Trading Volume* of our "Stock Market Example" and how to represent it by means of a range-based bitmap index [75]. We, therefore, assume a maximum trading volume of 20,000,000 shares per day, which is quite a reasonable assumption for a stock exchange. We now divide the attribute *Trading Volume* into two equally sized ranges

```
[10,000,000; 20,000,000]: (0 0 0 0 0 0 0 0 0 1 0 1)
[0; 10,000,000):          (1 1 1 1 1 1 1 1 1 0 1 0)
```

We can easily see, for instance, that the 10th and 12th stock are traded in a volume greater than 10,000 stocks per day since the 10th and the 12th bit of this bitmap vector are set.

The great advantage of the range-based index over the simple index is that a lower number of bitmap vectors needs to be stored. However, the resulting query process might be longer because of the additional query processing for the ranges (see above).

Lets us now discuss how data is retrieved. We will demonstrate this by means of a simple example. We assume that we are interested in all stocks at NYSE that have a trading volume of more than 4 million shares. Thus, the two bitmap vectors for the attribute *Exchange* and the range [0; 10,000,000) are ANDed together:

```
[0; 10,000,000):    (1 1 1 1 1 1 1 1 1 0 1 0)
NYSE:               (0 1 1 1 1 0 1 1 1 0 1 0)  AND
candidates          (0 1 1 1 1 0 1 1 1 0 1 0)
```

Evaluation Algorithms for Selection Queries Using Range-Encoded Bitmap Indexes.

Input: n is the number of components in the range-encoded index.
$<b_n, b_{n-1}, \ldots, b_1>$ is the base of the index.
op is the predicate operator, $op \in \{<, >, \leq, \geq, =, \neq\}$.
v is the predicate value.
B_{nn} is a bitmap representing the set of records with non-null values for the indexed attribute.

Output: A bitmap representation of the set of records that satisfies the predicate "A op v".

Algorithm `RangeEval`

$B_{GT} = B_{LT} = B_0$;
$B_{EQ} = B_{nn}$;
let $v = v_n v_{n-1} \ldots v_1$;
for $i = n$ **downto** 1 **do**
  if $(v_i > 0)$ **then**
    $B_{LT} = B_{LT} \vee (B_{EQ} \wedge B_i^{v_i - 1})$;
    if $(v_i < b_i - 1)$ **then**
      $B_{GT} = B_{GT} \vee (B_{EQ} \wedge \overline{B_i^{v_i}})$;
      $B_{EQ} = B_{EQ} \wedge (B_i^{v_i} \oplus B_i^{v_i - 1})$;
    else
      $B_{EQ} = B_{EQ} \wedge \overline{B_i^{b_i - 2}}$;
  else
    $B_{GT} = B_{GT} \vee (B_{EQ} \wedge \overline{B_i^0})$;
    $B_{EQ} = B_{EQ} \wedge B_i^0$;
$B_{NE} = \overline{B_{EQ}} \wedge B_{nn}$;
$B_{LE} = B_{LT} \vee B_{EQ}$; $B_{GE} = B_{GT} \vee B_{EQ}$;
return B_{op};

Algorithm `RangeEval-Opt`

$B = B_1$;
if $(op \in \{<, \geq\})$ **then** $v = v - 1$;
let $v = v_n v_{n-1} \ldots v_1$;
if $(op \in \{<, >, \leq, \geq\})$ **then**
  if $(v_1 < b_1 - 1)$ **then** $B = B_1^{v_1}$;
  for $i = 2$ **to** n **do**
    if $(v_i \neq b_i - 1)$ **then** $B = B \wedge B_i^{v_i}$;
    if $(v_i \neq 0)$ **then** $B = B \vee B_i^{v_i - 1}$;
else
  for $i = 1$ **to** n **do**
    if $(v_i = 0)$ **then** $B = B \wedge B_i^0$;
    else if $(v_i = b_i - 1)$ **then** $B = B \wedge \overline{B_i^{b_i - 2}}$;
    else $B = B \wedge (B_i^{v_i} \oplus B_i^{v_i - 1})$;
if $(op \in \{>, \leq, \neq\})$ **then**
  return $\overline{B} \wedge B_{nn}$;
else
  return $B \wedge B_{nn}$;

Figure 3.1: Algorithm `RangeEval` and `RangeEval-Opt`.

The resulting 8 candidates, which are represented by the 1-bit, now need to be checked against the value "larger than 4 million".

To sum up, we see that with the range-based index two search steps are necessary instead of only one which is true for the simple index. However, one of the great difficulties with this index is to find an optimal partitioning of the range in order to keep the processing time in step 2 low.

3.5 Encoded Bitmap Indices

Another encoding technique based on binary encoding is proposed by [71] where an attribute value is represented in binary form with only $\lceil log_2 |A| \rceil$ bitmaps. Obviously, the storage overhead is much less for high cardinality attributes when compared to equality encoding or range encoding but even according to the authors, an optimal solution for evaluating the queries might not always exist.

Let us again give an example taken from a typical data warehouse application. Suppose that we have a fact table SALES with n tuples and a dimension table PRODUCTS with 12,000 different products. Building a simple bitmap index on PRODUCTS requires 12,000 bitmap vectors of n bits in length. However, by using encoded bitmap indexing we only need

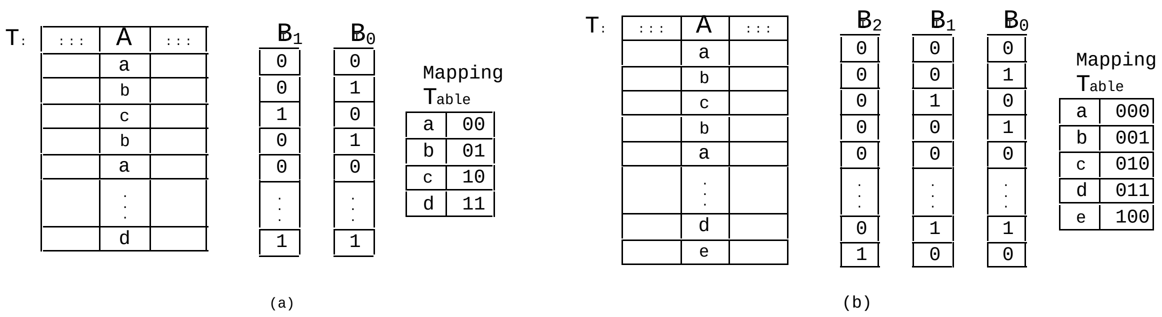

Figure 3.2: Hufmann encoded bitmap index.

$\lceil log_2 12,000 \rceil = 14$ bitmap vectors plus a mapping table which is a very considerable reduction of the space complexity.

In the next example (see Figure 3.2) we will show how Huffman encoding can be used for reducing the space complexity of bitmap indices. Suppose that our attribute domain is given by the table T is a, b, c. The encoding schema of encoded bitmap indices is stored in a separate table called "mapping table" and simply encodes the values from a simple bitmap index by means of Huffman encoding and thus reduces the number of bitmaps vectors. In particular, we use only $\lceil log_2 3 \rceil = 2$ encoded bitmap vectors instead of 3 simple bitmap vectors. This means that 2 bits are used to encode the domain a,b,c. For example, the attribute value of a is represented by the bit string 100 in the table of the simple bitmap index whereas in the table of encoded bitmap index the attribute value a is encoded as 00.

3.6 Miscellaneous Techniques

Static and dynamic query optimisation for *continuous range selections* (i.e. one-sided and two-sided range queries) and *discrete range selections* (i.e. queries of the form $v \in a$ and $v \notin a$ are presented in [74]. Static query optimisations are questions concerning the optimal design of bitmaps and algorithms based on logical reductions. Dynamic query optimisation tries to answer questions on inclusion and exclusion for bit-sliced and encoded bitmap indices.

The most well known work on bitmap indices for HEP is presented in [58]. Their work is based on a hybrid approach of equality encoded [14] and range-based bitmap indices [75] on top of a mass storage system. They also use bitmaps for their query optimiser to provide a quick estimate of the size of the requested data.

3.7 Bitmap Compression

One of the main problems of uncompressed (verbatim) bitmap indices with high cardinality attributes is their high storage costs and as a consequence also high I/O costs. By compressing the bitmaps, following performance advantages can be identified [39]:

- Less disk space is required to store the indices. Thus, the bitmaps can be read from disk into memory faster and also more indices can be kept in the memory cache.
- Due to a smaller index, Boolean operations between compressed bitmaps might be faster than between verbatim ones.

However, some Boolean operations might require the decompression or interpretation of compressed bitmaps which might outweigh the savings in disk space or the bitmap loading time.

Accoring to [39] bitmap compression might introduce following complications into the design of bitmap indices:

- The compressibility of the bitmap depends on the bitmap compression algorithm. The compression ratio of the algorithm again depends on the bit patterns in the bitmap.
- Boolean operations on compressed bitmaps can be performed in different ways. The simpliest form is to decompress the bitmaps and then perform the bitwise Boolean operations one word at a time. Another possibility would be to perform operations directly on compressed bitmaps.

Thus, one of the most important features of bitmap compression algorithms is not only to show good compression ratios but to show good performance of bitwise Boolean operations between bitmaps.

[39] evaluated following three bitmap compression algorithms in a DBMS setting:

- *Lempel-Ziv compression* [76] based on the `zlib` library.
- Variable bit length encoding by means of the *ExpGol* [44] algorithm.
- Variable byte length encoding by means of BBC (Byte-Aligned Bitmap Compression) codes [1].

3.7.1 LZ Encoding

Lempel-Ziv encoding is based on the following simple principle. Long repeated strings in a text are replaced by short compression codes. LZ compression software is available, for example, both as part of the *gzip* file compression tool and the *zlib* data compression library [24].

3.7.2 ExpGol Encoding

A γ code is a basic variable bit length representation of integers [24]. The gamma code of integer n, $\gamma(n)$ is $\lfloor log_2(n) \rfloor$ zero bits followed by the least significant $\lfloor log_2(n) \rfloor + 1$ bits of the binary representation of n. The truncated binary representation of n will always start with a 1. Consider following examples. $\gamma(1) = 1$, $\gamma(2) = 010$, $\gamma(3) = 011$, $\gamma(4) = 00100$, etc.

3.7.3 Byte-Aligned Bitmap Codes

Byte-Aligned Bitmaps Codes (BBC) were proposed by [1]. The advantage of this compression technique is the speed since all operations are performed on full bytes. What is more, Boolean operations on compressed bitmaps can be significantly faster than on verbatim ones.

BBC codes can be *one-sided* and *two-sided*. In principle, every BBC code consists of two parts [39], namely:

- gap
- ending

The gap represents the number of zero bytes that precede the ending. The ending in turn can either be a *bit* (a byte with a single bit set) or a *verbatim* sequence of bitmap bytes. When the bitmap is sparse, bit endings are used, while verbatim endings are used for dense bitmaps. A short gap is expressed in one byte. Long gaps are expressed with multi-byte codes [39]. For further details of the compression algorithm we refer to [1, 39]. However, we will present a simple example to demonstrate parts of the algorithm of two-sided BBC. Consider the following verbatim bitmap comprising 12 bytes.

```
[0] 11111111 [1] 11111111 [2] 11111111 [3] 00001111
[4] 00110000 [5] 00000110 [6] 00000000 [7] 00000000
[8] 00000000 [9] 00011100 [10] 00000000 [11] 00000000
```

We assume that a length 0-3 gap followed by a verbatim ending is expressed according to the following code word [39]. Note that the least significant bit is on the left side.

```
0 [gap length (2)] [fill bit (1)] [verbatim length (4)] (verbatim)
```

Let us now see how our example is interpreted using the description above. The bitmap consists of a length 3 gap with fill bit '1' followed by a length 3 verbatim ending. Next there is a length 3 gap with fill bit '0' followed by a length 1 verbatim ending. Finally the bitmap contains a length 2 gap with fill bit '0'. A possible representation in BBC2 code is as follows:

```
[0] 01110011 [1] 00001111 [2] 00110000 [3] 00000110
[4] 01100001 [5] 00011100
[6] 00100001 | [7] 00000000
```

Bytes 0, 4, 6 and contain the information about the length of the gap, the fill bit and the length of the verbatim bytes. The remaining bytes are represented directly as verbatim. Note that the last length 2 gap is represented as a length 1 gap following by a verbatim. This trick is needed in order to indicate the termination gap.

In [39] these three bitmap compression algorithms presented above are evaluated in more detail. Typical compression ratios of LZ, ExpGol, one-sided BBC (BBC 1S) and two-sided BBC (BBC 2S) on uniformly distributed data values are shown in Figure 3.3. We can see

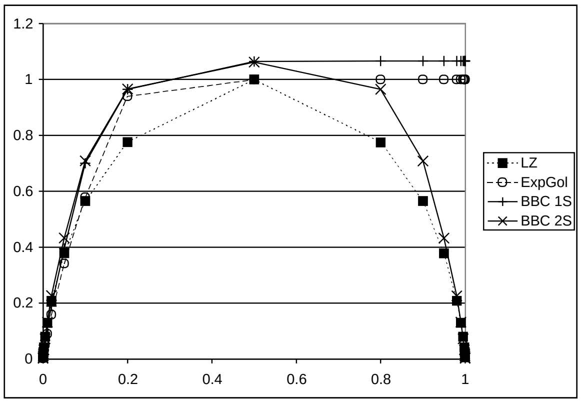

Figure 3.3: Compression ratios of various bitmap compression algorithms. Bit desity (x-axis) vs. compression ratio (y-axis).

that *ExpGol* and BBC Codes show the best compression ratios for very sparse and very dense bitmaps. For the other cases, *LZ Encoding* has the best compression ratios.

The times for compressing and uncompressing bitmaps with these compression algorithms is shown in Figure 3.4. Dense (non-sparse) bitmaps are those bitmaps where more than 10% of the bits are set. We can observe that *ExpGol* shows the best compression and uncompression performance for spare bitmaps whereas for non-sparse bitmaps the "optimal" compression algorithm heavily depends on the bit pattern.

	sparse		non-sparse	
Measurement	uniform	clustered	uniform	clustered
compression rate	ExpGol	ExpGol	ExpGol	LZ, BBC 2S
uncompression time	ExpGol	ExpGol	LZ	BBC

Figure 3.4: Time for compression and uncompression of various bitmap compression algorithms.

In Figure 3.5 we present a summary on the performance of the algorithms for Boolean operations [39]. The performance highly depends on the Boolean operator and the density of the compressed bitmap.

We want to conclude this chapter with a reference to [38] who provide a framework for evaluating different index data structures analytically depending on nine well defined parameters. In particular, two tree-based indices are compared to equality and range encoded bitmap indices. Their results show that especially due to changes in disk technology bitmap indices will most likely outperform tree-based index structures even more significantly in the future.

operation	foundset type	sparse foundset		non-sparse foundset	
		sparse bmp	non-sparse bmp	sparse bmp	non-sparse bmp
AND	uniform	Merge BBC, Direct	Basic	Inplace ExpGol	Basic
AND	clustered	Direct	Direct	Basic	Basic
OR	uniform	Inplace BBC	Basic	Inplace BBC	Basic
OR	clustered	Inplace BBC	Direct	Inplace BBC	Basic

Figure 3.5: Compression algorithm with best performance on Boolean operations.

Chapter 4

Current Access Methods for HEP Queries

4.1 Introduction

Currently most access methods for querying physics data are based on sequentially scanning the base objects and no multi-dimensional indices are used up to now due to the well known "curse of dimensionality". In this chapter we will analyse the performance of sequentially scanning physics data which are clustered in two different ways, namely *object-wise* and *attribute-wise*. We regard these performance benchmarks as the basis for further studying and comparing the performance of multi-dimensional access methods.

As we discussed already in Chapter 1, tags describe physics quantities that are frequently used to select events for analysis. They maintain a logical connection to the data which they summarise and thus allow keeping them consistent with this data. In addition, they support transparent navigation to the original data after a selection has been performed.

The HepODBMS [29] tag implementation (based on Objectivity) which most frequently uses so-called *generic tags* clusters all attributes of a particular event together (event-wise or object-wise clustering). For certain selection types, i.e. selections which only reference a small number of attributes from a tag, a different clustering strategy may result in better performance. We call this *sliced tag* [69] since it clusters all values of a given attribute close together (attribute-wise clustering). We implemented the sliced tag in such a way that it has the same interface as the generic tag and can thus be easily used by physicists who currently use generic tags for their analysis.

A previous performance study in [54] described a benchmark with a PAW-ntuple [53] of 81,060 tags with 302 attributes. The results show that the column-wise ntuple (similar to sliced tag) is more efficient than the row-wise ntuple (similar to generic tag) if less than 10% of all the attributes of a tag are selected. In the worst case, i.e. when all attributes of a tag are selected, the response time for the column-wise ntuple is about 4-5 times higher than for a row-wise ntuple.

In the following sections we give some implementation details and present a performance analysis of sequentially scanning generic vs. sliced tags.

4.2 Implementation Issues

HepODBMS is a C++ class library that provides a simplified and consistent interface to underlying ODMG-compliant [49] object databases. It provides high-level clustering and locking strategies, simplifies database session and transaction control and offers features important to HEP applications, such as highly scalable event collections and event tags [29].

Currently, most of the physics analysis is done in form of writing C++ code rather than using any particular query language. A simple example about using HepODBMS tags for analysis is given below.

First we present a short example about creating a tag collection of 5 attributes:

```
HepExplorable *cd = HepExplorable::findExplorable(tagName);

// create a tag collection
HepExplorableGenericTags myTags;

// start creating a new field
if (!highPt.createDescription(tagName))
description
  fatal("could not create new tag");

// define all fields that belong to genTag
// 5 typical tag attributes could look like follows
TagAttribute<long>  eventNo (myTags,"eventNo");
TagAttribute<double> jet1E (myTags,"jet1E");
TagAttribute<double> jet2E (myTags,"jet2E");
TagAttribute<double> jet1Phi (myTags,"jet1Phi");
TagAttribute<double> jet1Theta (myTags,"jet1Theta");

for (int tag=0; tag < noTags; tag++)
{
  myTag.newTag();
  eventNo = tag;
  jet1E = ...
  if (jet1E > 34.5) ...
}
```

It is important to note that *TagAttribute* is a transient class which keeps a pointer to the underlying persistent-capable tag class. What is more, all objects of the type *TagAttribute* can be treated in the same way like C++ data types and can thus be used within complex mathematical expressions.

After creating a tag collections, a simple analysis program looks like follows:

```
HepExplorable *myTag = HepExplorable::findExplorable(name);
```

```
// define all fields that belong to genTag
TagAttribute<long>  eventNo (myTags,"eventNo");
TagAttribute<double> jet1E (myTags,"jet1E");
TagAttribute<double> jet2E (myTags,"jet2E");
TagAttribute<double> jet1Phi (myTags,"jet1Phi");
TagAttribute<double> jet1Theta (myTags,"jet1Theta");

for(int more = myTag->start(); more ; more = myTag->next())
{
  // apply some cuts on jet1Phi and jet1Theta
  double jetE = 4;
  const double sinThetaCut = 0.9;

  if ( jet1E > jetE   && jet2E < -jetE &&
       fabs(sin(jet1Theta)) < sinThetaCut
     )
  ...
}
```

The example above shows the usage of the generic tag. Since the interfaces for the sliced tag are the same, it can be used in the same way after replacing following statement during the creating of the tag:

```
HepExplorableGenericTags myTags;
```

is replaced by:

```
HepExplorableSlicedTags myTags;
```

4.3 Performance Analysis

Our basic implementation of sliced tags in HepODBMS shows a very similar behaviour to the results presented in [54]. However, by using prefetch optimisation (read-ahead), the relative performance of the sliced tag is improved by a factor of two. Thus, the sliced tag is more efficient than the generic up to an attribute selectivity of 25% (i.e. 25% of all attributes of a tag are accessed) rather than 10% without prefetch optimisation. (We give a brief description of the prefetch optimisation below.) What is more, in the worst case, the sliced tag is only 2.5 times slower than the generic tag. Figure 4.1 a) shows the response time of accessing various numbers of tag attributes based on generic tags, "basic" sliced tags and sliced tags with prefetch optimisation. All our tests were performed at Caltech's "tier2b" machine (Dual 933 MHz Pentium III Linux server, 900 MB RAM, using a 600 GB 3ware RAID 0 array).

We based our prefetch optimisation on findings presented in [31]. In short, rather than fetching single database pages of parallel data streams, i.e. multiple attributes, we prefetch multiple database pages in chunks of 1 MB. The advantage of this approach over fetching

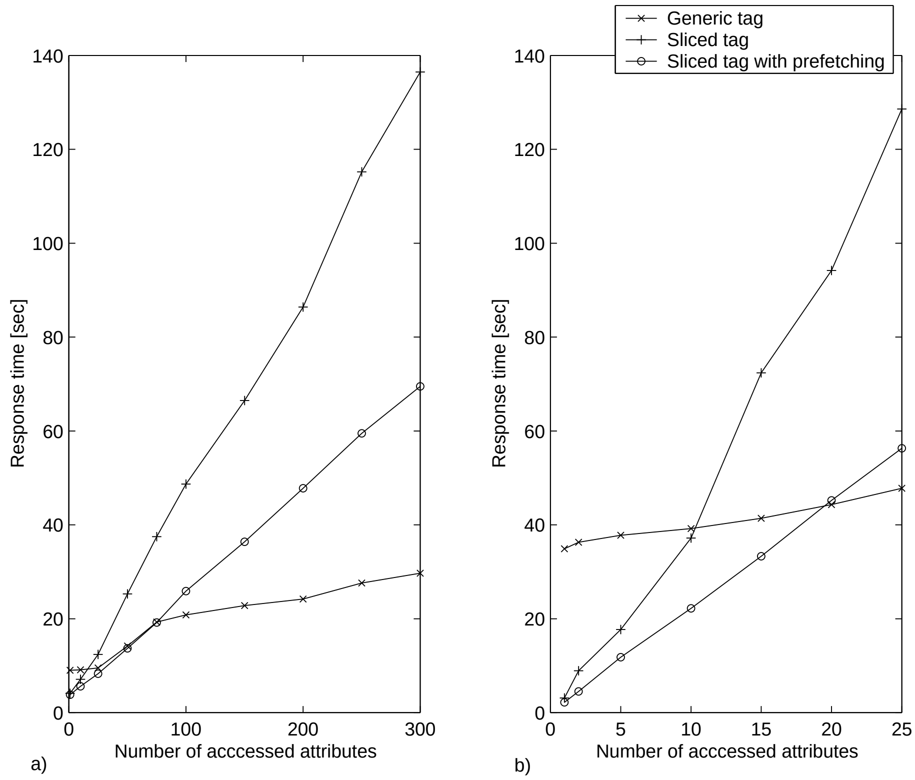

Figure 4.1: Response times for selecting attributes based on various different tag implementations. a) 81,060 tags with 302 attributes b) 1,000,000 tags with 25 attributes.

single database pages is the reduced number of disk head movements for random access and consequently reduced response times. Figure 4.2 shows the typical pattern of the disk arm for random access of two parallel streams without a) and with b) prefetch optimisation.

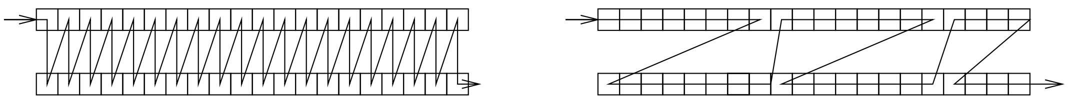

Figure 4.2: Access patterns of disk head movements for reading parallel streams a) without prefetch optimsation b) with prefetch optimisation.

We also carried out some performance benchmarks for 1,000,000 tags with 25 attributes. In this case, the advantage of sliced tags over generic tags is even more significant (see Figure 4.1 b). The sliced tag outperforms the generic tag up to an attribute selectivity of 80%. This is due to the bad performance of Objectivity for handling small objects such as the generic tag with 25 attributes.

4.4 Conclusions

In this chapter we demonstrated the performance of accessing typical physics tags which are clustered in two different ways. We gave a simple example in order to demonstrate a typical physics analysis code and compared the performance of the two clustering strategies.

The sliced tag (attribute-wise clustering) with prefetching is "optimal" up to 25% attribute selectivity - above this threshold the generic tag (object-wise clustering) performs better. In short, for large tags with many attributes the sliced tag is to be preferred if only a subset (up to 25%) of the attributes is selected. However, since tags for physics analysis will consist of some 1000 attributes whereas only up to a few tens will accessed at the same time, the sliced tag is clearly the better choice of implementation.

Chapter 5

Bitmap Indices for Scientific Data

5.1 Introduction

As we pointed out in the introductory chapter, currently most access methods in High Energy Physics analysis are based on sequentially scanning a multi-dimensional search space. Experiences from the past show that often the result set of the analysis is quite small and thus there is a good chance to improve the performance by applying a suitable multi-dimensional index data structure to prune the search space.

We therefore propose using bitmap indices, which are optimised for processing complex multi-dimensional ad-hoc queries in read-mostly environments. [14] [15] [74] studied different kinds of bitmap encoding techniques but only for discrete values. However, additional complexity is imposed on the design and implementation of bitmap indices for non-discrete values since different optimisation techniques to the ones proposed so far have to be applied.

In this chapter we will introduce a novel cost model for analysing the I/O costs for evaluating equality-encoded bitmap indices based on non-discrete attribute values. In particular, we will apply bitmap indices for high performance physics experiments and show that traditional physics analysis can be considerably improved by bitmap indices.

By means of the cost model, we will analytically study the optimal number of bins, and thus the optimal size of multi-dimensional bitmap indices. In particular we will evaluate equality-encoded bitmap indices based on so-called generic tags and compare them to the performance of so-called sliced tags.

The cost model serves as the basis for further evaluation of different bitmap encoding techniques which we will discuss in the next chapters.

5.2 Bitmap Indices for HEP

The typical query profile of physicists who wish to retrieve data for their analyses can be regarded as partial range queries, i.e. queries that do not cover all dimensions of the whole search space and thus only a subset of all dimensions of the data is retrieved. What is more, data is read-mostly and skewed.

In our prototype implementation we created a bitmap index for HEP data comprising 10^6

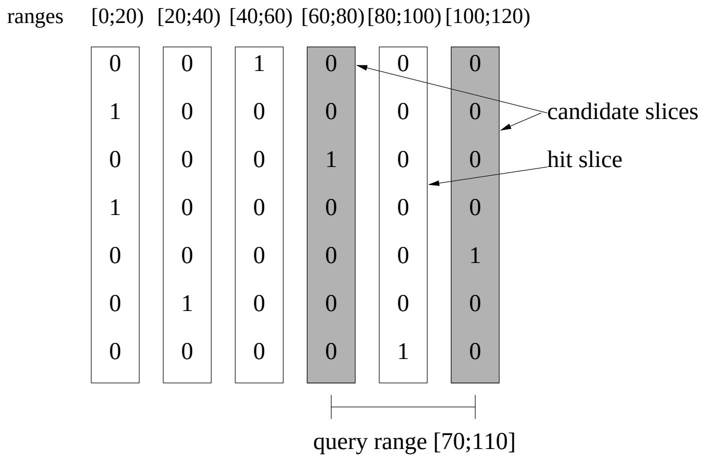

Figure 5.1: One sided-range query on a range encoded bitmap index.

objects, i.e. 1 million events with up to 20 independent attributes. This can be regarded as an index table with a length of 10^6 and a width of 20. We assume that the order of the objects, that are stored in the index, does not change.

Similar to [58] we also use a hybrid approach of equality encoded [14] and range-based bitmap indices that we call *partitioned equality encoding* or short *equality encoding*. The properties or attributes are partitioned into bins, for example the attribute `energy` can be binned into several ranges like [0;20) GeV (Giga electron Volt), [20;40) GeV, etc. Afterwards, a bit slice is assigned to each bin, where *1* means that the value for the particular event falls into this bin and *0* otherwise.

The steps for performing a two-sided range query of the form $Q_{2r} : v_1 \text{op } a_i \text{ op } v_2$ where $op \in \{<, \leq, >, \geq\}$ are as follows. First, the query range has to be interpreted in terms of bins. Thus, we can easily compute how many bins need to be scanned for answering our query. Since each bin represents an attribute range rather than a distinct value, the edge bins might only be partially covered by the query condition. In order to sieve out the correct events from the *candidate slices*, we need to fetch the event data from disk and check the attribute value against the query condition. We refer to this as the *candidate check overhead* that makes the index highly I/O bound for a large number of candidates in the two candidate slices. Those slices that are covered 100% by the query range, are called *hit slices*. In this case all events that are represented by this slice are hits and do not need any additional checking. A typical example of a two-sided range query $70 \leq x \leq 110$ with 2 candidate slices and 1 hit slice is depicted in Figure 5.1.

5.3 Implementation on Objectivity/DB

Basis for our implementation is Objectivity/DB, which is a distributed object database management system for high performance applications. Objectivity/DB provides a robust, scalable multi-threaded database engine. Both the event data and the bitmap index are implemented

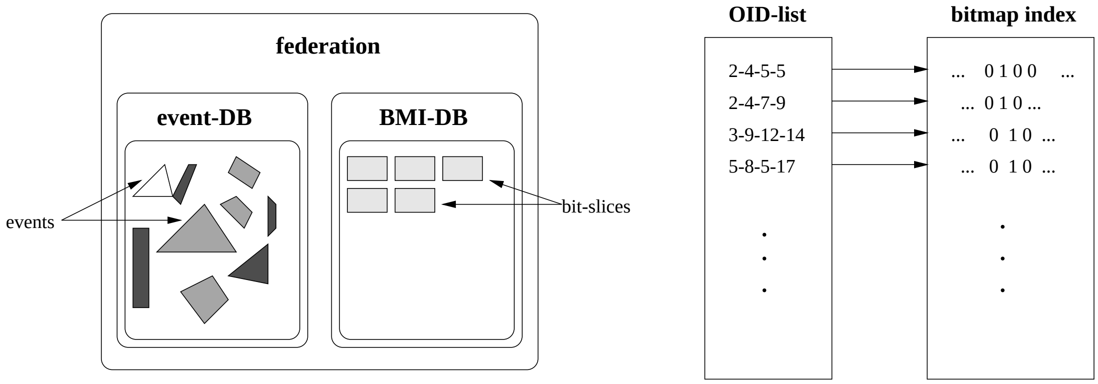

Figure 5.2: Architectural overview of the bitmap index on top of Objectivity/DB.

in separate databases under one federation which in turn is the highest level of abstraction in Objectivity/DB and allows to access physically distributed databases. Note that in Objectivity a database corresponds to one operating system file. From the point of view of the programmer, the whole database system is one logical unit. The main architectural aspects are depicted in Figure 5.2.

The implementation of the event is based on the traditional usage of object databases, i.e. each event is considered as one persistent object. In other words, the event objects are stored according to the generic tag (as discussed in Chapter 4). Throughout the thesis we will use the word event and tag as synonyms since all our investigations of bitmap indices are based on tags which represent a summary of a physics event.

Any persistent object in Objectivity/DB can be directly accessed by its object identifier (OID) which we use for keeping track of the event data. In particular, each physics event is stored as an object and can thus be directly accessed via its OID. This step is necessary, for example, for checking the *candidate slices*.

As we can see on the right side of Figure 5.2, one OID-list is maintained in addition to the bitmap index. For instance, if we want to check the event at position x, we simply refer to the OID list at position x and fetch the event from disk for checking the attribute value against the query condition.

5.4 Brief Justification of the Bitmap Index Approach

Our first focus of interest was the performance comparison of the bitmap index with the sequential scan of Objectivity/DB in order to justify any further research of bitmap indices for HEP data. In particular, we compared the performance of sequentially scanning event data stored event-wise (corresponds to generic tag in Chapter 4) in Objectivity/DB to the performance of reading the data via our bitmap index.

We carried out our benchmarks on a Pentium II 400 under Linux Red Hat 6.1. The bitmap index is implemented on top of Objectivity/DB version 5.1.2. Throughout the rest of this chapter all experiments operate on 10^6 events.

Method	Number of attributes	size [MB]	time [sec]
seq. scan	1	35	36
bitmap index	1	12	9.5
seq. scan	10	73.5	38
bitmap index	10	48	33.5

Table 5.1: Sequential scan vs. bitmap index.

We first report on the performance of two-sided range queries over 1 and 10 attributes. In particular, we are interested in the behaviour of queries with a selectivity of 100%, which can be regarded as the worst case. As for the bitmap index all benchmarks are carried out with 32 bins and the queries cover all attributes.

As we can see in Table 5.4, the size of the event data for 1 and 10 attributes is 35 MB and 73.5 MB respectively. The size of the index is 12 MB and 48 MB respectively, including a constant overhead of 8 MB for the OID-list.

The performance of the bitmap index is in all cases better than the performance of the sequential scan. However, as for the bitmap index not the whole amount of event data is actually accessed but only those of the candidate slices which need to be checked against the query constraint. Obviously, in order to answer a query with a selectivity of 100%, the whole bitmap index must be scanned first in order to sieve out candidates from hits.

We also have to stress that the performance of Objectivity/DB for scanning small objects, i.e. much smaller than the page size, is very low in comparison to the raw sequential I/O for this disk [33].

In our next set of benchmarks we analysed the behaviour of our bitmap index with changing query selectivities. The number of dimensions covered by the query is 10. The number of bins is again 32. The results of this performance study are depicted in Figure 5.3.

From our empirical observations we would conclude that a lower selectivity has a better impact on the performance of a range query. However, taking a closer look at the results reveals that the bottleneck of the bitmap index is the candidate check, which is highly I/O bound. The time which is spent on the Boolean operations for retrieving the final hit- and candidate-slices is very low for this number of events and attributes. Intuitively we would therefore recommend to increase the number of bins and thus decrease the inherent I/O bottleneck due to the selective scan over the event data.

5.4.1 Cost Model for Equality Encoded Bitmap Indices

In this section we will analyse the size of the index and the I/O operations needed in order to evaluate a query via equality encoded bitmap indices. We do not give any details about the number of logical operations since these CPU-operations have only a minor impact on the performance of the index as compared to the more expensive I/O operations [14].

The right number of bit slices (bins) can be regarded as one of the key parameters of this kind of bitmap index. A detailed discussion on the behaviour of the bitmap index with a differ-

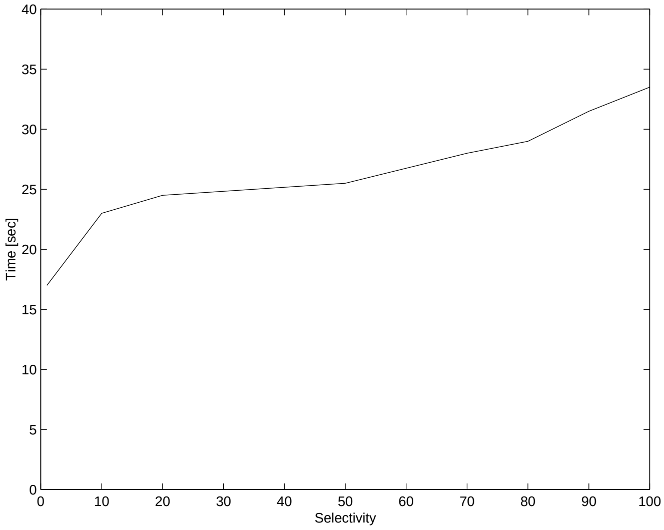

Figure 5.3: Variable query selectivities for two sided-range queries.

ent number of bins and a different number of indexed attributes is vital for the understanding of bitmap indices for any application. To our best knowledge, this kind of investigation has not been done before for this algorithm based on scientific data. The main motivation was a similar implementation of bitmap indices presented in [58] where 20 bins were chosen. However, no analysis or justification for this key parameter is given.

Before we go into detail with analysing the query costs, we first discuss the index size.

5.4.2 Size of the Index

In general, the size (in bytes) of a bitmap index is given by the following formula:

$$size = \frac{O}{8} \sum_{i=1}^{d} b_i \tag{5.1}$$

where O is the number of objects, d the number of dimensions (attributes) and b_i the number of bins for dimension i. For example, the size of a bitmap index for 1,000,000 objects with 25 attributes and 100 bins each is

$$size = \frac{1,000,000}{8} \cdot 100 \cdot 25$$

which is roughly 300 MB. In contrast, the size of the base objects is roughly 120 MB.

When we consider also the OID-list, the size of the index can be computed as follows:

$$size = \frac{O}{8} \sum_{i=1}^{d} b_i + 8O \tag{5.2}$$

We assume that the size of one OID is 8 bytes.

5.4.3 I/O Complexity of the Index

The I/O complexity for evaluating a query via the bitmap index can be separated into two parts:

- *Index Evaluation Phase*
- *Candidate Check Phase*

The *Index Evaluation Phase* corresponds to all I/O operations during scanning the bit slices to evaluate a query. The *Candidate Check Phase* includes the additional I/O operations for checking the candidate objects against the query constraint. In this phase, the base objects must be fetched from disk.

5.4.4 Maximal Page I/O Costs for Index Evaluation Phase

We will now analyse the maximal number of page accesses needed for the *Index Evaluation Phase*. When we assume uniformly distributed and independent data, at most 50% of the bit slices have to be scanned for evaluating one-sided range queries. In other words, for attribute selectivities of 50%, half of the bit slices per indexed attribute have to be scanned. If the attribute selectivity is higher than 50%, then the query can be handled by negating the query expression.

For example, if we assume an attribute range of [0;100], then the attribute query selectivity of the following one-sided range query $a_0 < 30$ is 30%. The attribute selectivity of the next query $a_0 < 63$ is 63% but the query can be evaluated as $a_0 \geq 63$ which results in an attribute selectivity of 27%. The result set needs to be negated afterwards in order to fulfil the query constraint of $a_0 < 63$. As we have seen, in the worst case, 50% of the bit slices need to be scanned for evaluating one-sided range queries via equality-encoded bitmap indices.

In most database management systems the read/write operations are based on the database page level (or block level) rather than on the object level. This means that when a single object on one page is accessed, the whole page has to be read (see Figure 5.3). As for the *Index Evaluation Phase* we need to express the number of bitmaps to be accessed in terms of database pages.

For a one-dimensional query based on a bitmap index with b_i bit slices, in the worst case $n_i = \frac{b_i}{2}$ bit slices have to be scanned.

The number of page accesses p_s for scanning one bit slice is given as:

$$p_s = \frac{O}{8s_p} \tag{5.3}$$

where s_p is the size of one database page.

The I/O costs for evaluating a one-dimensional query are then:

$$p_s = \frac{n_iO}{8s_p} \tag{5.4}$$

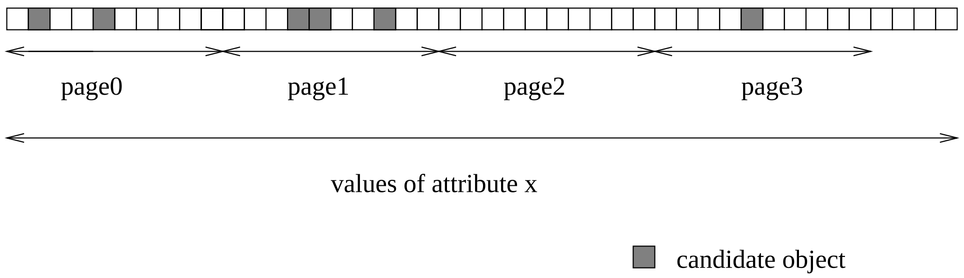

Figure 5.4: Access granularity of a database in terms of pages rather than objects.

where n_i is the number of bit slices to be scanned. n_i must can be calculated from the attribute selectivity. For example, assuming an index of 32 bins and an attribute selectivity of 50%, the number of bit slices to be scanned is 16.

The total I/O costs C_i for the *Index Evaluation Phase* in terms of accessed database pages for handling a d-dimensional query are given as:

$$C_i = \frac{O}{8s_p} \sum_{i=1}^{d} n_i \tag{5.5}$$

where n_i refers to the number of bit slices to be read for attribute (dimension) i.

5.4.5 Page I/O Costs for Candidate Check Phase

The calculation of the page I/O costs for the *Candidate Check Phase* C_c needs some more explanation. Basically, the main bottleneck of bitmap indices on non-discrete attribute values is the additional disk I/O overhead for checking the candidate objects against the query constraint.

Similar to previous studies on cost models for index data structures we assume uniformly distributed and independent data values. The expected number of candidate objects E_c per bit slice (which corresponds to one dimension) is given as

$$E_c = \frac{O}{b_i} \tag{5.6}$$

where b_i is the number of bins for this particular dimension.

Let us assume a bitmap index with 100 bins and 1,000,000 objects, then the number of expected candidate objects per dimension would be 10,000.

5.4.6 Total I/O Costs

The total I/O costs consist of the costs for the *Index Evaluation Phase* and the *Candidate Check Phase* for each dimension, i.e. for each indexed attribute. Depending on the clustering of the base data, the total costs are calculated in two different ways.

Parameter	Description
O	total number of objects
E_c	expected number of candidate objects which need to be checked against query constraint
d	number of dimensions
b_i	number of bit slices of dimension i
s_p	page size (in bytes)
p_i	total number of pages for storing all objects of dimension i
p_s	number of pages for storing one bit slice

Table 5.2: Parameters of the cost model.

Total I/O Costs for Generic Tags

Let us first start with calculating the total costs for object-wise clustered base data, i.e. generic tags.

Equation 5.6 gives the number of expected candidate objects for one dimension. However, we still need to calculate the number of candidate objects for a d-dimensional query. We assume that all query dimensions are "AND"ed together, e.g.

$a_1 < 30$ AND $a_2 > 85$ AND $a_3 < 20$ AND $a_4 > 95$

The total number of candidate objects for evaluating a d-dimensional query is given as:

$$E_{c_{gen}} = O(1 - \prod_{i=1}^{d}(1 - \frac{1}{b_i})) \tag{5.7}$$

where O refers to the total number of objects.

Since for generic tags, all attributes of one event (tag) are clustered together (into one persistent object), we assume that the candidate check is done after computing all candidate objects. We will see later when we discuss the candidate check for sliced tag that the actual fetching of the candidate objects (candidate check) is done for each attribute separately.

Given the number of candidate objects $E_{c_{gen}}$ (see Equation 5.7), the total "candidate I/O costs" $C_{c_{gen}}$ in terms of database pages to be read are [52]:

$$C_{c_{gen}} = p_{tot}(1 - e^{-\frac{E_{c_{gen}}}{p_{tot}}}) \tag{5.8}$$

where p_{tot} denotes the total number of pages for storing the base objects of and e denotes the exponential function. The parameters for furthers equations are listed in Table 5.2.

Finally, the total costs $C_{tot_{gen}}$ in terms of page I/Os is the sum of the I/O-costs for the *Index Evaluation Phase* C_i (see Equation 5.5)and the *Candidate Check Phase* $C_{c_{gen}}$ (see Equation 5.8):

$$C_{tot_{gen}} = \frac{O}{8s_p} \sum_{i=1}^{d} n_i + p_{tot}(1 - e^{-\frac{E_{c_{gen}}}{p_{tot}}}) \tag{5.9}$$

Let us interpret this equation briefly in order to understand the implications of the bitmap index design. As the number of bins increases, the overhead for the index operations increases with constant query selectivities. This is reflected by parameter n_i.

On the other hand, as the number of bins b_i increases, the overhead for the candidate check phase decreases (see Equations 5.6 and 5.8). We will analyse the optimal number of bins for different query selectivities in Section 5.5.

Total I/O Costs for Sliced Tags

As for sliced tags, we assume to evaluate the candidate objects for each dimensions separately since that base objects are clustered attribute wise rather than event-wise (object wise). Given the number of candidate objects E_c as defined in Equation 5.6, the candidate I/O costs $C_{c_{sliced}}$ for one dimension in terms of database pages to be read are [52]:

$$C_{c_{sliced}} = p_i(1 - e^{-\frac{E_c}{p_i}}) \tag{5.10}$$

where p_i denotes the total number of pages for storing the base objects of one dimension. Remember, p_{tot} in Equation 5.8 for generic tags refers to the total number of pages for all attributes.

Consider again the 4-dimensional example query from above:

$a_1 < 30$ AND $a_2 > 85$ AND $a_3 < 20$ AND $a_4 > 95$

Assuming uniformly distributed data values in the range of [0;100] and 100 bins, the query selectivity for each attribute and thus the expected number of candidates for each dimension can be computed.

The expected number of candidate objects for the first attribute according to Equation 5.6 is E_{c1} = 10,000. For the second attribute, the number of expected candidate objects is based on the attribute selectivity of a_1, i.e. on the remaining hits (result set) after evaluating the first dimension) E_{c2} = 10,000 * 0.3 = 3,000. For attribute a_3 and a_4 the expected number of candidate objects are E_{c3} = 3,000 * 0.15 = 450 and E_{c4} = 450 * 0.2 = 90 respectively.

In general, the total number of candidate objects $E_{c_{sliced}}$ for a d-dimensional query is given by:

$$E_{c_{sliced}} = \frac{O}{b_1} + \sum_{i=2}^{d} \frac{O}{b_i} sel_{i-1} \tag{5.11}$$

where sel_i is the selectivity of attribute i.

Finally, the total costs $C_{tot_{sliced}}$ in terms of page I/Os is the sum of the I/O-costs for the *Index Evaluation Phase* C_i and the *Candidate Check Phase* C_c:

$$C_{tot_{sliced}} = \frac{O}{8s_p} \sum_{i=1}^{d} n_i + p_i(1 - e^{-\frac{E_{c_i}}{p_i}}) \qquad (5.12)$$

where E_{c_i} is defined according to Equation 5.11 and p_i is the number of pages for storing the base attribute i.

Similar to Equation 5.9 for generic tags, also for sliced tags the number of bins has a direct impact on the number of I/O operations for handling queries. The main difference, however, is that for sliced tags we perform the candidate check for each attribute separately.

5.5 Analytical Results

We will now evaluate analytically the optimal number of bins for queries of different dimensionality and various selectivities. In particular, we will analyse the I/O costs for evaluating range queries via equality encoded bitmap indices based on generic and sliced tags. The main motivation is to compare the I/O costs to the sequential scan and identify up to which query selectivity the bitmap index has lower I/O costs than the sequential scan.

5.5.1 Equality Encoding on Generic Tags

In our first set of tests we computed the optimal number of bins for the worst case queries, i.e. queries with an attribute selectivity of 50%. All our calculations are based on the cost model presented in the previous sections. The total number of objects O is 10^6 and the size of the database page s_p is 8 KB.

In Figure 5.5 we plotted the I/O costs for handling queries over various dimensions and calculated the optimal number of bins for each of them. In addition, we also plotted the I/O costs for sequentially scanning the generic tags.

As we can see, for one-dimensional queries the optimal number of bins is in the order of 1300. What is more, the resulting I/O costs are far below the sequential scan. For two-dimensional queries, the optimal number of bins is in the order of 220 which is significantly less than for one-dimensional queries. However, the I/O costs are higher.

For higher dimensional queries we can observe that the optimal number of bins gets even smaller. We can also see that for 15-dimensional queries the I/O costs for the optimal number of bins is slightly above the I/O costs for the sequential scan. For 25-dimensional queries the sequential scan always outperforms the bitmap index.

In Figure 5.6 we plotted the same 50% attribute selectivity queries but we split the I/O costs for the bitmap index into following two additive phases, namely

- index-I/O costs, i.e. costs for the *Index Evaluation Phase*
- candidate-I/O, i.e. costs for the *Candidate Check Phase*

We can see that the costs for index-I/O increase linearly with the number of bins (see Equation 5.5) whereas the costs for the candidate I/O decrease (see Equation 5.8).

In our last calculations based on generic tags, we calculated the optimal number of bins for queries with lower attribute selectivities, namely 10%, 1% and 0.5%. (see Figure 5.7). Let us

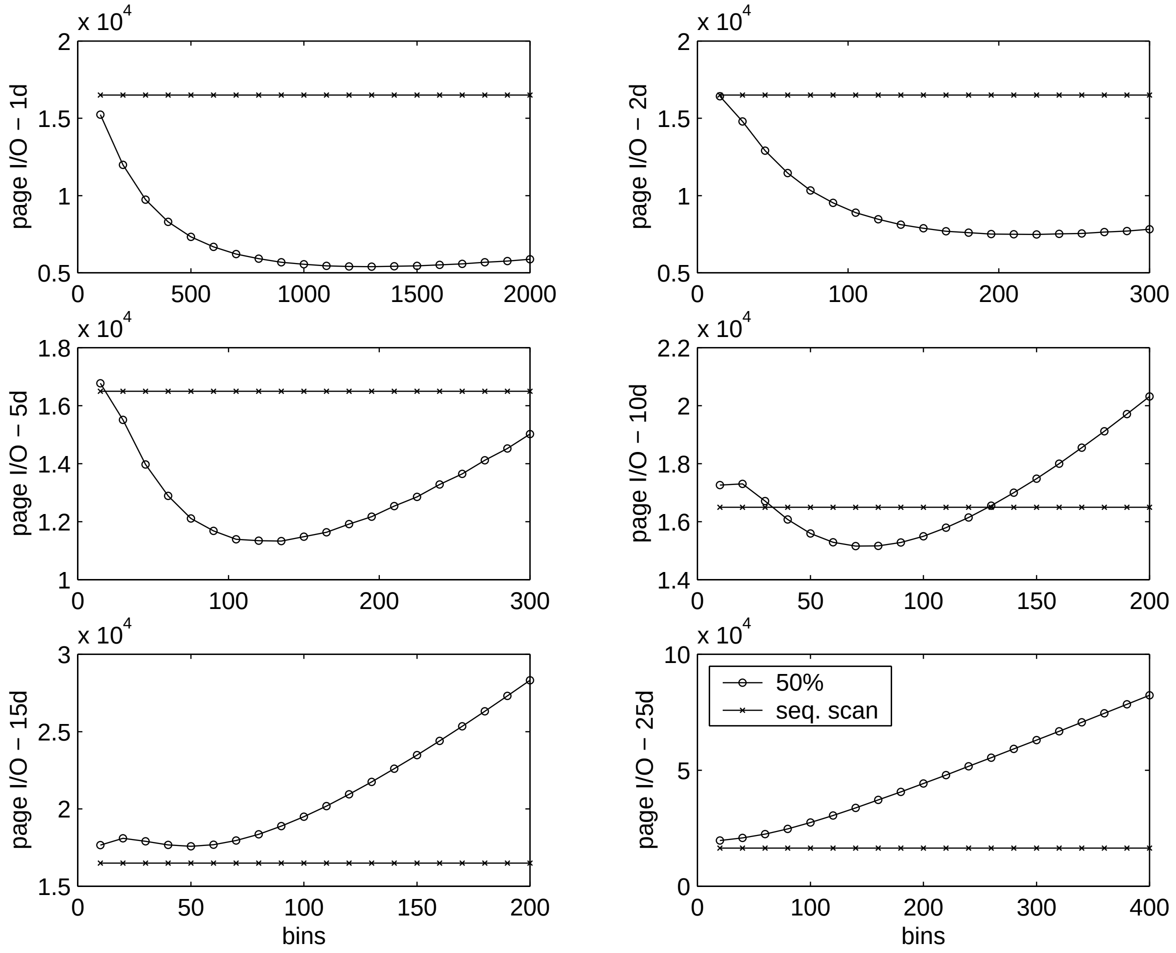

Figure 5.5: Generic tag - I/O costs for optimal number of bins for 50% attribute selectivity (worst case) compared to sequential scan.

start with analysing one-dimensional queries. For queries with an attribute selectivity of 10%, the optimal number of bins is in the order of 700. For attribute selectivities of 1% the optimum is in the order of 2000 bins whereas for queries 0.5% attribute selectivity the optimum is in to order of 3000 bins.

We can observe the same behaviour for multi-dimensional queries. To sum up, for these low selectivities we can always find a optimal number of bins where the I/O costs are below the costs for the sequential scan. For higher selectivities the optimal number of bins is low, whereas for lower selectivities the optimal number of bins is high. The reason for this behaviour is that with high selectivities the index-I/O costs have a larger overhead than the resulting reduced overhead for the candidate I/O.

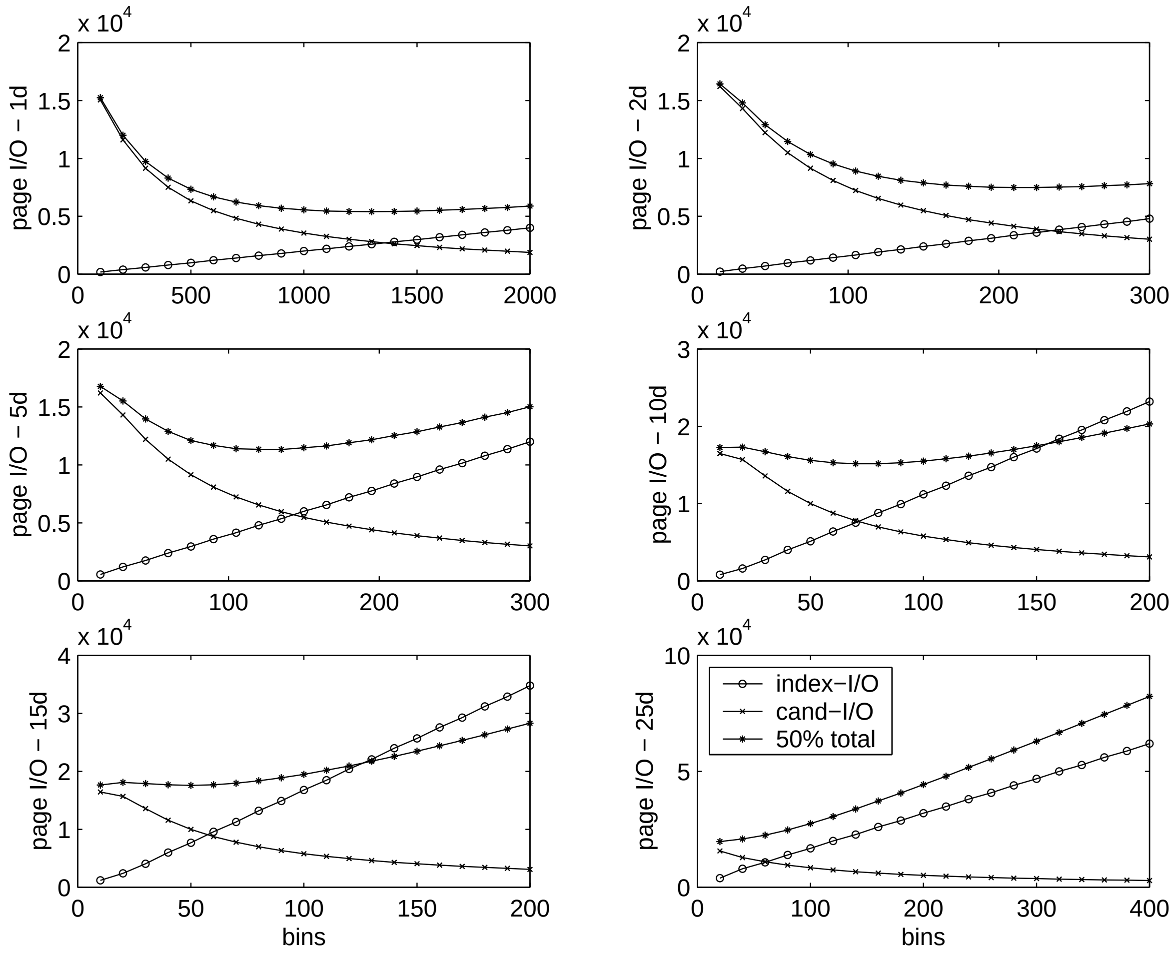

Figure 5.6: Generic tag - I/O costs for optimal number of bins for 50% attribute selectivity (worst case) are split into index-I/O and candidate-I/O.

5.5.2 Equality Encoding on Sliced Tags

We will now look at the results for equality encoded bitmap indices based on sliced tags. Again we start with the analysis of the worst case for one-sided range queries, namely queries with attribute selectivities of 50%.

In Figure 5.8 we can see a major difference to the worst case analysis for generic tags since for sliced tags, no typical optimum can be found. In particular, in all cases the I/O costs are increasing linearly with the number of bins and thus in all cases a low number of bins is desirable. What is more, the index is only more efficient than the sequential scan for queries over 10 dimensions and more. One of the main reasons for the good performance of the sequential scan is the advantage of the clustering for sliced tags.

By looking at the separate cost components for the bitmap index, we can better interpret

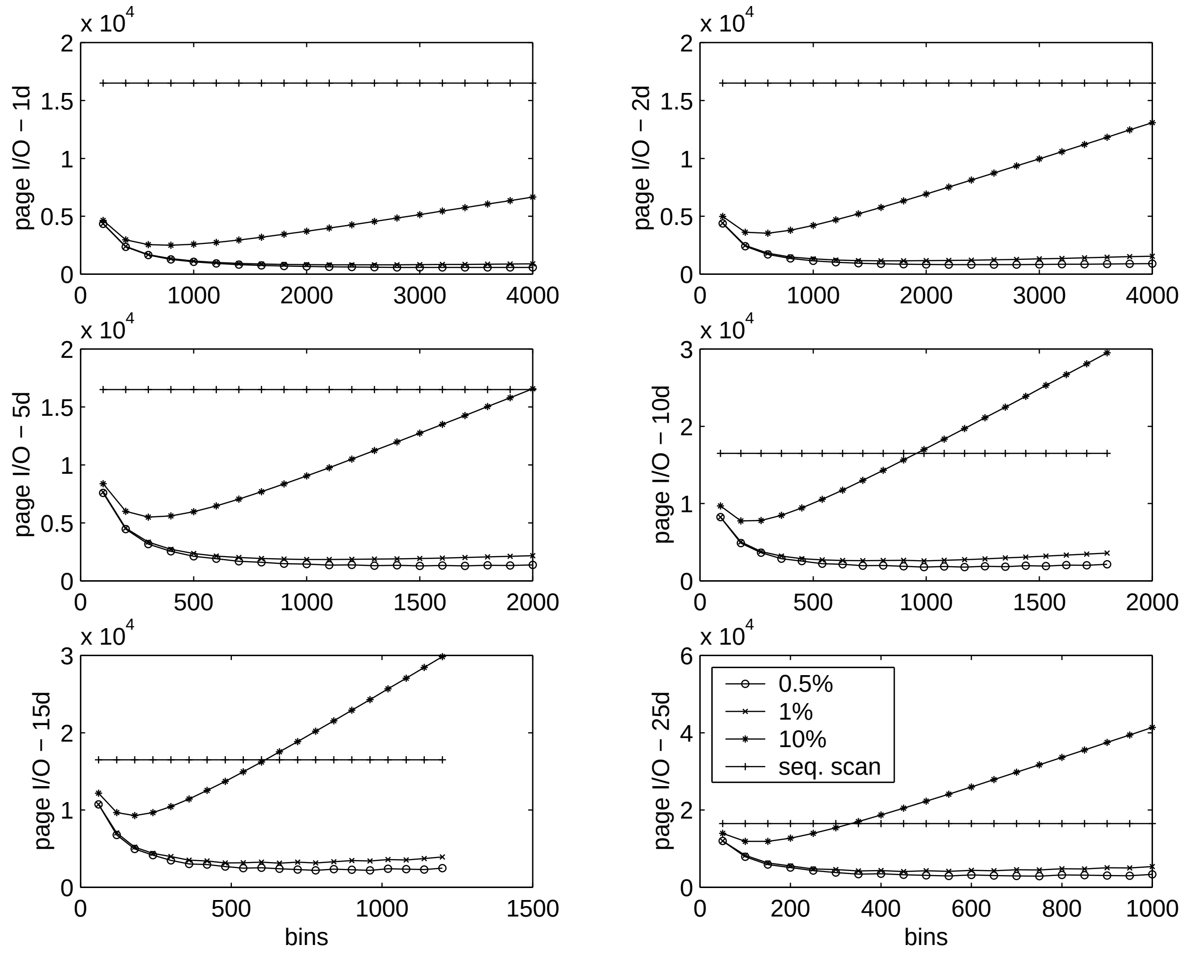

Figure 5.7: Generic tag - I/O costs for multiple attribute selectivities compared to sequential scan.

the results. In Figure 5.9 we see that the index-I/O again linearly increases with the number of bins. This is the same effect as for generic tags. However, since the candidate check is performed for each attribute separately, the costs for the candidate I/O (cand-I/O) are on average lower than the cand-I/O costs for generic tags. What is more, the cand-I/O decreases only marginally as the number of bins are increased.

In our final analysis we evaluated the impact of queries with various selectivities. In short, we can again observe that for high selectivities a lower number of bins is better whereas for queries with low selectivities, a higher number of bins shows better I/O characteristics. One of the most important findings is that especially for high dimensional queries, the equality encoded bitmap index based on sliced tags has significantly lower I/O costs than the sequential scan.

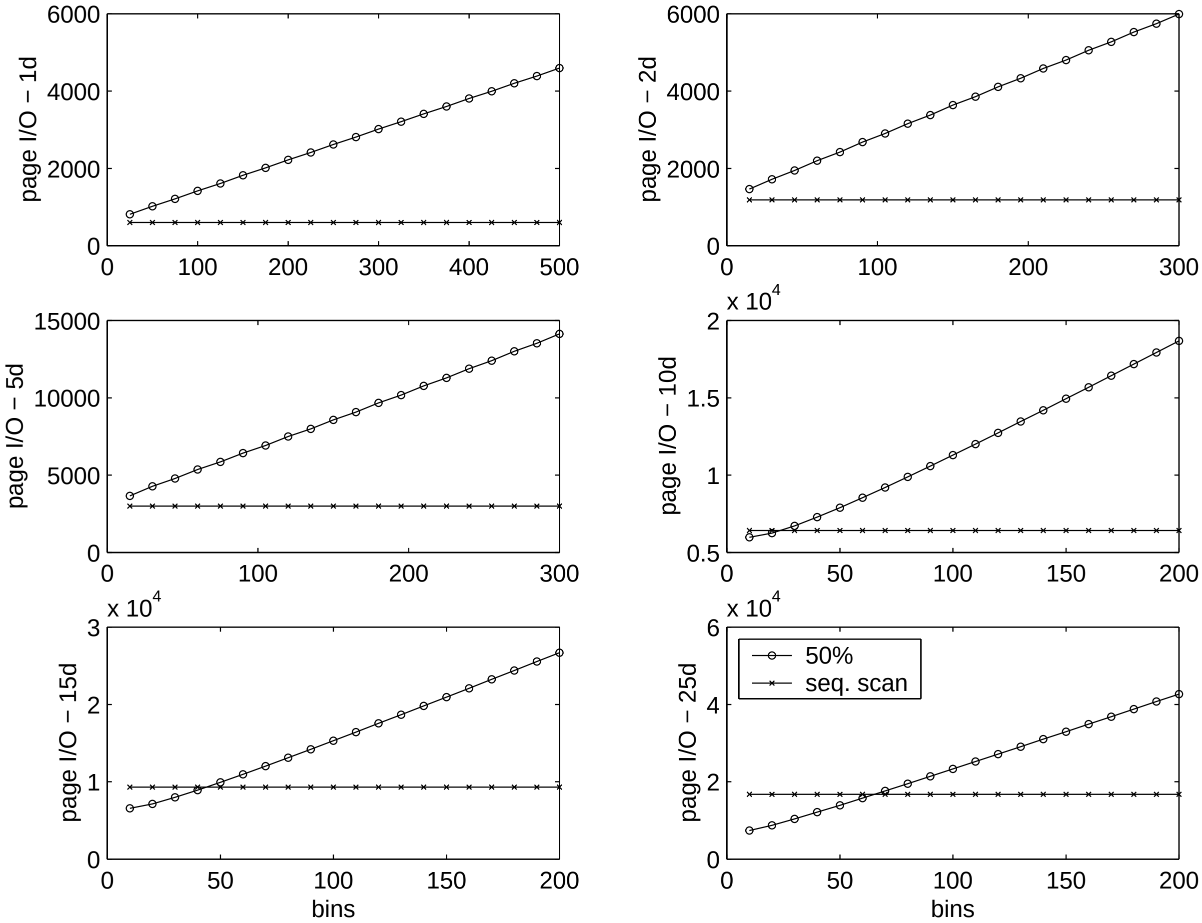

Figure 5.8: Sliced tag - I/O costs for optimal number of bins for 50% attribute selectivity (worst case) compared to sequential scan.

5.5.3 Comparison - Equality Encoding Generic Tags vs. Sliced Tags

When we compare equality encoded bitmap indices based on generic tags with indices based on sliced tags we can see that in most cases the total I/O costs are lower for bitmap indices based on sliced tags. We will thus base all our future analysis on sliced tags.

5.6 Partitioned Range Encoding

Since one-sided range queries (Q_{1r}) are the most common kind of queries in HEP, we also analysed range encoding [14] that performs considerably better than equality encoding. In contrast to [14] who based their optimisation techniques on discrete attribute values (where the problem of candidate and hit slices does not occur), we apply this method to contiguous values and thus a new optimisation method has to be considered. We refer to our approach as

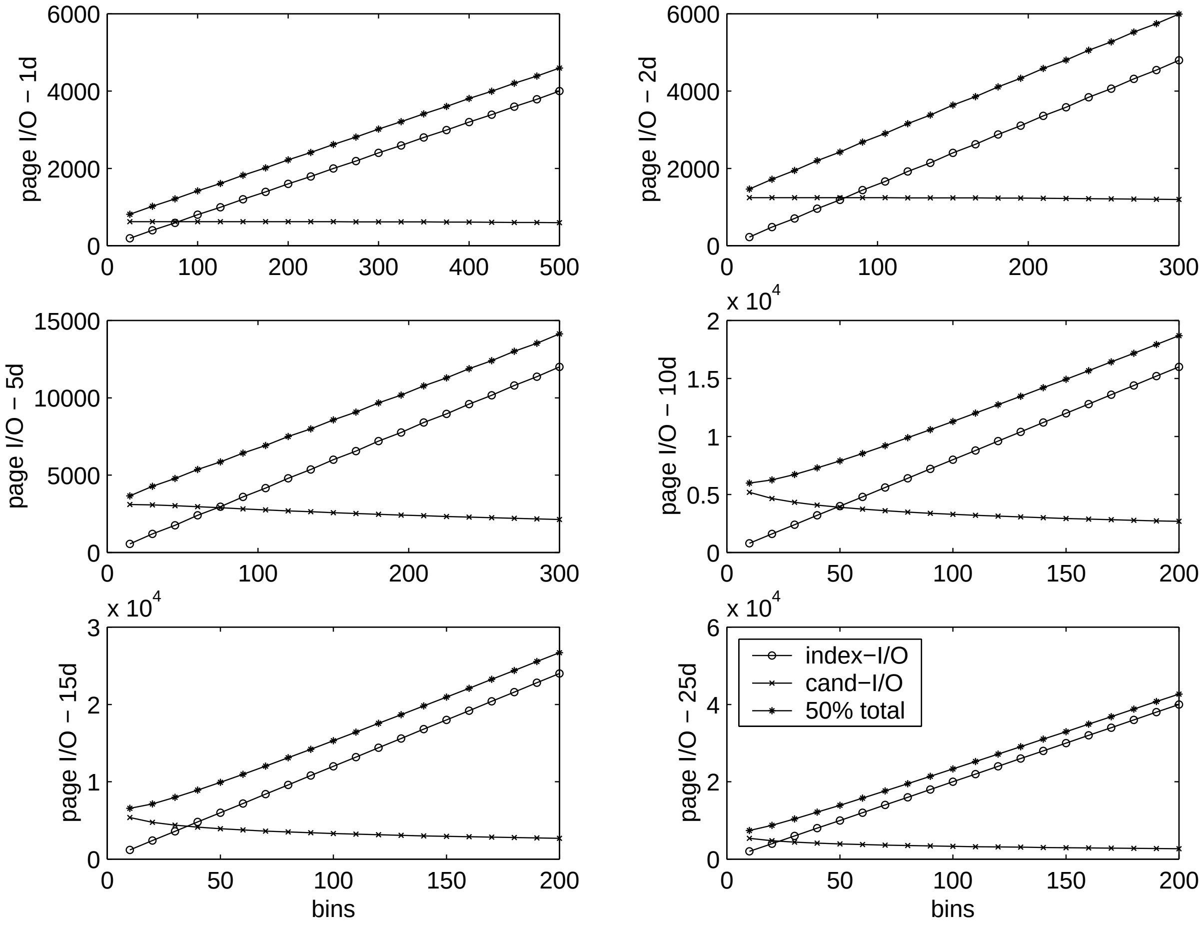

Figure 5.9: Sliced tag - I/O costs for optimal number of bins for 50% attribute selectivity (worst case) are split into index-I/O and candidate-I/O.

partitioned range encoding or in short *range encoding*.

The main advantage over equality encoded bitmap indices is that in the worst case only one bit slice has to be scanned for one-sided range queries per dimension (independent of the selectivity of the query). As for equality encoded bitmaps, in the worst case half of the bit slices have to be scanned.

Since we have already studied the behaviour of equality encoded bitmap indices and raised the I/O problem of candidate slices, we can easily conclude from these observations on the impact of range encoding. As already mentioned, in the worst case one bit slice needs to be scanned for one-sided range queries per dimension. This also implies that we have to consider only one *candidate slice* and no *hit slice* at all.

Let us analyse the performance characteristics of one-sided range queries with both equality encoded and range encoded bitmap indices with variable selectivities, 10^6 objects and 10

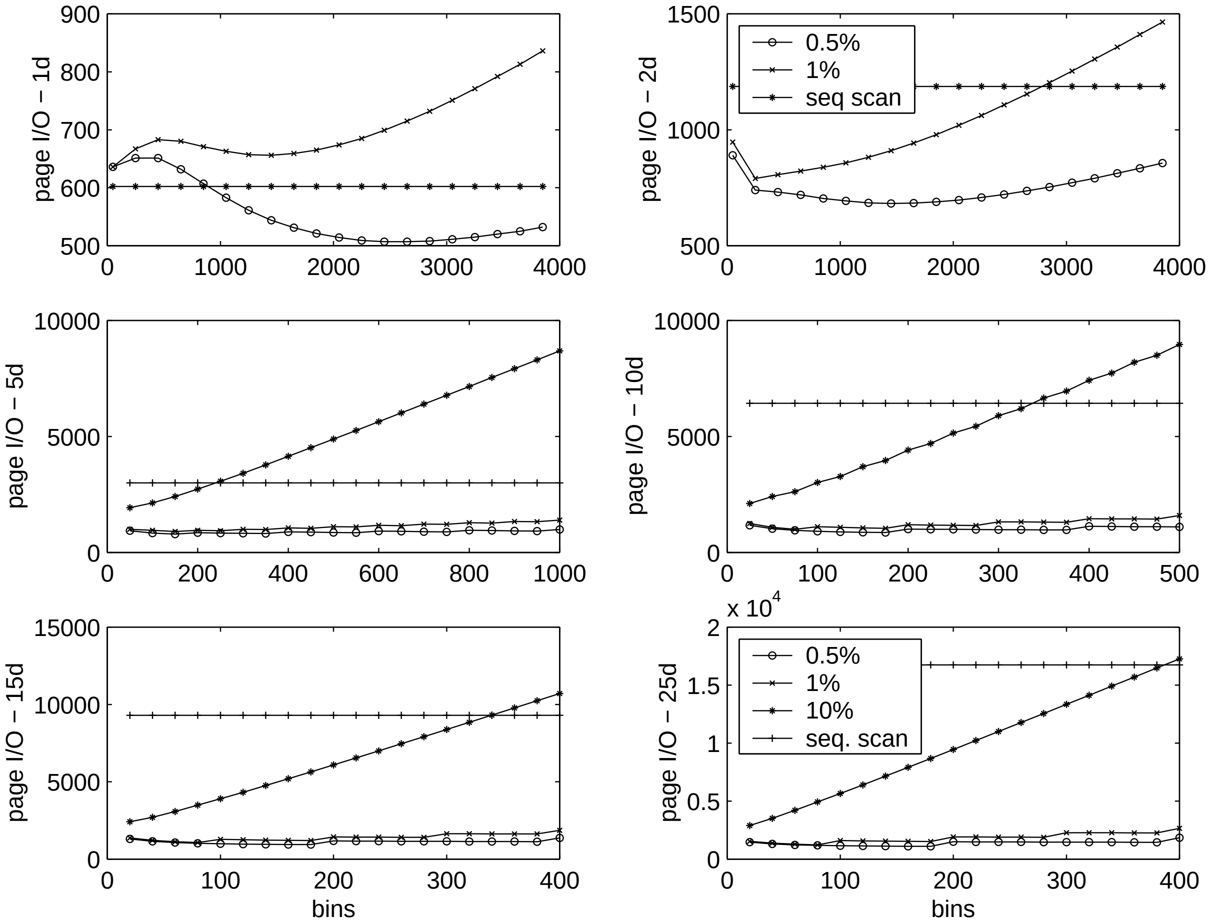

Figure 5.10: Sliced tag - I/O costs for multiple attribute selectivities compared to sequential scan.

indexed attributes. Again we studied the performance characteristics of the bitmap index on uniformly distributed data.

As we can see in Figure 5.11, the query time for equality encoded bitmap indices increases with increasing selectivities (i.e. a higher number of *hit slices* has to be read) whereas the query time for range encoded bitmap indices is more or less constant for all selectivities (since no *hit slices* at all have to be scanned).

The fact that only one bit slice needs to be scanned gives us much more freedom in extending the number of bins until the theoretical maximum of the cardinality of the attribute value. Since we are dealing with contiguous values, we do not have a finite cardinality and hence are mainly "restricted" by the space complexity. We thus end up in a "practical" optimisation problem, that is constraint by the available disk space.

Experimentally we can show that the behaviour of a range encoded bitmap index both for Q_{1r} and Q_{2r} corresponds to a partial scan over the event data. The performance character-

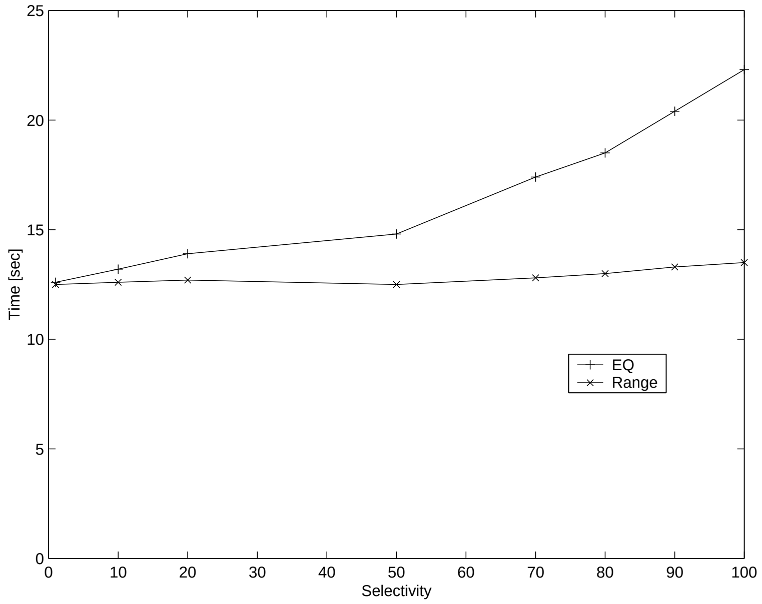

Figure 5.11: Partitioned Equality Encoding (EQ) vs. Partitioned Range Encoding (Range).

istics are thus highly dependent on the characteristics of the underlying OODBMS, namely Objectivity/DB and the disk. In particular, for page selectivities between 5% and 100%, the read rate is almost linear. However, a significant speedup is achieved, if the page selectivity is smaller than 5% [33] which in turn is very common for multi-dimensional range queries in HEP analysis. A detailed analysis on range encoded bitmap indices is given in the next chapter.

5.7 Conclusions

We have given analytical performance studies of bitmap indices for scientific data and pointed out the main difference to other studies on bitmap indices - those that concentrated on only discrete attribute values. The main bottleneck has been shown to be the checking of the candidate slices due to the additional I/O for fetching the event data from disk in addition to the I/O for the bitmap index.

We have designed and implemented our bitmap indices on top of a commercial object database management system, namely Objectivity/DB and used different bitmap encoding techniques for our analysis. We presented a novel cost model for evaluating equality encoded bitmap indices and showed analytically the optimal number of bins for various multi-dimensional range queries with different query selectivities. This optimum can be regarded as a trade-off between a high number of candidates and consequently more I/O on the event data vs. a low number of candidates and therefore a higher number of bins.

Since HEP queries are mainly one-sided range queries, we also studied *partitioned range encoding*. We showed that the performance of range encoded bitmap indices clearly outperforms equality encoded bitmap indices.

Chapter 6

Generic Range Encoding

6.1 Introduction

In the previous chapter we discussed *equality encoded* bitmap indices and introduced a cost model for calculating the optimal number of bins for various queries. We also gave a brief outlook to *range encoded* bitmap indices. The advantage of this encoding technique is that in the worst case only one bit slice needs to be scanned for evaluating one-sided range queries. As for *equality encoded* bitmap indices in the worst, half of the bit slices need to be scanned

Chan and Ioannidis [14] introduced an index evaluation algorithm based on range encoding which guarantees that only one bit slice needs to be scanned for evaluating one-sided range queries. However, the algorithm is designed for *discrete attribute values* but does not take into account *non-discrete attribute values* such as floating point values which are very typical for scientific data.

In this chapter, we present a novel algorithm for processing both one-sided and two-sided range queries via bitmap indices over floating point values. Our algorithm is called `GenericRangeEval` since it is a generalisation of the other algorithms [14] for one-sided range queries and supports efficient query processing for scientific data.

We also extend the cost model introduced in the previous chapter for predicting the performance of *range encoded* bitmap indices based on uniformly distributed and independent data values. We evaluate the minimal required index size for querying high-dimensional search spaces and discuss the trade-off between index size and performance.

Besides a detailed description of our analytical model we also give a performance study and show up to which dimensionality and query selectivity the bitmap index performs better than sequentially scanning the base objects.

6.2 Example: Range Encoding for Non-Discrete Attribute Values

As we mentioned already previously, the range-encoding technique introduced by [14] is only optimised for discrete attribute values. The main difference of indexing non-discrete data is that not the attribute cardinalities can be encoded but attribute ranges. For instance,

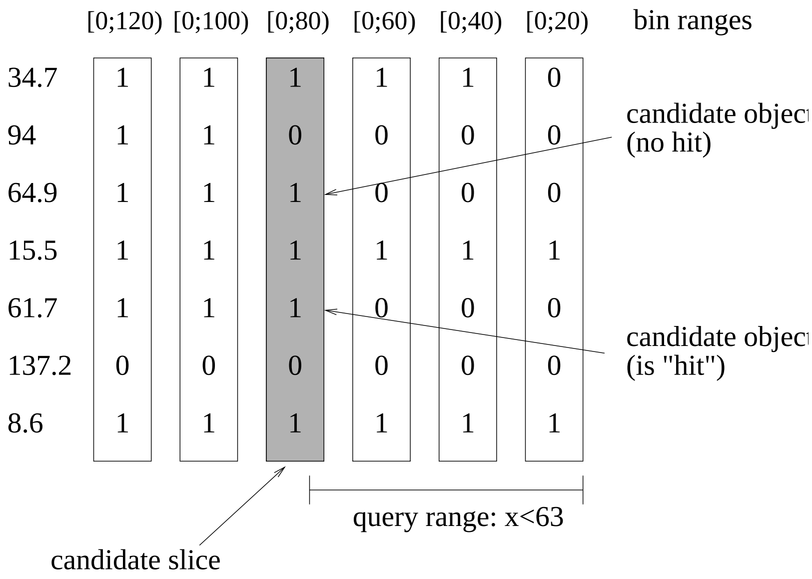

Figure 6.1: One-sided range query on a range encoded bitmap index.

conventional bitmap indices are most efficient for attributes with low cardinalities. A major problem of encoding attribute ranges rather than attribute cardinalities is that the results yielded from the bitmap index are not definite hits. They still need to be checked against the query constraint to decide whether they can be kept in the foundset or whether they must be dropped.

In the previous chapter we discussed this problem for *equality-encoded* bitmap indices and called it *candidate check problem*. Let us explain it here for *range encoded* bitmap indices [68].

Assume a range encoded bitmap index with 6 bins. According to [14] this index would represent an attribute with cardinality 7. In our case, 7 attribute ranges are represented. If we assume that our search space is in the range of [0;140], then each bin represents a range that is a multiple of 20 as depicted in Figure 6.1.

When we issue following query $x < 63$, the preliminary foundset would be bin 3 (highlighted in grey). Since this bin represents values in the range of [0;80), the foundset contains not only so-called *hits* (values 34.7, 15.5 and 8.6) but also *candidates* (values 64.9 and 61.7) that need to be checked against the query constraint. Candidates are all those attribute values in the current bin B_i that are not represented in bin B_{i-1}. We assume that the first bin is denoted by B_0. As we can see, for instance, the candidate value 64.9 needs to be discarded since it does not fulfil the query constraint whereas the candidate 61.7 fulfils the query constraint. Throughout the thesis we will refer to this process of sieving out the hits from the candidates as the *candidate check*.

Input:	
D	number of dimensions, i.e. number of indexed attributes
n_d	number of bit slices in dimension d
w_d	width of the bit slices in dimension d
$B_{d,s}$	s^{th} bit slice of the bitmap index for the d^{th} dimension
op_d	predicate operator for dimension d where $op_d \in \{<, \leq, >, \geq\}$
q_d	query range of dimension d
Output:	
H	bit slice representing hit objects
Internal:	
$T, T_1, T_c, T_{ch}, T_{ret}$	temporary bit slices
C_d	bit slice representing candidate objects for dimension d
CH_d	bit slice representing candidate and hit objects for dimension d

Table 6.1: Parameters for algorithm `GenericRangeEval`.

6.3 Generic Range Encoding - A Novel Algorithm

Range encoding is currently considered as the best encoding technique for evaluating one-sided range queries [14] via bitmap indices. A first algorithm called `RangeEval` was presented by [52]. This algorithm applied to range encoded bitmap indices guarantees that at most two whole bitmaps need to be scanned per indexed attribute independent of the query selectivity. Note that for "simple bitmap" indices in the worst case $\left\lfloor \frac{|A|}{2} \right\rfloor$ bitmaps need to be scanned where $|A|$ is the attribute cardinality. [14] presented an improved evaluation algorithm called `RangeEval-Opt` [14] that reduces the number of bitmap operations by about 50% and requires only one full bitmap scan for evaluating one-sided range queries independent of the query selectivity.

The main motivation for extending `RangeEval-Opt` is that it only supports the evaluation of discrete attribute values whereas in many scientific applications non-discrete data values, for example floating point values, are very common. Thus, the bins (slices) of the bitmap index represent attribute ranges rather than attribute cardinalities. Hence, `RangeEval-Opt` needs to be extended by an explicit operation in order to separate *candidate objects* from *hit objects*.

Let us now describe our extended version of `RangeEval-Opt` which we call `GenericRange-Eval` [68]. The parameters are given in Table 6.3. We distinguish between two main steps, namely

- *Query Mapping Step* and
- *Query Evaluation Step*

Starting from the actual user query, the first step of the algorithm is to map the query range to the affected bit slice s. Throughout the thesis bin and bit slice will be used as synonyms. We assume equi-depth bit slices, i.e. the ranges of all bit slices have the same width w, since it guarantees a constant number of candidate objects in each bin for uniformly distributed data. A typical *mapping function* looks like follows:

$$s(q) := \left\lfloor \frac{q-l}{w} \right\rfloor \tag{6.1}$$

where q is the actual upper/lower bound of the query, for instance for the query $x < 63$ it would be 63, l is the lower bound of the search space for this dimension and w the width of the bit slices. In the following we assume that the lower bound of the search space is zero.

In the second step, the query is evaluated, i.e. the actual range encoding algorithm can be applied. Again, it must be considered that the bins represent attribute ranges and that an additional "candidate check" needs to be done. We will explain this by an example.

Assume that we want to evaluate the query expression from the previous example, namely: $x < 63$. Further assume the range encoded bitmap index as depicted in Figure 6.1, i.e. our search space for this particular dimension is in the range of [0;140] and each bit slice has a width of 20.

In the first step, the upper bound of the query, namely 63, needs to be mapped to the corresponding bit slice s. We obtain this by applying Function 6.1: $\left\lfloor \frac{63-0}{20} \right\rfloor = 3$ (which corresponds to line 1.3 of `GenericRangeEval`. Thus, bit slice B_3 is our "candidate slice" which we store temporarily in T (see line 1.22). Due to the binning we have chosen, slice B_3 represents objects that have values less than 80 (see Figure 6.1), however we are interested in all objects with values less than 63. This means, our temporary bit slice T contains candidates (represented by B_3) and hits (represented by B_2). We refer to this bit slice now as "candhits" slice CH_d (see line 1.27 - CH_d = T). The actual "candidates" C_d are yielded by "XOR"ing B_3 with B_2 (see lines 1.30, 1.36 and 1.31). The resulting objects represented by C_d have the values 64.9 (object 3) and 61.7 (object 5) respectively.

What follows is the typical "candidate check" where all candidate objects in C_d are checked against the query constraint (line 1.37) and if the candidates are do not fulfil the query constraint, the corresponding bit in the "candhits-slice" CH_d is turned off. In our example, object 3 which has the value 64.9 does not fulfil the query constraint $x < 63$ and thus the 3rd bit in CH_d is switch off.

To demonstrate a further aspect of the algorithm, consider following query $x > 63$ rather than $x < 63$ as previously. The query mapping step (line 1.3) again yields slice 3 but due to the "greater operator" in the query predicate, the temporary bit slice T is now $T = B_2$ (see lines 1.6 and 1.22). However, B_2 represents all objects with values less than 60. By negating this bit slice (line 1.11) we obtain all objects with values greater or equal 60, i.e. range [60;140). We call this bit slice again "candhits" (line 1.27 $CH_d = T$).

The "candidate mask" is similar to the previous example. After increasing the current bit slice by one (see line 1.29), the actual candidate objects without including the hit objects are yielded by $C_d = B_3$ XOR B_2 (lines 1.36 and 1.31).

It is important to note that additional "operator checks" need to be included into the algorithm which guarantee that the query range is mapped correctly to the corresponding bit slice and is then also evaluated correctly. Just to give one example of this additional complexity, assume that instead of $x > 63$ our query is $x \leq 60$. In that case, bit slice B_2 contains (with

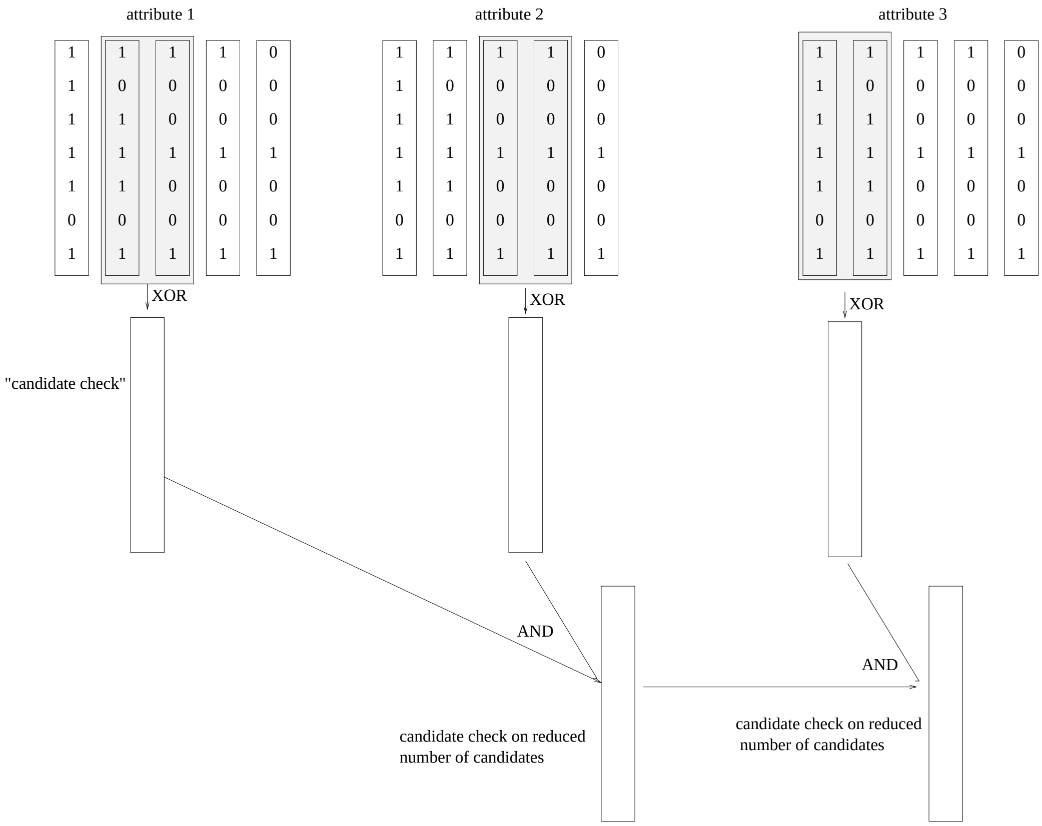

Figure 6.2: Candidate check in multi-dimensional space. For each attribute the candidates are checked separately.

certainty) only hits whereas bit slice B_3 is our “candhits” slice CH_d where we would need to filter out the hit objects.

The behaviour of the bitmap index in the multi-dimensional space is depicted in Figure 6.2. Note that for each attribute the candidate check is done separately, e.g after “XOR”ing the “candhits” slice with the “previous” slice as shown for the query $x < 63$. However, for all remaining attributes, the bit slice which is yielded after “XOR”ing, is “AND”ed together with the “global” hit slice H (see line 1.14). This means, for example, that for attribute 2 only these candidate objects need to be checked against the query constraint that are hits of attribute 1. The resulting positive effect of this approach is that with a low “attribute query selectivity” the number of candidate objects for each further dimension gets reduced. Finally, the hits of each dimension are “AND”ed together (line 1.16).

The whole algorithm for evaluating one-sided range queries based on range-encoded bitmap indices is shown below. Note that for simplicity we assume that all attributes are global so that parameters do not need to be passed explicitly when a function is called.

Algorithm: `GenericRangeEval` for one-sided range queries

1.01) for d=0 to D-1

evaluate_query()

Function: evaluate_query()
s = q_d mapped to the bit slice
$s_0 = s$; cand_mask=false;
if $op_d \in \{>, \geq\}$ OR ($op_d \in \{<\}$ AND (q_d mod w_d == 0)) then
  s=s-1
  if $op_d \in \{<\}$ AND $q_d > n_d * w_d$ then s=s+1
if (s<0) then s=0

if $op_d \in \{=\}$ then evaluate_equality(T)
else evaluate_cands()
if $op_d \in \{>, \geq\}$ then T=NOT T
merge_candhits()
if (cand_mask==false) then mask_cands()
if ($d > 0$) then $C_d = C_d$ AND H
candidate_check()
$H = H$ AND CH_d

Function: evaluate_cands()
if ($s < n_d$) then
  if $op_d \in \{>, \geq\}$ AND $s_0 == 0$ then
    T=set all bits to 0
    C_d=B_0
    cand_mask=true
  else T=$B_{d,s}$
else
  T=set all bits to 1
  C_d=NOT $B_{d,n_{d-1}}$
  cand_mask=true

Function: merge_candhits()
$CH_d = T$

Function: mask_cands()
if op_d <> $\{=\}$ then
  if $op_d \in \{>, \geq\}$ AND s_0 <> 0 then s=s+1
  evaluate_equality(T_1)
  C_d=T_1
else C_d=T
cand_mask=true

Function: evaluate_equality(T_{ret})
if (s==0) then return T_{ret}=$B_{d,s}$

1.35) else if (s==n_d) then return T_{ret}=NOT $B_{d,s-1}$
1.36) else return T_{ret}=$B_{d,s}$ XOR $B_{d,s-1}$

Function: candidate_check()
1.37) check C_d against query constraint
1.38) if candidate is no hit then turn off bit in CH_d

The algorithm described above handles the case of one-sided range queries. For the slightly more complex case of two-sided range queries, a few changes need to be made in three functions. In short, the main function `evaluate_query` needs to be called twice, namely the first time for the "lower range" (line 2.04) and the second time for the "upper range" (lines 2.05 - 2.07) of the query predicate. When then "upper range" is evaluated, the "candhits" of the particular dimension must be "AND"ed together (line 2.20), whereas the "cands" must be "OR"ed together (lines 2.07, 2.15, 2.27 and 2.32).

Algorithm: `GenericRangeEval` for two-sided range queries
Function: evaluate_cands()
2.01) if ($s < n_d$) then
2.02) if $op_d \in \{>, \geq\}$ AND $s_0 == 0$ then
2.03) T=set all bits to 0
2.04) if (lower_range) then $T_c = B_0$
2.05) else
2.06) $C_d = B_0$
2.07) $C_d = C_d$ OR T_c
2.08) cand_mask=true
2.09) else T=$B_{d,s}$
2.10) else
2.11) T=set all bits to 1
2.12) if (lower_range) then T_c = NOT $B_{d,n_{d-1}}$
2.13) else
2.14) C_d = NOT $B_{d,n_{d-1}}$
2.15) $C_d = C_d$ OR T_c
2.16) cand_mask=true

Function: merge_candhits()
2.17) if (lower_range)then $T_{ch} = T$
2.18) else
2.19) $CH_d = T$
2.20) $CH_d = CH_d$ AND T_{ch}

Function: mask_cands()
2.21) if op_d <> $\{=\}$ then
2.22) if $op_d \in \{>, \geq\}$ AND s_0 <> 0 then s=s+1
2.23) evaluate_equality(T_1)

```
2.24)     if (lower_range)then T_c = T_1
2.25)     else
2.26)       C_d = T_1
2.27)       C_d = C_d OR T_c
2.28)   else
2.29)     if (lower_range)then T_c = T 2.30)     else
2.31)       C_d = T
2.32)       C_d = C_d OR T_c
2.33)   cand_mask=true
```

6.4 Cost Model for `GenericRangeEncoded` Bitmap Indices

In the previous chapter we discussed the optimal number of bins for equality encoded bitmap indices and introduced a cost model for studying analytically the I/O complexity of the bitmap index. We will now extend this cost model for range encoded bitmap indices based on the novel algorithm called `GenericRangeEval`.

6.4.1 I/O Complexity of the Index

In order to evaluate the I/O complexity of the bitmap index including the overhead for the candidate check, we separate the algorithm `GenericRangeEval` into two phases, namely

- *Index Evaluation Phase*
- *Candidate Check Phase*

Let us first start with the *Index Evaluation Phase*, i.e. the number of I/O operations needed for scanning the bitmap index in order to evaluate a query. The *Index Evaluation Phase* does not include the I/O operations for the candidate check.

According to [14] only one full bit slice scan is needed for evaluating a one-sided range query with a conventional range-encoded bitmap index. However, `GenericRangeEval` requires slightly more bit slice scans due to the masking of candidate objects. In particular, one bit slice scan is needed for identifying the "candhits" slice (see line 1.22). In addition, one or two bit slice scans are required for filtering out (masking) the candidates from the "candhits" slice (lines 1.34 - 1.36). To sum up, three bitmaps scans are required for a one-dimensional one-sided range query.

Informally for a d-dimensional query, the following number of bit slice scans b are required in the worst case:

$$b = 3 \cdot bitmap \cdot d \tag{6.2}$$

where *bitmap* refers to a full bitmap scan and d to the number of dimensions covered by the query. We will interpret *bitmap* in terms of page-I/Os in the next sub-section.

Parameter	Description
O	total number of objects
E_c	expected number of candidate objects which need to be checked against query constraint
d	number of dimensions
b_i	number of bit slices of dimension i
s_p	page size (in bytes)
p_i	total number of pages for storing all objects of dimension i
p_s	number of pages for storing one bit slice

Table 6.2: Parameters of the cost model.

After the index evaluation phase, the candidate objects are checked against the query constraint. The number of candidates depends on the query selectivity and the number of bins (bit slices).

6.4.2 Page I/O Costs for Index Evaluation Phase

Similar to the previous chapter, for the *Index Evaluation Phase* we need to express the number of bitmaps to be accessed in terms of database pages. In addition, the number of pages for reading the expected number of candidates during the *Candidate Check Phase* needs to be calculated. By adding the number of page I/Os for these two phases, the costs of accessing the data objects via the bitmap index can be compared to the costs of accessing the data directly without the index.

The parameters of our model are described in Table 6.4.2. If we substitute *bitmap* in Equation 6.2 with p_s, the I/O costs in terms of database pages for the *index evaluation phase* C_i can simply be computed as:

$$C_i = 3p_s d \tag{6.3}$$

where p_s is the number of database pages for storing one bit slice and is given as:

$$p_s = \frac{O}{8s_p} \tag{6.4}$$

Note that we divide O by 8 in order to reflect the size of a bit slice in bytes.

6.4.3 Page I/O-Costs for Candidate Check Phase

Basically, the main bottleneck of bitmap indices on non-discrete attribute values is the additional disk I/O overhead for checking the candidate objects against the query constraint. A straightforward approach to relax this bottleneck is to increase the number of bins and thus the number of candidate objects in each bin. The main question to be answered is what is the minimal index size needed and which index size is "practically feasible". The equation for

#bins	index size [bytes]	#cand. objects	#pages to read	page selectivity
10	1'179'648	100'000.00	620	1
50	6'422'528	20'000.00	620	1
100	12'976'128	10'000.00	620	1
200	26'083'328	5'000.00	620	1
500	65'404'928	2'000.00	596	0.960276
1'000	130'940'928	1'000.00	497	0.800692
1'500	196'476'928	666.67	409	0.658794
2'000	262'012'928	500	344	0.553561
4'000	524'156'928	250	206	0.331839

Table 6.3: Effect of number of bins on the number of candidate objects for one dimension.

calculating the I/O overhead for the *Candidate Check Phase* corresponds to Equations 5.10 and 5.11 for sliced tags of the previous chapter. For ease of understanding, we will repeat them here again:

$$C_{c_{sliced}} = p_i(1 - e^{-\frac{E_c}{p_i}}) \tag{6.5}$$

and

$$E_{c_{sliced}} = \frac{O}{b_1} + \sum_{i=2}^{d} \frac{O}{b_i} sel_{i-1} \tag{6.6}$$

For all our tests assume 10^6 objects and a database page size s_p of 8KB. Thus, the base objects for one dimension comprises 620 pages, in our example with 100 bins we need to read 620 pages for the candidate check. In other words, with this low number of bins, the bitmap index does not yield any improvement over sequentially scanning over the base objects without using an index.

In Table 6.4.3 we show the effect of various numbers of bins on the number of pages to be read (page selectivity) during the candidate check.

As we can see, with 1000 bins only 497 pages need to be read which corresponds to a page selectivity of 80%. With 2000 bins the page selectivity drops to 55%. However, the main disadvantage is certainly the size of the index. For example, with 1000 bins the size of the index is about 25 times the size of the base objects. In short, we can conclude that this kind of index is not very "practical" for one-dimensional queries, however, the bitmap index shows its strength in high-dimensional space with queries of relatively low selectivities (see Figure 6.2). We will go into further detail in the next section.

The total costs C_{tot} for evaluating one-sided range queries in terms of page I/Os is the sum of the I/O-costs for the *Index Evaluation Phase* C_i (see Equation 6.3) and the *Candidate Check Phase* C_c (see Equations 6.5 and 6.6) :

$$C_{tot} = 3p_s d + \sum_{i=1}^{d} p_i (1 - e^{-\frac{E_{c_i}}{p_i}}) \tag{6.7}$$

where p_s is defined according to Equation 6.4. E_{c_i} is defined according to Equation 6.5.

6.5 Analytical Results

On presenting a cost model for predicting the I/O complexity of the bitmap index, we will now evaluate analytically the performance of the bitmap index for multi-dimensional queries.

During our tests we carried out one-sided range queries over various dimensions starting from 1 to 25. For each dimension we varied the selectivity from 10% up to 100%. Similar to the previous chapter, the selectivity of the queries is depicted as the selectivity per dimension. For example, in the case of 5 dimensions the query selectivities for each dimension range from 0.1 to 1.0. If we calculate the corresponding total selectivity for uniformly distributed and independent data values then it ranges from 0.1^5 (0.00001) to 1.0^5 (1.0).

In particular, we are interested in studying following additive phases for evaluating a query via bitmap indices:

- Index-I/O: Overhead for scanning the bit slices of the index.
- Candidate-I/O: Index-I/O including the I/O operations for checking the candidate objects against the query constraint.
- Hit-I/O 1: Candidate I/O including the I/O operations for retrieving the hits.
- Hit-I/O 2: Candidate I/O including the I/O operations for retrieving the hits of only one attribute.

We also varied the number of bins from 10 to 1000 to study the effect on the candidate I/O. Finally, we plotted the results for the theoretical maximum of 1,000,000 bins, i.e. for each attribute value one bin.

In Figure 6.3 we plotted the page I/O costs for queries over multiple dimensions against the bitmap index with 10 bins per indexed attribute. For each dimension we can observe a constant index-I/O overhead independent of the query selectivity. As we can see in the cost model (see Equation 6.7), for each dimension at most 3 bit slices need to be scanned in order to evaluate any kind of one-sided range query.

The page I/O costs for the candidate I/O are always higher than the costs for the sequential scan up to three-dimensional queries (see Figure 6.3). For five-dimensional queries, the candidate I/O is below the sequential scan up to an attribute selectivity of 20%. For ten and 25 dimensions, the candidate I/O overhead is below the sequential scan up to 60% and 80% attribute selectivity respectively. In short, due to the low number of bins, the candidate I/O overhead gets only outweighed for high-dimensional queries.

We also studied the impact on the costs for retrieving the hits. *hit I/O 1* shows the I/O costs for retrieving the hits of each indexed attribute after all candidate objects for each attribute are evaluated.

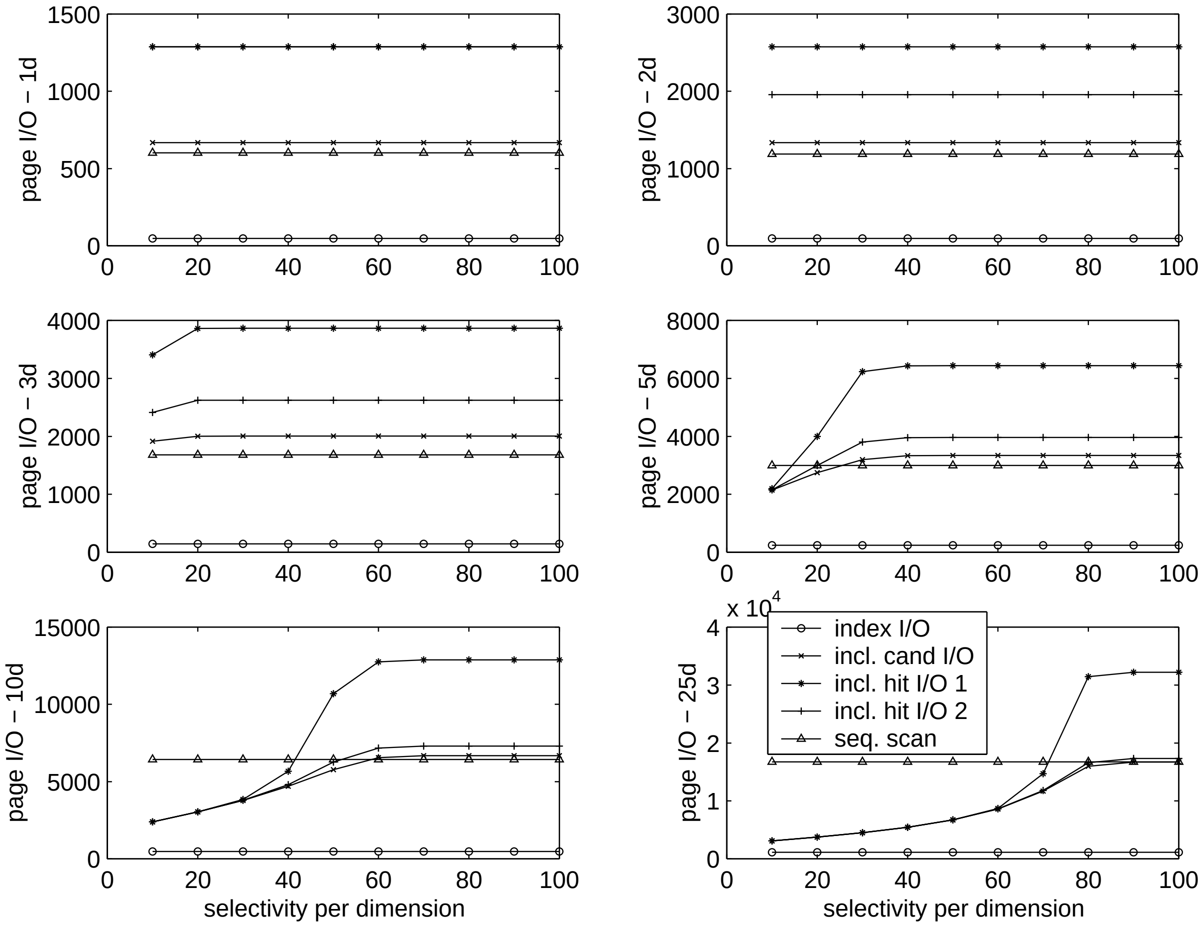

Figure 6.3: Analytical results: Page I/O for queries over multiple dimensions with various selectivities. 10 bins.

In High Energy Physics it is very common to perform multi-dimensional queries but later on only plot the results of one attribute in a histogram. Thus, we also evaluated these hit I/O costs. Throughout the analytical results these I/O costs are referred to *hit I/O 2*.

In Figure 6.4 we carried out the same set of queries on a bitmap index with 100 bins per indexed attribute. The main effect of the increased number of bins is a reduced candidate I/O overhead since each bin contains fewer candidate objects than with 10 bins (as we have seen in the previous figure). Thus, the pay-off for the index can already be seen for lower dimensional queries with higher attribute selectivities. In particular, the candidate I/O is below the sequential scan for three-dimensional queries up to 20% attribute selectivity. For five-dimensional queries the threshold moves up to 50% attribute selectivity, for ten dimensions up to 80% and for 25 dimensions up to 90%.

Considering also the hit I/O in Figure 6.4, the bitmap index is more I/O efficient than the sequential scan starting from five dimensions up to attribute selectivities of 20% for *hit I/O 1*

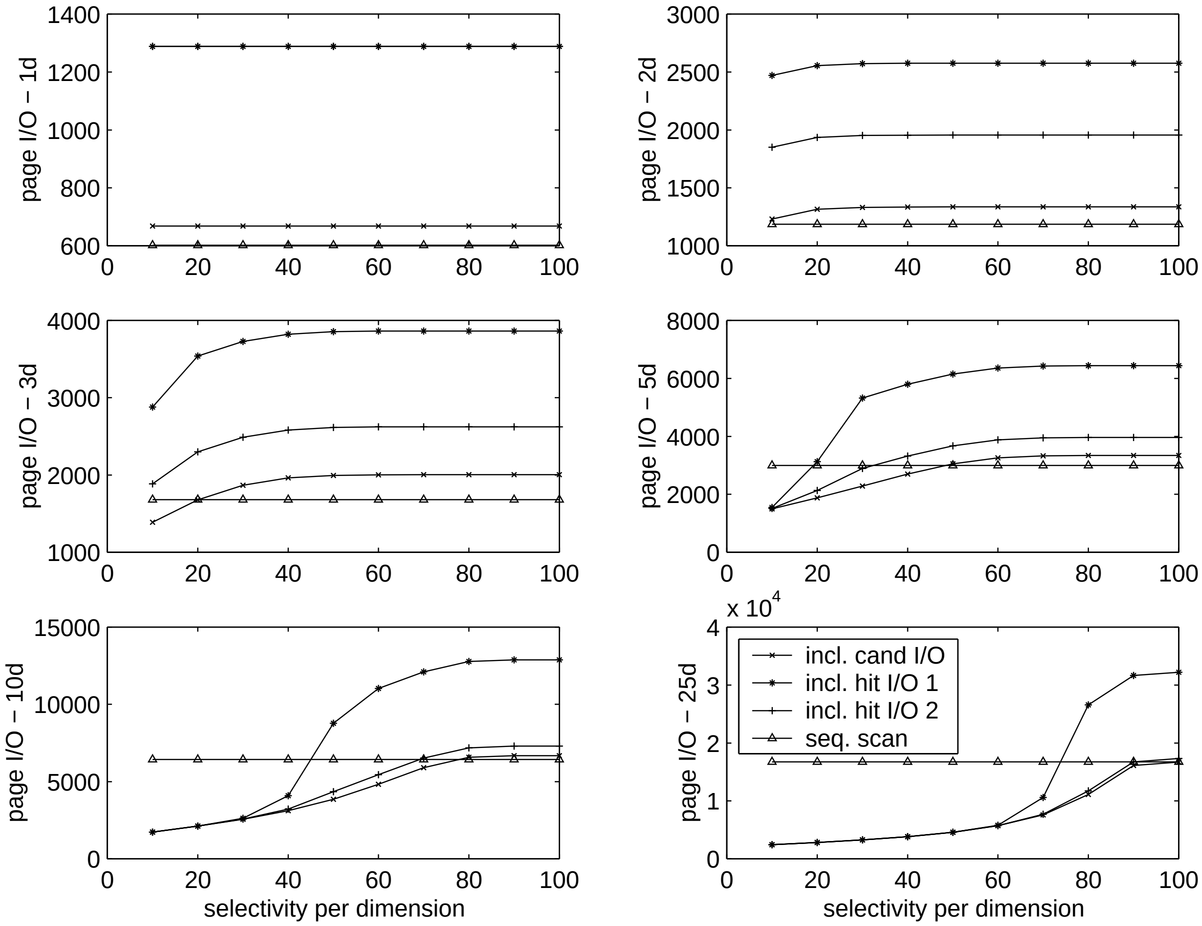

Figure 6.4: Analytical results: Page I/O for queries over multiple dimensions with various selectivities. 100 bins.

and up to 30% for *hit I/O 2*. For ten and 25-dimensional queries, the bitmap index becomes even more efficient.

In the following two figures (6.5 and 6.6) we show that by increasing the number of bins, the candidate I/O overhead can even be more decreased and thus the bitmap index shows even better performance characteristics. This is especially true for lower dimensional queries.

Figure 6.6 shows the I/O overhead for queries against a bitmap index with 1,000,000 bins per indexed attribute. This index size is the theoretical upper limit for 1,000,000 base objects. Since each object is represented by a bit slice, the overhead for the candidate I/O converges to zero and thus only the constant overhead for the index I/O is left.

We have thus demonstrated that the I/O overhead for the bitmap index based on discrete attribute values can be reduced by increasing the number of bins and thus the size of the index. However, reduced bitmap index overhead results in additional storage overhead.

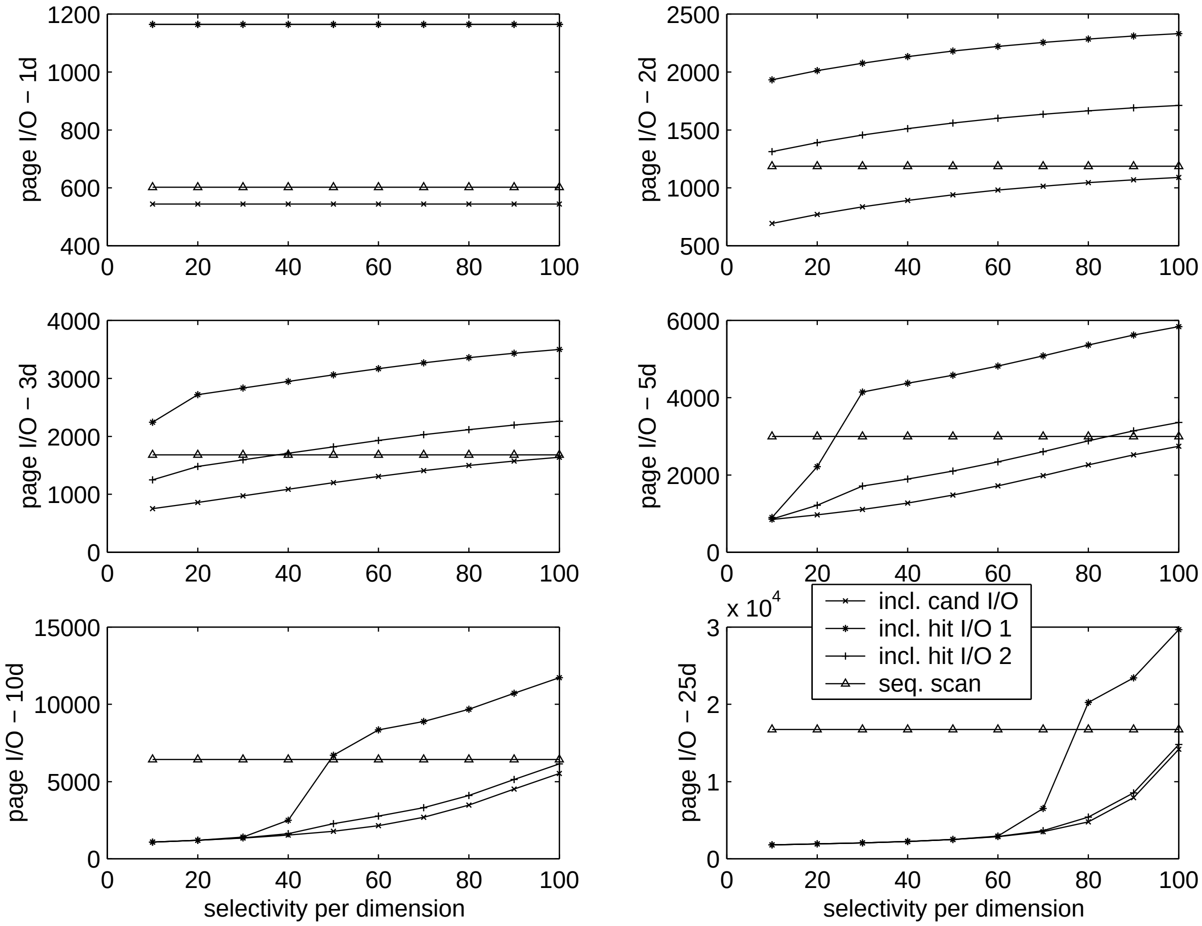

Figure 6.5: Analytical results: Page I/O for queries over multiple dimensions with various selectivities. 1,000 bins.

6.6 Experimental Results

In this section we evaluate the performance of the bitmap index experimentally and compare the results to our analytical study. One of the main questions which we will answer throughout the experiments is to find out up to which dimensionality under a given query selectivity the presented index data structure performs better than the sequential scan.

We carried out all our experiments on a Pentium II 400 under Linux Red Hat 6.1. The multi-dimensional bitmap index is implemented on top of Objectivity/DB version 5.1.2. All experiments operate on 10^6 objects with 1 to 25 attributes (dimensions) with uniformly distributed and independent data values. Table 6.6 shows the size of these objects and the query response time for sequentially reading them.

Similar to the previous section on analytical results, we carried out one-sided range queries over various dimensions starting from 1 to 25. The size of the index is 100 bins per attribute

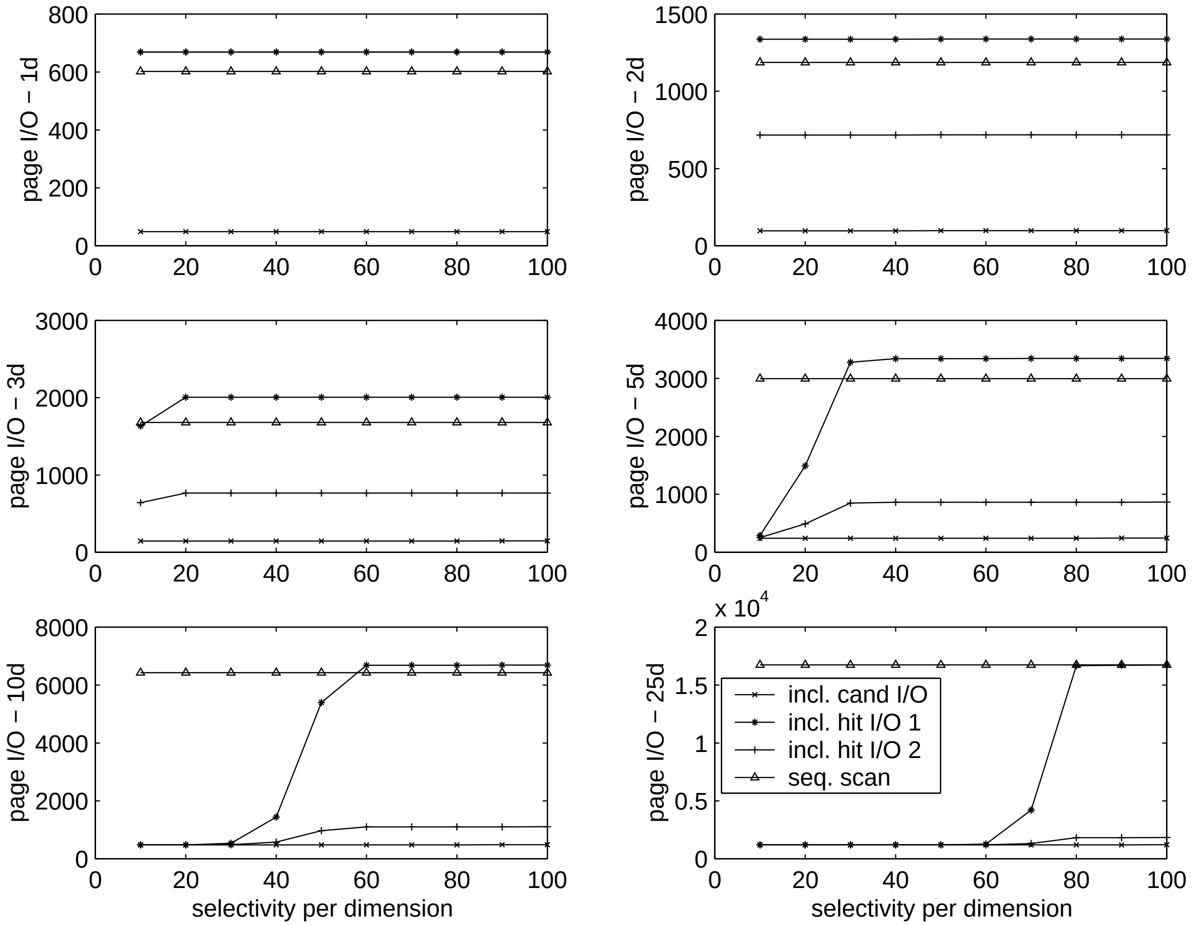

Figure 6.6: Analytical results: Page I/O for queries over multiple dimensions with various selectivities. 1,000,000 bins.

resulting in a total bitmap index size which is in the order of three larger than the base objects. For each dimension we varied the selectivity from 10% up to 100% in order to study the following phases:

- Index-I/O: Overhead for scanning the bit slices of the index.
- Candidate-I/O: Index-I/O including the I/O operations for checking the candidate objects against the query constraint.
- Hit-I/O 1 (non-interleaved): Candidate I/O including the I/O operations for retrieving the hits.
- Hit-I/O 2 (interleaved): Candidate I/O including the I/O operations for retrieving the hits. In this case, the hits are retrieved in smaller "segments" and thus the result of the first set of objects can be retrieved earlier.

Dimensions	data size [MB]	response time
1	4.9	8.7
2	9.7	11.4
5	24.5	22.9
10	52.7	42.5
15	80.1	62.4
25	137.1	103.3

Table 6.4: Response time for sequential scan over 10^6 objects with various number of attributes (dimensions) - attribute-wise clustering.

- Seq. scan: Comparison of the bitmap index with sequentially scanning the base objects.

Let us first analyse the case of a one-dimensional query against the bitmap index with 100 bins and compare it to the sequential scan which takes 8.7 seconds and requires 602 page I/Os (see Figure 6.7). The overhead for scanning the bit slices of the one-dimensional index is constant and requires 112 page I/Os or 1.3 seconds (index-I/O). Note that for the case of 100% selectivity both the page I/O and the response time go slightly down. This is because the last bin requires fewer bit slice operations for evaluating the query.

Next we plotted the candidate I/O which corresponds to the index-I/O and the additional I/O operations for checking the candidate objects against the query constraint. The costs for the page I/O are about 10% more than the costs for the sequential scan (sequential case). This is because with 100 bins the page selectivity for the candidate check is 100% which means that all attribute values must be fetched from disk in addition to the index-I/O. Consequently, also the response time for the query via the index is slightly higher than the sequential case.

The next two plots include the I/O-operations for fetching the hits from disk. In the case *hit-I/O 1* all objects are evaluated and then the hits are fetched from disk. As for *hit-I/O 2* a subset of the objects is evaluated and fetched from disk interleaved. In particular, the first 10,000 objects are evaluated and then the hits are fetched from disk. Afterwards, the next 10,000 objects are evaluated and then fetched from disk, etc. The advantage of *hit-I/O 2* is that fewer number of pages need to be read.

For two-dimensional queries (see Figure 6.7) all page I/O operations are above the sequential case. However, *hit-I/O 2* is more efficient than the sequential case up to an attribute selectivity of 40%, which corresponds to a total query selectivity of 16%.

Three-dimensional queries are more efficient than the sequential case up to an attribute selectivity of some 60% (total selectivity 21%). Note that the difference between *hit-I/O 1* and *hit-I/O 2* gets smaller since the total number of hits is reduced and thus the positive effect of interleaved hit I/O is less significant.

Starting from five-dimensional queries (see Figure 6.8) we can observe a further effect in the slope of the query response time, namely a “flat segment” of the curve for *hit-I/O 1* and *hit-I/O 2* for attribute selectivities between 20% and 50%. This is due to the overhead of the disk head movements for random access. In short, even though the page I/O decreases as the

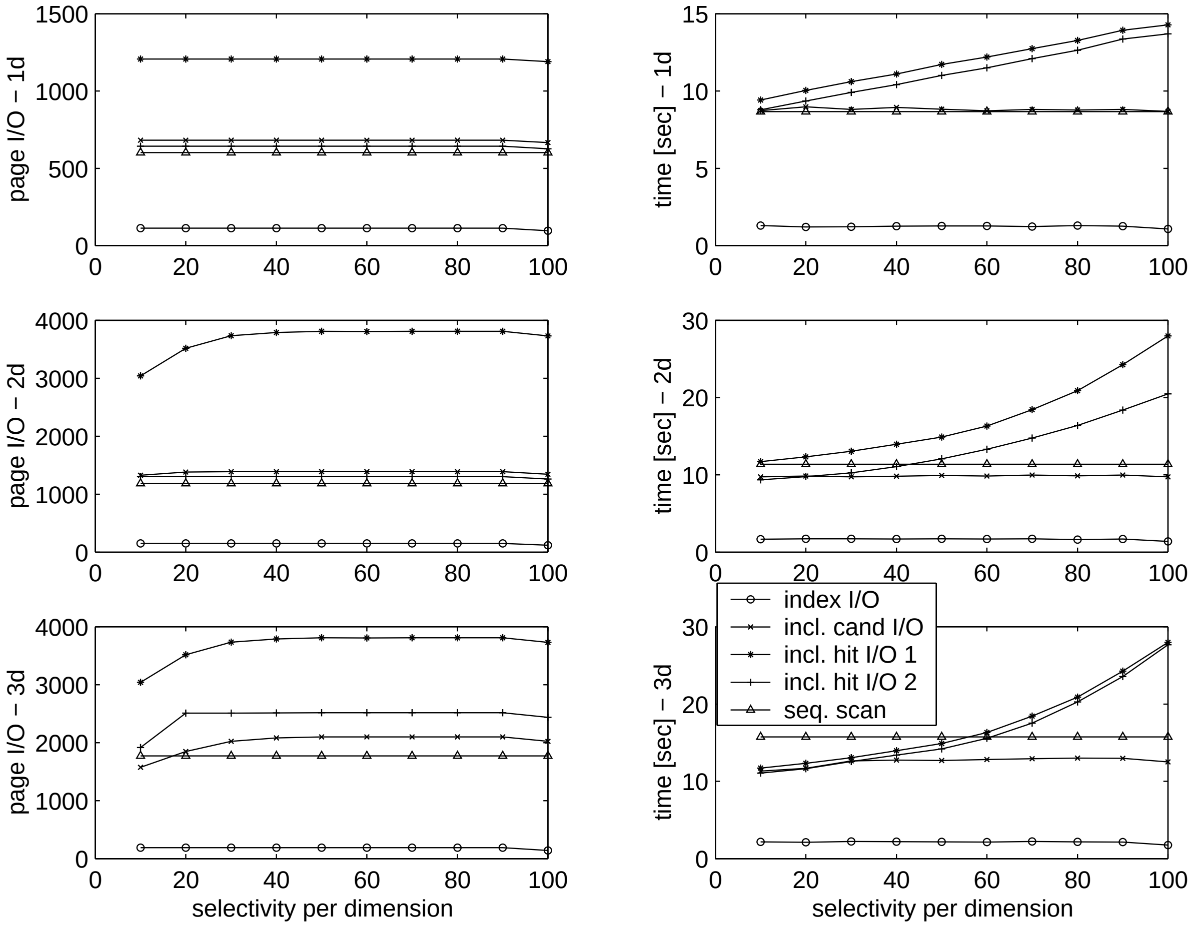

Figure 6.7: Experimental results: Page I/O and response for queries over multiple dimensions with various selectivities - one to three dimensions.

query selectivity decreases, the advantage of random access vs. sequential access does not pay off for any random access pattern.

With 25-dimensional queries the bitmap index including hit-I/O is more efficient than the sequential case up an attribute selectivity of 90% (total selectivity 0.9^{25}). It is important to note that for low selectivities, the response time of the hit-I/O is about twice as much as the response time for the index-I/O. For high selectivities, in particular for an attribute selectivity of 90%, the ratio increases up to a factor of 8.

6.7 Analytical vs. Experimental Results

The experimental results in Figures 6.7 and 6.8 show that the cost model can predict the I/O complexity for the bitmap index with an error of 10% to 20% for 25-dimensional queries. In

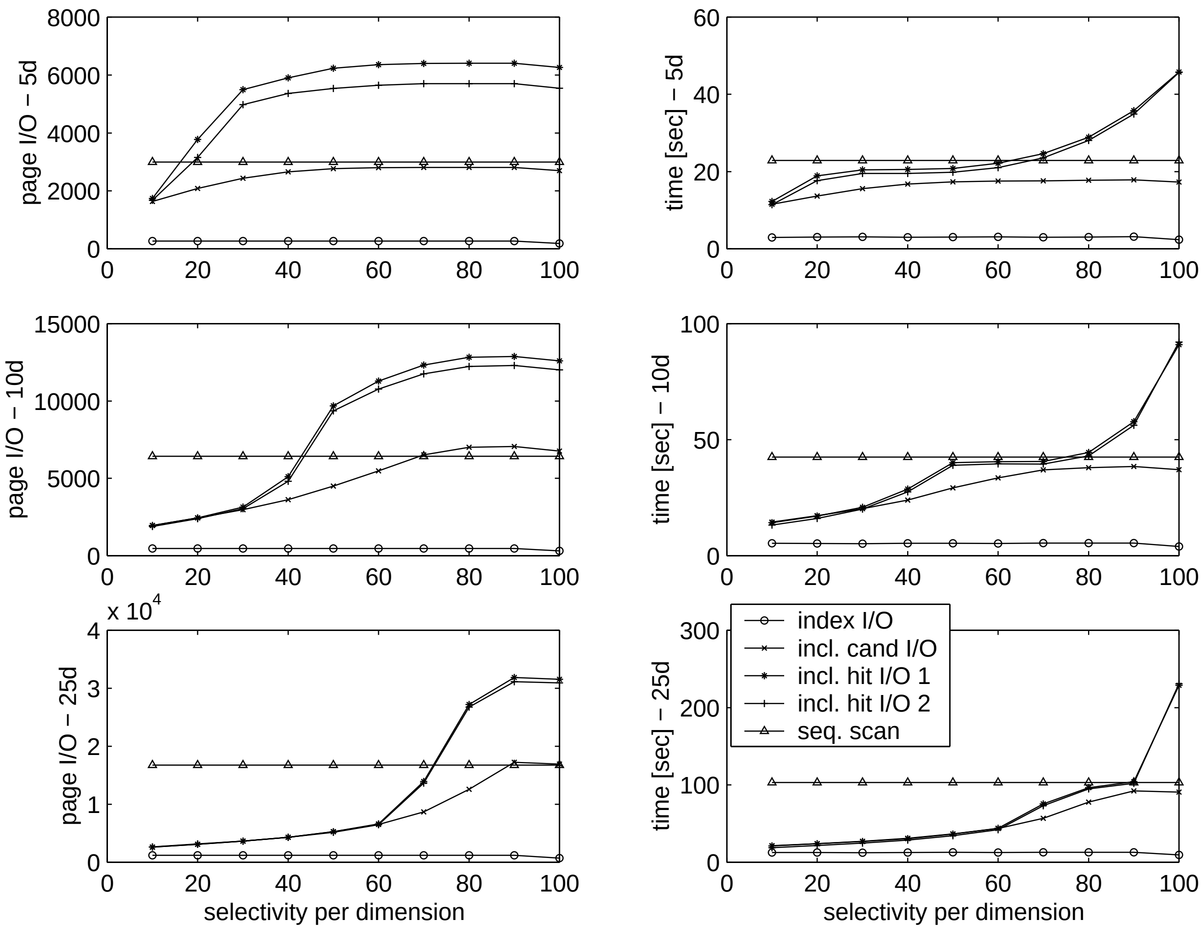

Figure 6.8: Experimental results: Page I/O and response for queries over multiple dimensions with various selectivities - five to 25 dimensions.

particular, the cost model sytematically underestimates the real costs between 10% to 20%. However, this accuracy is good enough for a query optimiser to decide between two access methods, namely whether to access all the data sequentially without using an index or to use the index.

6.8 Conclusions

We presented a novel algorithm for evaluating range queries called `GenericRangeEval` and studied analytically and experimentally the performance behaviour of multi-dimensional queries with various query selectivities. Initially the algorithm was designed to evaluate typical one-sided range queries but by a simple extension, the algorithm can also efficiently handle the more general case of two-sided range queries.

One of the most crucial design parameters is the number of bins of the index. In short, a larger number of bins results in a lower number of candidate objects which need to be checked against the query constraint. In other words, the larger the number of bins, the lower the response time of the bitmap index. On the other hand, a larger number of bins also increases the size of the bitmap index significantly. Thus, the choice of the number of bins is always a trade-off between speed and space.

Experimentally we evaluated the results yielded from the analytical cost model. We demonstrated that the cost model gives a fairly accurate estimation about the I/O complexity of the bitmap index. Thus, by calculating the I/O costs of the bitmap index prior to query execution, a query optimiser can easily decide whether to handle the query via the bitmap index or whether it is more efficient to sequentially scan over the base objects without using an index at all.

We based our experimental results on an index with 100 bins which is about three times the size of the base objects. We showed that for low dimensional queries the candidate check overhead of the bitmap index is the main bottleneck and thus the sequential scan is often faster. However, for high dimensional queries with low attribute selectivities (which is the main use case for HEP) the bitmap index shows a significant performance improvement over the sequential scan up to a factor of 5 for a 25-dimensional search space.

Chapter 7

Advanced Features

7.1 Introduction

In the previous chapters we analysed the impact of different kinds of bitmap encoding techniques on the query response time. We concluded that equality encoding is optimised for *exact match queries* whereas range encoding is optimised for *range queries*. In this chapter we will discuss further optimisation techniques for tuning the performance of the bitmap index, such as an adaptive index, different binning strategies and bitmap compression. In particular, we will analyse the impact of bitmap compression on uniformly and non-uniformly distributed data.

7.2 Adaptive Index

One of the simplest and most efficient adaptive techniques of the index based on `GenericRangeEncoding` is to evaluate first the attribute (dimension) which has the lowest attribute selectivity. The advantage of this approach is that the search space gets pruned already very early so that the result set for the evaluation of higher dimensions is reduced. Consider this simple 3-dimensional query based on a search space in the range of [0;100] with uniformly distributed data values. A possible query execution plan is as follows:

$a_0 < 40$ AND $a_1 > 95$ AND $a_2 < 47$

The attribute selectivity of a_0 is 40%, of a_1 5% and of a_2 47%. Thus, the query can be more efficiently evaluated by ordering the attributes (dimension) according to increasing attribute selectivities. Assuming left-to-right evaluation, the optimal query execution plan is as follows:

$a_1 > 95$ AND $a_0 < 40$ AND $a_2 < 47$

7.3 Binning Strategies

A further important turning parameter of the bitmap index is the binning. Up to now, we only assumed so-called *equi-width* bins, which means that each bin has the same width and thus the distribution of the data values is also reflected in the number of entries for each bin. In Chapter 5 we briefly discussed this kind of binning strategy. Besides equi-width binning, there is also the possibility of *equi-depth* binning which guarantees that each bin has the same number of entries. Both binning strategies have advantages and disadvantages which we will discuss in this section.

In short, the binning strategy depends on two factors, namely the *data distribution* and the *query access patterns* (or the query distribution). Equi-depth binning guarantees nearly constant access time for all kind of queries independent of the data distribution. One would chose this kind of binning when no query access patterns are available.

Since equi-width bins reflect the data distribution, this kind of binning is preferable if the query access patterns are such that those bins are queried most, which have the least number of entries.

If the query access patterns are well understood, the binning could be even more adapted. Thus, for heavily queried regions in the search space, the bin ranges should be narrow such that these bins only have a small number of entries. On possible solution to this problem is given in [40].

7.4 Bitmap Compression on Uniformly Distributed Data

In Chapter 6 we discussed the typical *space/time trade-off* when designing a bitmap index. In short, the larger the number of bins, the lower the query response time of the bitmap index. On the other hand, a larger number of bins also increases the size of the bitmap index significantly. One way to reduce the space complexity is to compress the index. However, the requirement for a good bitmap compression algorithm is not only to have a good compression ratio but also to show good performance during the logical operations between the bit slices. In the following sub sections we discuss the impact of compression on equality encoded and range encoded bitmap indices. According to [15] equality-encoding shows good compressibility due to the sparse bit slices whereas range-encoding shows better performance on a verbatim bitmap index. We will extend this study and compare the relative performance for all combinations thereof, i.e. equality encoding vs. range encoding on compressed and verbatim bitmap indices, in order to classify the best encoding technique for different kinds of query types.

Together with Theodore Johnson from AT&T Labs-Research we implemented a slightly improved version of the original Byte-Aligned Bitmap Compression [39] algorithm by [1]. The algorithm is briefly described in Chapter 3.

7.4.1 Equality Encoded Bitmap Index

We again base our experiments on 1,000,000 objects with uniformly distributed data values. First we compare the size of a bitmap index of 100 bins for verbatim bit slices and for compressed ones. The size of the verbatim bitmap index for one attribute is some 13.5 MB whereas the size of the compressed bitmap index is some 2 MB. The compression factor of each bit slice is roughly 9. However, the compression factor of the whole bitmap index is only around 7 due to some constant overhead of an Objectivity database (which is about 0.5 MB). For 1000 bins the size of the verbatim bitmap index is 140 MB and of the compressed index 8.5 MB. This corresponds to a compression factor of 16.5 for the whole bitmap index.

For the following tests, we studied the I/O costs and the response time of a 1-dimensional query based on a compressed equality encoded bitmap with 100 and 1000 bins. Since we want to evaluate the compression algorithm, we only report on the results of evaluating the query via the bitmap index and ignore the candidate I/O at this stage.

With 100 bins, the query response time of the compressed bitmap is lower than the verbatim bitmaps for an attribute selectivity above 20% (see Figure 7.1). Even though the number of page I/Os is always lower than for the verbatim case, there is additional CPU overhead of the compressed index which results in a worse performance for queries with selectivities below 20%.

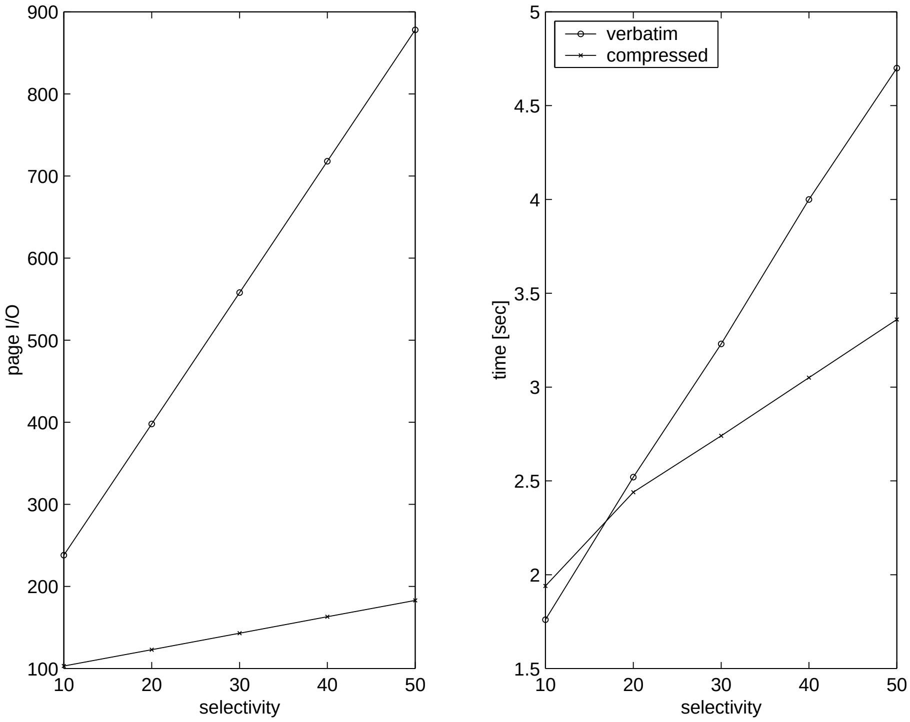

Figure 7.1: Query response time of verbatim vs. compressed equality encoded bitmap index - 100 bins.

With 1000 bins the advantage of the compressed bitmap index over the verbatim one becomes very significant (see Figure 7.2). For a query selectivity of 10% the query response time of the compressed bitmap index is a factor of two lower than for the verbatim one. For a query selectivity of 50% the compressed bitmap index is even a factor of 3 faster than the verbatim one.

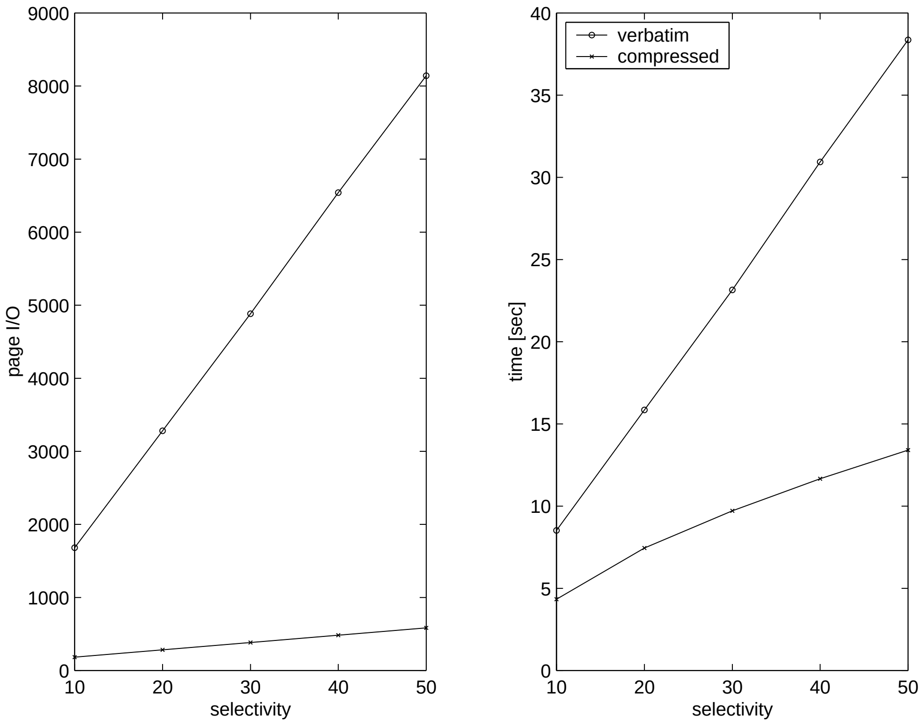

Figure 7.2: Query response time of verbatim vs. compressed equality encoded bitmap index - 1000 bins.

These results clearly demonstrate the advantage of compressed over verbatim equality encoded bitmap indices.

7.4.2 Range Encoded Bitmap Index

As already pointed out by [15] the compression ratio for range encoded bitmap indices is relatively poor in comparison to the sparser representation of equality encoded bitmaps. We evaluated the compression ratio for a bitmap index of 100 bins on 1,000,000 objects with uniformly distributed data values.

In Figure 7.3 we plotted the compression ratio of each single bin of the bitmap index. As we can see, the compression ratio is only below 0.5 for 7% of the bins. All remaining bins are hardly compressible (until about 15% of the bins) or not compressible any more - the latter is true for 85% of the bins.

For 1,000 bins the compression ratio shows a similar pattern. To sum up, bitmap compression for range encoded bitmap indices with uniformly distributed attribute values is practically not feasible.

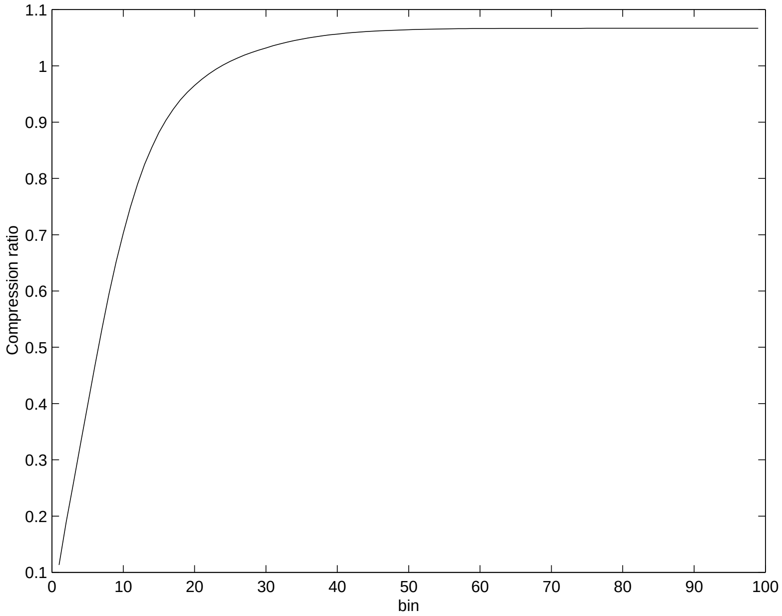

Figure 7.3: Compression ratio of range encoded bitmap index with 100 bins.

7.4.3 Compressed Equality Encoding vs. Verbatim Range Encoding

As we have demonstrated in the previous sections, equality encoded bitmap indices show good compressibility and also a better performance than verbatim equality encoded bitmap indices. As for range encoding, verbatim bitmap indices show better performance characteristics due to the relatively poor compressibility of the bitmaps. Thus, in this section we compare the two "winners" of our evaluation, namely compressed equality encoded vs. verbatim range encoded bitmap indices and show up to which selectivity one technique shows better performance than the other one.

Let us now evaluate the impact of bitmap compression for multi-dimensional queries. We assume an equality encoded bitmap index with 1000 compressed bit slices and compare it to a range encoded bitmap index with 100 bins. Due to the good compressibility of the equality encoded index, the size of the compressed bitmap index with 1000 bins is about 60% of the size of the uncompressed bitmap index with 100 bins.

Our main focus of interest is a direct comparison of the compressed equality encoded bitmap index with a verbatim range encoded one. We thus perform similar benchmarks as depicted in Figures 6.7 and 6.8 of the previous chapter. In particular, we will study the number of page I/Os and the response time for queries over multiple dimensions with various selectivities. Since the compression has only an impact on the index-I/O and indirectly on the candidate-I/O, we omit the response times including the hit-I/O and only report on results including candidate I/O.

As we can see in Figure 7.4 for 1-dimensional queries, the compressed bitmap index performs nearly uniformly well as the verbatim one for query selectivities of 10%. For higher

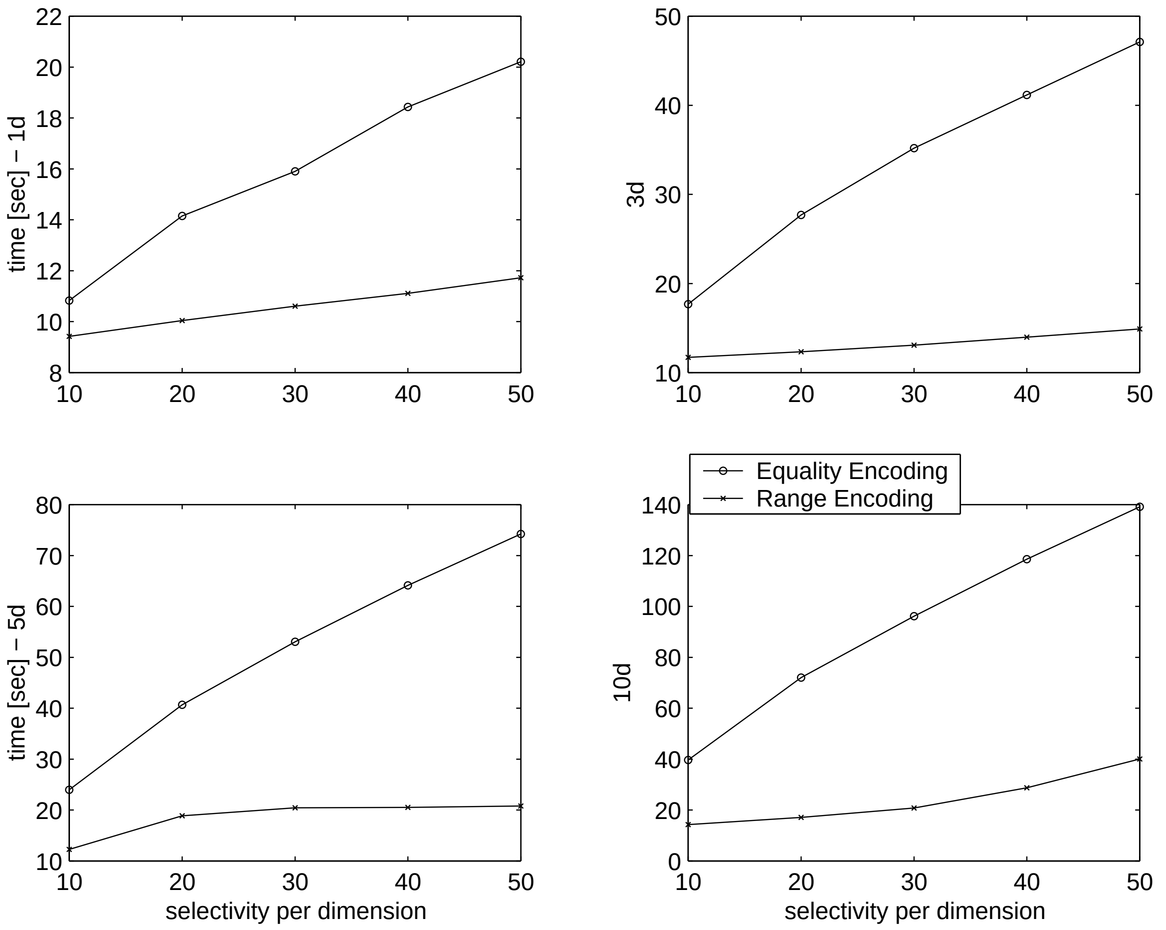

Figure 7.4: Query response time for compressed equality encoding vs. verbatim range encoding.

selectivities the overhead for reading more bit slices outweighs the advantage of bitmap compression. Also for multi-dimensional queries the verbatim range encoded bitmap index shows better performance characteristics than the compressed equality encoded bitmap index.

Since the compressed bitmap index is only more efficient than the verbatim one for certain attribute selectivities mostly up to 10%, we study in more detail the behaviour of the compressed bitmap index for attribute selectivities between 1% and 10%.

For these tests, we again consider only the case where the candidate-I/O is included (see Figure 7.5). For 1-dimensional queries the compressed bitmap index shows better performance than the uncompressed one up to an attribute query selectivity of 5%. For 2 to 4 dimensions the threshold drops to 3%, and for 5 and 10 dimension even to 2%. Thus, for the majority of the cases the verbatim range encoded bitmap index outperforms the compressed equality encoded bitmap index.

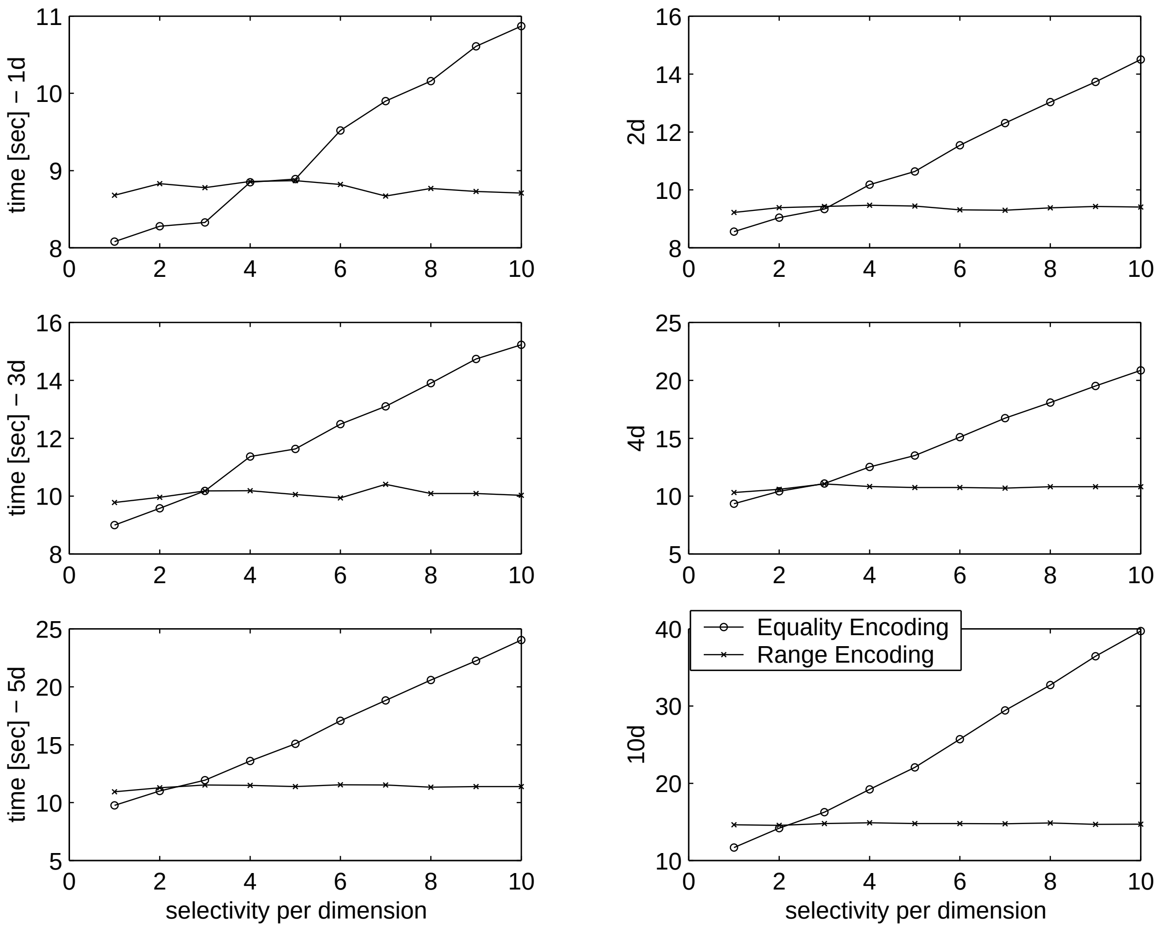

Figure 7.5: Query response time for compressed equality encoding vs. verbatim range encoding.

7.5 Bitmap Compression on Non-Uniformly Distributed Data

Up to now we only discussed bitmap compression for uniformly distributed data and we showed that range encoded bitmap indices do not compress very well with byte-aligned bitmap compression. In this section we analyse the performance of compressed bitmap indices based on data following a typical exponential distribution.

Our first set of benchmarks operates on 1,000,000 objects with 25 attributes. Each range encoded bitmap index consists of 100 equi-width bins per indexed attribute. Figure 7.6 a) depicts a typical exponential distribution of the form e^x.

Due to the cumulative nature of range encoded bitmap indices, the number of set bits increases for each bit slice (see Figure 7.6 b)). The number of candidate objects is given as the difference between two adjacent bit slices. For example, if bit slice B_i has 1,000 bits set to 1 and bit slice B_{i+1} has 1,200 bits set to 1, then the number of candidate objects for bit slice B_{i+1} is 200.

In Figure 7.6 c) we can see the compression ratio for each bin using two-sided byte-aligned bitmap compression (BBC2). We can observe that especially bins on the right side, i.e. bins with a high ordinal number, show good compressibility whereas bins with a low ordinal number show poor compressibility.

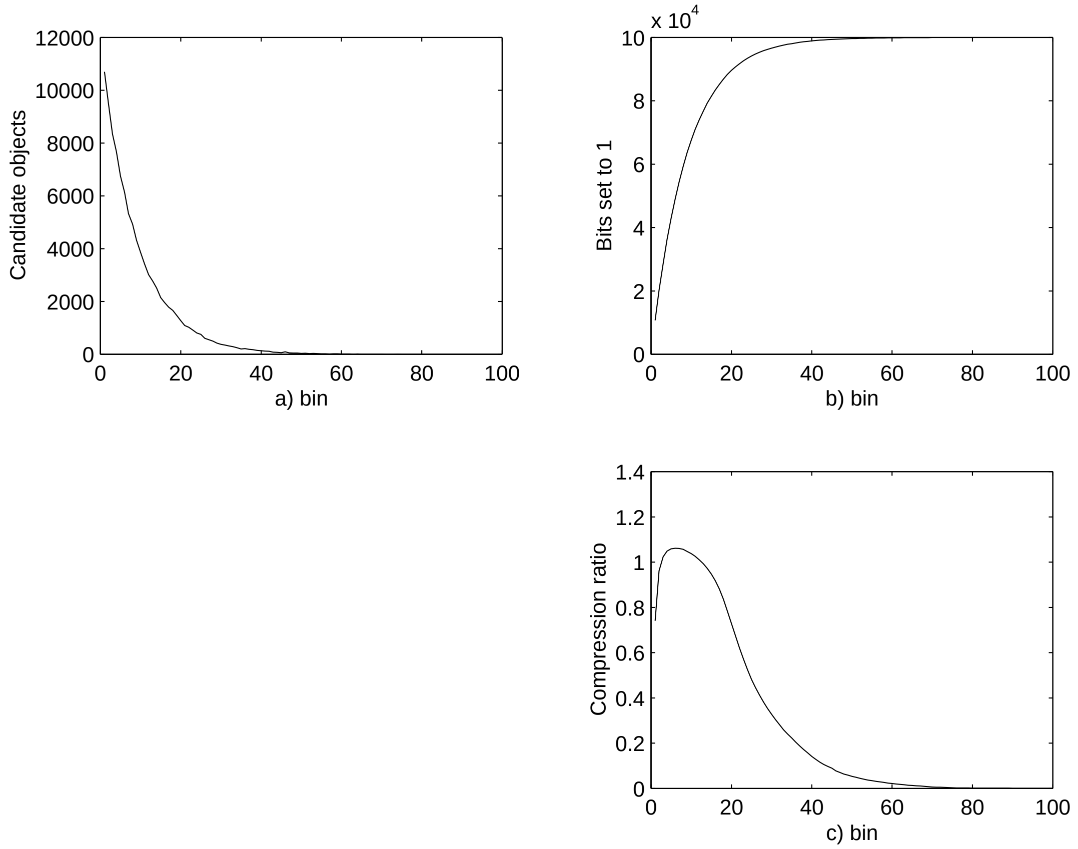

Figure 7.6: Range encoded bitmap index based on data following an exponential distribution.

In Figure 7.7 a) and c) we plotted the query response time for the index I/O, i.e. not including the I/O operations for the candidate check. The graphs show the response times for 10-dimensional and 25-dimensional queries with various query selectivities. We will first analyse one-sided range queries including only the "<"-operator, for example $a_1 < 5$.

Since with non-uniformly distributed data, not all bins contain the same number of candidate objects, we introduce a new term called *effected bin* which corresponds to the bit slice holding the candidate objects. Consider, for example, an exponential distribution with values in the range of [0;10). When we further assume 100 equi-width bins and a query $a_1 < 5$ then the effected bin, i.e. the bin which holds the candidate objects, is bin 50.

We can see that for this data distribution and this CPU-I/O configuration, the compressed bitmap index (BBC2) performs better than the verbatim one if the effected bin is larger than 30. When referring to the compression ratio in Figure 7.6 c) we can observe that the compressed

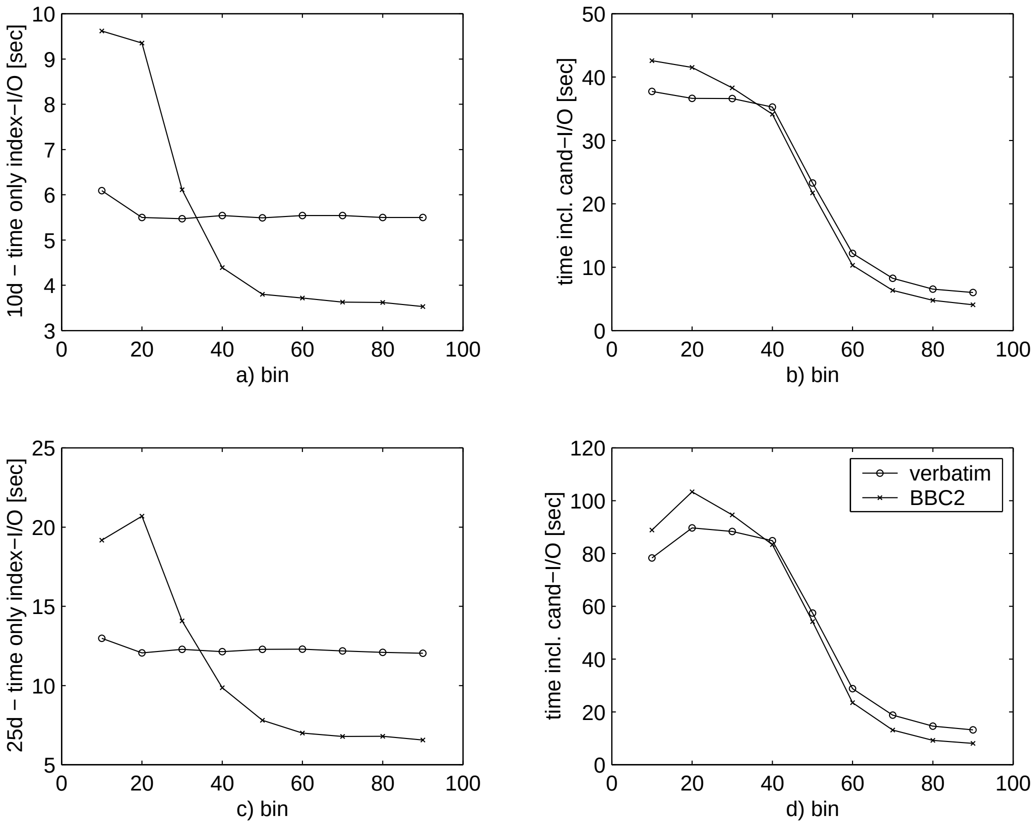

Figure 7.7: Response time for verbatim vs. compressed bitmap indices. Queries include the "<"-operator.

bitmap performs better for compression ratios below 0.1. In short, in the worst case the compressed bitmap index is about a factor of two slower than the verbatim one. However, in the best case, the compressed bitmap index is a factor of two faster than the verbatim bitmap index.

The query response time including I/O operations for the candidate check are shown in Figures 7.7 c) and d). Here we can observe the impact of bitmap compression on the total query response time.

We will now analyse the impact of bitmap compression on one-sided range queries including the ">"-operator. In Figure 7.8 we can see that in the best case, the query response time for the compressed bitmap index is slightly less than a factor of two faster than for the verbatim bitmap. Due to the lower attribute selectivity of these queries, the response time including the candidate check is lower than in Figures 7.7 b) and d).

In Figure 7.9 we directly compare one-sided range queries including the "<"-operator with one-sides queries including the ">"-operator. In these plots only the index I/O is included. The results show that in the best case queries including the "<" operator perform better

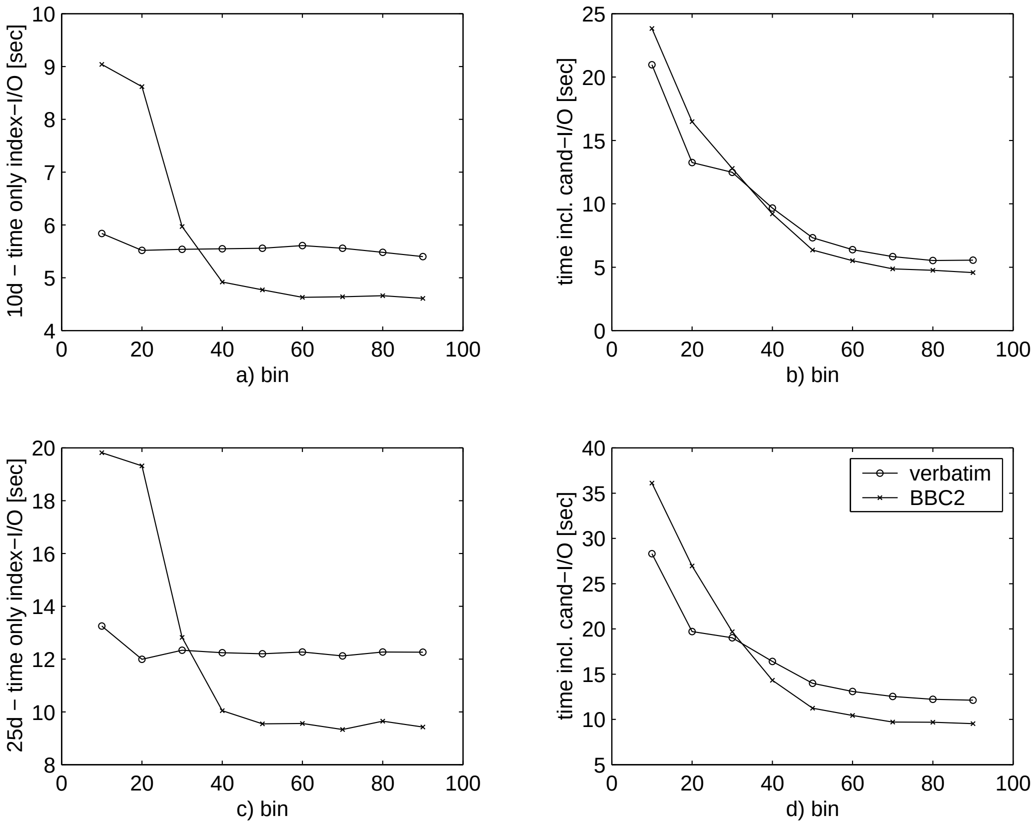

Figure 7.8: Response time for verbatim vs. compressed bitmap indices. Queries include the ">"-operator.

than queries including the ">"-operator. However, in the worst case, queries including the ">" perform slightly better. The worse performance of queries including the ">"-operator can partially be attributed to the additional NOT-operator in line 1.11 of the algorithm introduced in Chapter 6.3. In addition, further logical AND-operations between the resulting negated bit slice and a bit slice with many bits set to 1 (as is the case for bins with high ordinal numbers) show slightly worse performance characteristics.

In the next set of benchmarks we built an index with 400 compressed equi-width bins. The size of this index corresponds roughly to the size of a verbatim bitmap index with 100 equi-width bins. We ran the same set of queries including the "<"-operator and compared the query response time to a compressed bitmap with 100 equi-width bins. The results in Figure 7.10 show that the response time for the index I/O is slightly better for the case with 400 bins. However, the response time including the candidate check gets even further improved when compared to the case with 100 bins.

We finally executed the queries shown in Figures 7.7 and 7.8 on "tier2" which is a machine with a more powerful I/O subsystem. We can observe in Figure 7.11 that the impact of

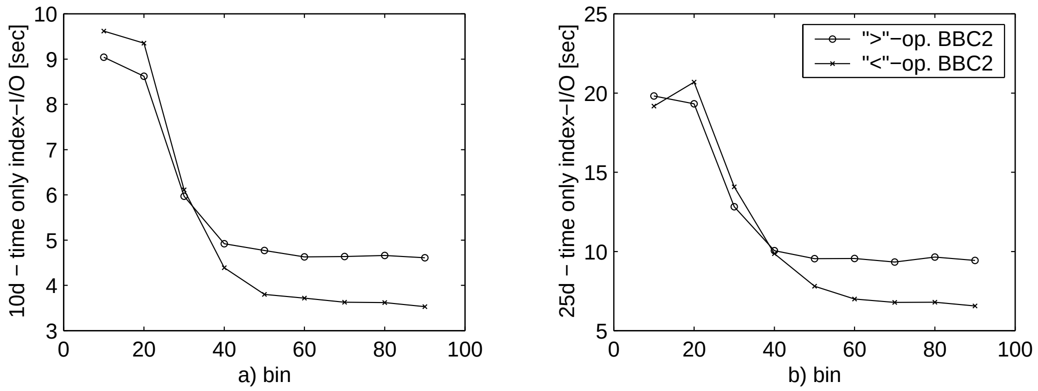

Figure 7.9: Response time for compressed bitmap indices. Queries including “$<$”-operator vs. queries including “$>$”-operator.

bitmap compression yields only a slight performance improvement for queries including the “$<$” operator. However, as we expect that in the future the CPU speed will increase faster than the I/O rate, the performance improvement due to bitmap compression will be more significant.

7.6 Conclusions

A simple optimisation technique for evaluating multi-dimensional queries via bitmap indices is to order the attributes according to their selectivities. By processing attributes with lower selectivities first, the search space gets pruned very early and thus the result set for the remaining dimensions is smaller. As a consequence, the overhead for the candidate check gets smaller with each additional dimension.

We also discussed different binning strategies when the query access patterns are known. By keeping those bins small, which keep the most frequently accessed attribute ranges, a significant performance improvement can be yielded. This is true because small bins have a lower number of candidate objects and thus the overhead for the candidate check gets also reduced.

We evaluated the impact of compression on the performance of equality encoded and range encoded bitmap indices based on uniformly distributed data. In short, equality encoded bitmap indices show good compression ratios due to the sparsity of the bit slices. Range encoding, on the other hand, shows only good compression ratios for a small percentage of the whole bitmap index. Thus, compression of range encoded bitmap indices based on uniformly distributed data does not give any additional performance improvement.

Apart from the compression ratio, we compared the query response time of verbatim and compressed equality encoded bitmap indices and demonstrated that the compressed version outperforms the verbatim one in most cases.

Next we compared the performance of compressed equality encoded with verbatim range

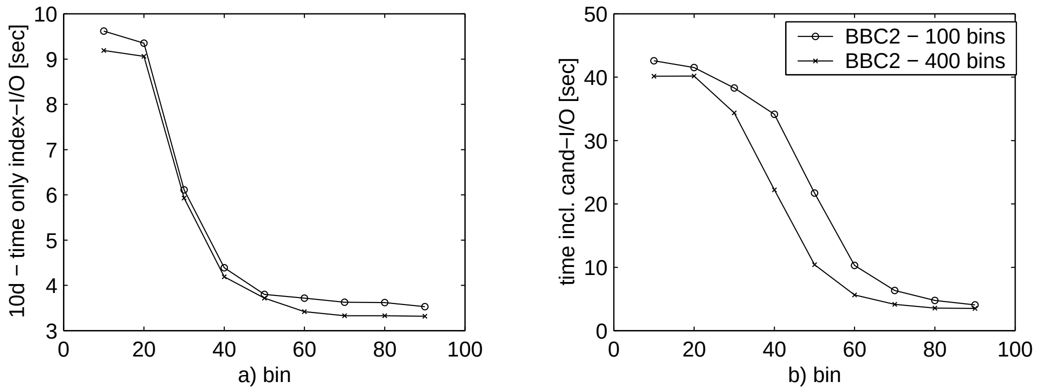

Figure 7.10: Response time for queries based on compressed bitmap index with 100 bins vs. 400 bins. Queries include the "<"-operator.

encoded bitmap indices. The results clearly show that verbatim range encoding always outperforms compressed equality encoding for attribute query selectivities above 10% (remember the total query selectivity is certainly lower). This is due to the fact that with range encoding based on the novel algorithm presented in Chapter 6, in the worst case only 3 bit slices need to be scanned for each dimension. As for equality encoding, in the worst case, half of the bit slices need to be scanned.

Compressed equality encoding outperforms verbatim range encoding for low attribute query selectivities between 1% and 5%. For attribute selectivities between 5% and 10% compressed equality encoding partially shows better performance characteristics than verbatim range encoding. In short, for attribute query selectivities below 5%, compressed equality encoding is the optimised bitmap index. For attribute query selectivities above 5% the verbatim range encoded bitmap index is the "overall winner".

We finally analysed the impact of bitmap compression on range encoded bitmap indices based on non-uniformly distributed data. The results show that the query response time for certain query types can be improved with bitmap compression. However, the factor of the improvement highly depends on the ratio of CPU versus I/O speed. Since the CPU speed is expected to increase faster than the I/O rate, the performance improvements gained with bitmap compression will be even more significant in the future.

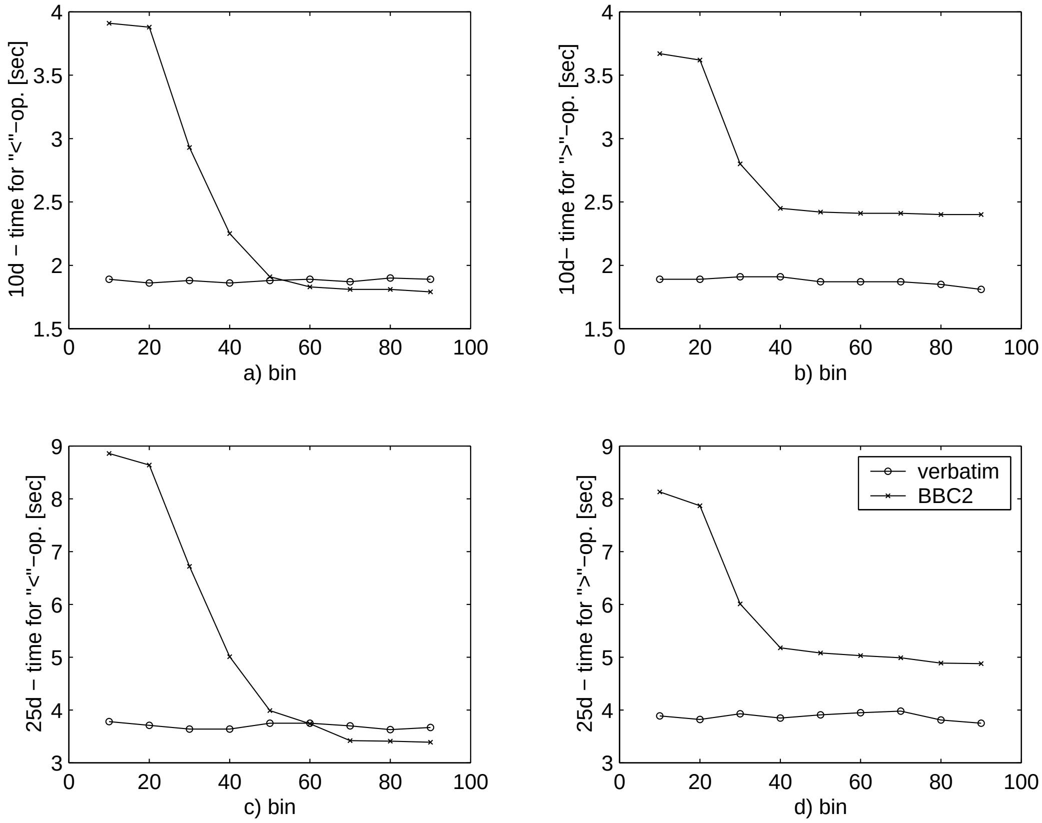

Figure 7.11: Response time for verbatim vs. compressed bitmap indices on “tier2”.

Chapter 8

Applications

8.1 Introduction

On evaluating analytically and experimentally the behaviour of bitmap indices against synthetic data, we will now apply bitmap indices for queries against real data. In particular, we study two different application domains, namely High Energy Physics and Astronomy. Both of these application areas are characterised by large amounts of read-mostly data with high-dimensional search spaces.

The main aim of this chapter is to demonstrate that bitmap indices can significantly improve the performance of certain analysis types of real world applications. We also discuss a possible extension of the analytical cost model presented in Chapter 6 for estimating the performance of multi-dimensional queries against real data with various distributions. Finally we evaluate the performance impact of compressed bitmap indices against real data.

8.2 Extended Cost Model

The cost model we presented in Chapter 6 was used for predicting the performance of range encoded bitmap indices based on uniformly distributed and independent data values. However, in order to apply it also for real data, two new factors need to be considered, namely

- *clustering* of the bits within the bit slices
- *correlation* of the attribute values (dimensions).

The *clustering* of the bits within the bit slices has a direct effect on the expected number of pages to be accessed for each candidate check. The *correlation*, on the other hand, has an impact on the reduction of candidate objects for higher dimensions. In this section, we will discuss both effects in more detail.

8.2.1 Clustering

As for random uniformly distributed data values, the candidate bits within one bit slice show hardly any clustering effect. However, as for non-randomly distributed data values, clusters of different lengths can be observed. In other words, attributes with similar values are highly clustered together on a single page. As a consequence, the expected number of page accesses which is calculated from the number of candidate objects according to Equation 6.5 will be an upper bound of the number of page accesses during the candidate check.

8.2.2 Correlation

In order to explain the effects of correlation on our cost model, let us shortly revise the example in Chapter 6 where we calculated the expected number of candidate objects for the following 4-dimensional query:

$a_0 < 30$ AND $a_1 > 85$ AND $a_2 < 20$ AND $a_3 > 95$

We assumed uniformly distributed data values in the range of [0;100] and 100 bins. We calculated the expected number of candidate objects for the first attribute according to Equation 5.6 which is $E_{c0} = 10{,}000$. For the second attribute, we calculated the number of expected candidate objects based on the attribute selectivity of a_0, i.e. $E_{c1} = 10{,}000 \cdot 0.3 = 3{,}000$. For attribute a_2 and a_3 the expected number of candidate objects are $E_{c2} = 3{,}000 \cdot 0.15 = 450$ and $E_{c3} = 450 \cdot 0.2 = 90$ respectively.

However, for highly correlated data values, this positive effect of candidate reduction does not hold. Thus, the correlation factor for the adjacent attributes in the query plan must be considered in our calculations. For example, if the correlation between attribute a_0 and a_1 is 1, then the expected number of candidates E_{c1} is 10,000 rather than $10{,}000 \cdot 0.3 = 3{,}000$. In general, the effect of correlation of attribute a_i and a_{i-1} on the number of expected candidates E_c is given as:

$$E_c = E_{c_i}(1 - (1 - sel_{a_{i-1}})(1 - corr_{a_i,a_{i-1}})) \tag{8.1}$$

where $sel_{a_{i-1}}$ is the selectivity of attribute a_{i-1} and $corr_{a_i,a_{i-1}}$ is the correlation factor between the attributes a_i and a_{i-1}. For example, if the correlation factor between the attributes a_0 and a_1 is 0.4, then the expected number of candidate objects E_c for the candidate check is 10,000 - 10,000 $\cdot$ (1-0.3) $\cdot$ (1- 0.4) = 5,800.

8.3 Query Parser

The main functionality of the query parser is two-fold. On the one hand, a query is specified in terms of C++-syntax and the parser is able to distinguish between indexed and non-indexed attributes and delivers them accordingly to the bitmap index. On the other hand, the parser is also be able to handle any kind of complex mathematical expressions on attribute values which cannot be directly indexed on. These expressions are evaluated by the bitmap index after all indexed attributes are processed. We implemented this kind of query processor together with Koen Holtman from Caltech.

Typical queries can be as follows:

```
Q1: Bitmaps ‘‘jet1E < 3.7 && jet2Phi > 0.3’’
```

```
Q2: Bitmaps ‘‘jet1E < 3.7 && sin(jet2Phi) > 0.3 && jet2E > 5.5’’
```

Query `Q1` is a typical two-dimensional range query with two indexed attributes. Query `Q2` is a three-dimensional query where the second dimension `sin(jet2Phi) > 0.3` is a mathematical expression which gets compiled and is evaluated after the two indexed attributes are processed.

8.4 High Energy Physics

In the following sections we will evaluate the bitmap index on real physics data taken from the CMS [16] experiment. In particular, we will analyse the behaviour of the index based on data values with different distributions and compare the behaviour to the conventional access method currently used in this experiment, namely the sequential scan.

The physics data (tag data) consists of 1,401,020 tags with 37 attributes each. The size of this data is 262 MB. The query response times for sequentially scanning 1 to 10 attributes (dimensions) are given in the following table. All the tests in this chapter are carried out at Caltech's "tier2" machine (Dual 933 MHz Pentium III Linux server, 900 MB RAM, using a 600 GB 3ware RAID 0 array). Currently all the data fits in main memory but in a few years the tag data of the experiments will be up to 1 TB and thus will most likely not fit into main memory any more.

Dimensions	data size [MB]	response time
1	6.9	3.6
2	13.7	5.5
3	20.4	8.0
4	27.1	10.1
5	33.9	12.3
10	67.6	23.2

Table 8.1: Response time for sequential scan over 1,401,020 objects with various number of attributes (dimensions) - attribute-wise clustering.

8.4.1 Data Distribution

Similar to the experiments carried out in the previous chapters, we created a bitmap index for each attribute consisting of 100 equi-width bins. The size of the whole bitmap index for all attributes is 790 MB. Out of these 37 attributes we have chosen randomly 10 attributes which we study more carefully. The distributions of some of these attribute values are reflected by the distribution of the bins (see Figure 8.1 and 8.2). In Figure 8.1 a) and c) we plotted the number of set bits in each of the range encoded bit slices of the queried attributes and also the number of candidate objects for each bit slice (Figure 8.1 b) and d)).

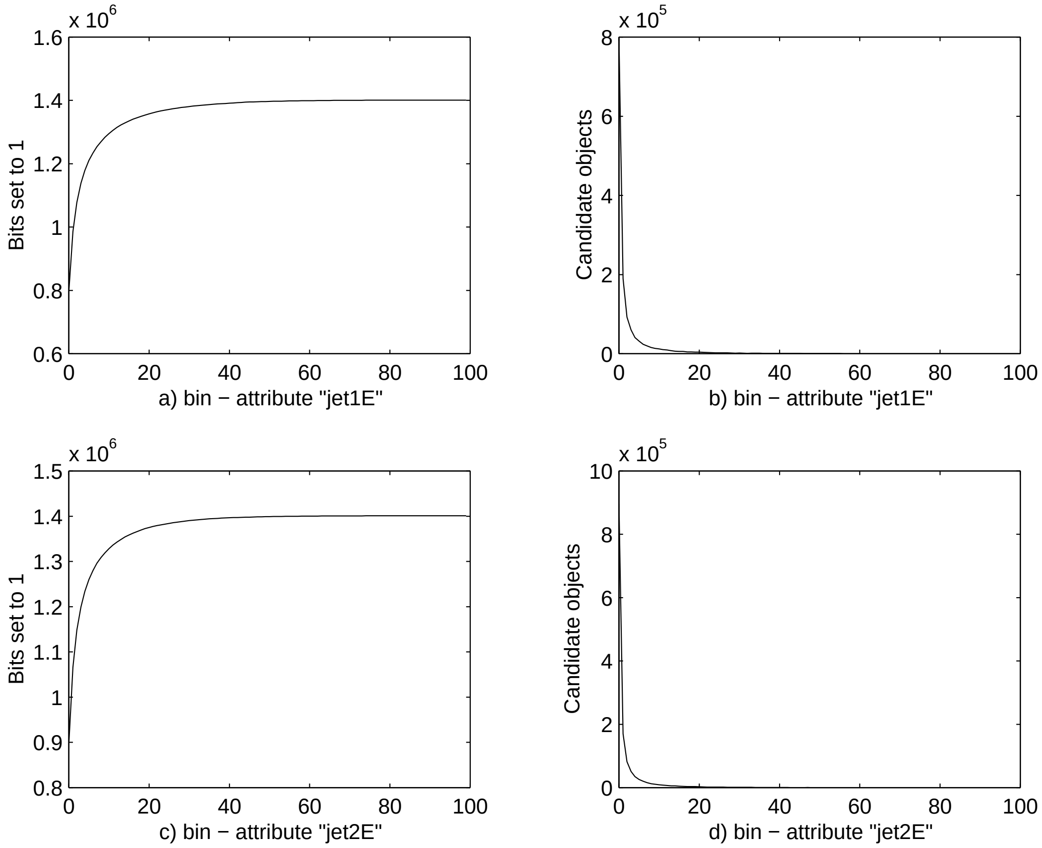

Figure 8.1: Effect of equi-width binning for the attributes `jet1E` and `jet2E`.

We see that most of the data values follow a typical exponential distribution. This is mainly true for physics quantities that measure energies such as $jet1E$ or $jet2E$ (see Figure 8.1). We can also observe that these values are highly correlated. Other values like $jet1Phi$ show a quite random distribution (see Figure 8.2).

8.4.2 Sample Queries

In order to study the efficiency of the bitmap index, we have chosen randomly a set of queries over one to ten dimensions. The queries are given below:

```
Q1: Bitmaps ‘‘jet1E > 0’’

Q2: Bitmaps ‘‘jet1E > 1000’’

Q3: Bitmaps ‘‘jet1E > 1000 && jet2E > 1000’’

Q4: Bitmaps ‘‘jet1E > 1000 && jet2E > 1000 && part1E > 1000’’
```

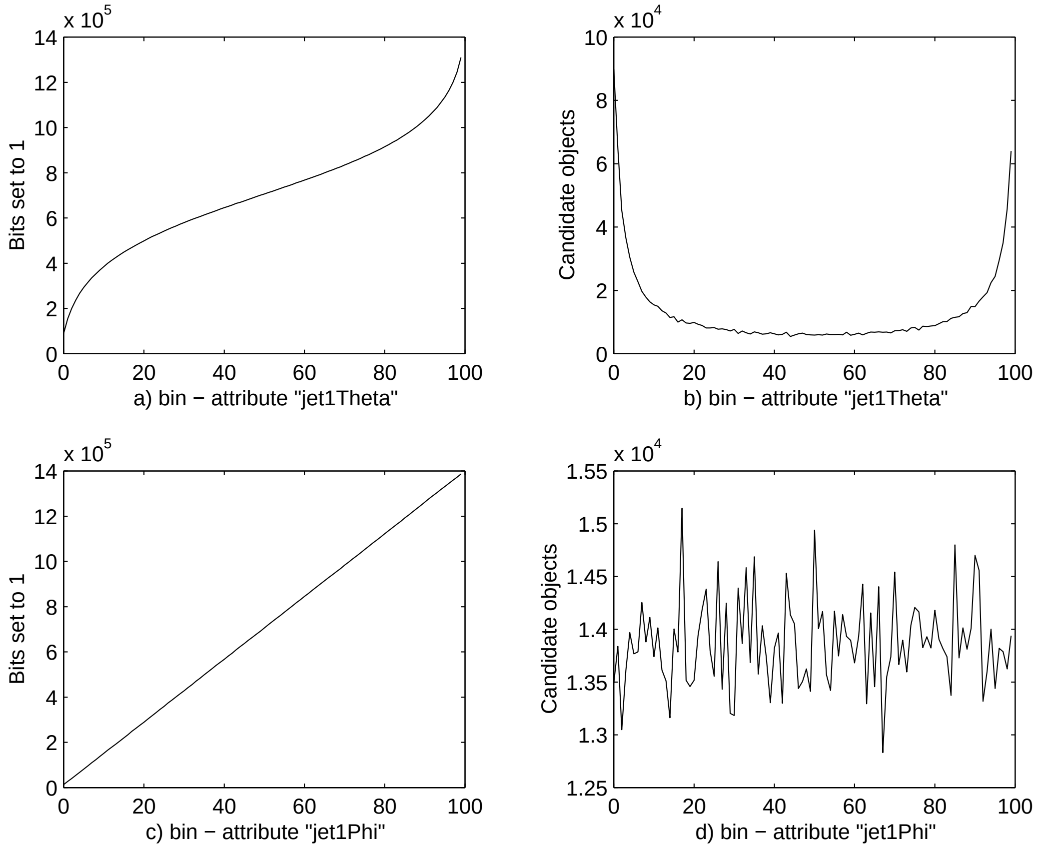

Figure 8.2: Effect of equi-width binning for the attributes `jet1Theta` and `jet1Phi`.

```
Q5: Bitmaps ‘‘jet1E > 1000 && jet2E > 1000 && part1E > 1000 && jet1Theta > 0.5’’

Q6: Bitmaps ‘‘jet1E > 1000 && jet2E > 1000 && part1E > 1000 && jet1Theta > 0.1’’

Q7: Bitmaps ‘‘jet1E > 1000 && jet2E > 1000 && part1E > 1000 && jet1Phi > 0’’

Q8: Bitmaps ‘‘jet1E > 1000 && jet2E > 1000 && part1E > 1000 &&
jet1Phi > 0 && part1Phi > 0’’

Q9: Bitmaps ‘‘jet1E > 10 && jet2E > 10 && part1E > 10 && jet1Phi > 0
&& part1Phi > 0’’

Q10: Bitmaps ‘‘jet1E > 1000 && jet2E > 1000 && part1E > 1000
&& jet1Phi > 0 && part1Phi > 0’’
```

```
Q11: Bitmaps ‘‘jet1E > 10 && jet2E > 10 && part1E > 10 && jet1Phi > 0
&& part1Phi > 0 && jet3E > 10 && jet2Phi && jet2Theta > 0.1’’

Q12: Bitmaps ‘‘jet1E > 10 && jet2E > 10 && part1E > 10 && jet1Phi > 0
&& part1Phi > 0 && jet2E > 10 && jet2Phi > 0 && jet2Theta > 0.1
&& part3E > 10 && part2Phi > 0’’

Q13: Bitmaps ‘‘jet2Theta > 1.4 && jet1Phi > 2 && jet2Phi > 2 && part1Phi > 2
&& part2Phi > 2 && jet1E > 1000 && jet2E > 1000 && part1E > 1000
&& jet3E > 1000 && part2E > 1000’’
```

In Table 8.4.2 the attribute ranges, i.e. minimum and maximum values, for each queried attribute are given.

attribute	min. value	max. value
jet1E	2.8	4429
jet2E	0	4511
jet3E	0	4916
jet2Theta	0	π
jet1Phi	$-\pi$	π
jet2Phi	$-\pi$	π
part1E	2.8	2050
part2E	2.8	1287
part1Phi	$-\pi$	π
part2Phi	$-\pi$	π

Table 8.2: Attribute ranges of queried attributes.

For each of these queries we show the number of page accesses and the query response time (see Table 8.4.2). From the results we see that the bitmap index mostly performs better than the sequential scan when the number of queried attributes is greater than two. In order to understand why the sequential scan performs better for some cases, we need to look at the selectivities of each of these queries and the effected bins of the query ranges. Consider, for example, query `Q2: Bitmaps ‘‘jet1E > 1000’’`. Since the attribute range of `jet1E` is [2.8; 4429] (see Table 8.4.2) and the bitmap index consists of 100 equi-width bins, the effected bin is roughly bin 44. By looking at Figure 8.1 we can see the number of candidate objects for this bin.

As we expect, for one-dimensional queries (`Q1, Q2`) sequentially scanning the data is faster than using the bitmap index. For higher dimensions (starting from five) the bitmap index is up to a factor of two faster. As we can see from these few sample queries, the performance of the bitmap index highly depends on the query selectivities and the effected bins whereas the time for the sequential scan is certainly always constant.

Query	#page I/Os	time [sec]	time seq. scan [sec]
Q1	975	6.9	3.6
Q2	966	5.8	3.6
Q3	1105	6.3	5.5
Q4	1158	6.6	8.0
Q5	1211	6.8	10.1
Q6	1211	6.8	10.1
Q7	1211	6.8	10.1
Q8	1264	7.0	12.3
Q9	4417	16.0	12.3
Q10	1264	7.0	12.3
Q11	6986	20.6	20.6
Q12	7630	23.1	23.2
Q13	3225	13.9	23.2

Table 8.3: Number of pages access and query response time for 13 sample queries.

However, after reorganising the query plan we gain a further performance improvement. In particular, we reordered the query according to the attribute selectivities (as we discussed in the previous chapter). After reordering the query plan for query `Q9` the response time dropped from 16.0 seconds to 15.1 seconds. For query `Q12` the response time dropped from 23.1 seconds to 20.8 seconds. The reordered queries look like follows:

```
Q9: Bitmaps "jet1Phi > 0 && part1Phi > 0 && jet1E > 100
&& jet2E > 100 && part1E > 100"

Q12: Bitmaps "jet1Phi > 0 && jet2Phi > 0 && part1Phi > 0
&& part2Phi > 0 && jet2Theta > 0.1 && jet1E > 10 && jet2E > 10
&& part1E > 10 && jet3E > 10 && part2E > 10 "
```

The previous 13 sample queries were all one-sided range queries including the ">"-operator. In Table 8.4.2 we list the page I/O costs and the query response times for the same queries but now including "<"-operator, i.e. the effected bins are the same but the attribute selectivities differ. Since the result set of the queries is in most cases larger than in the first set of sample queries, the bitmap index only performs better than the sequential scan for high-dimensional queries.

8.4.3 Comparison with the Cost Model

In order to compare the experimental results with the cost model, we evaluated for each queried attribute the effected bin and then calculated the number of page I/Os based on the cost model. The results show that the cost model fairly accurately predicts the I/O complexity for handling

Query	#page I/Os	time [sec]	time seq. scan [sec]
Q1	953	6.8	3.6
Q2	966	5.8	3.6
Q3	1797	9.1	5.5
Q4	1887	9.5	8.0
Q5	2767	10.3	10.1
Q6	2767	10.3	10.1
Q7	2767	9.9	10.1
Q8	3647	12.5	12.3
Q9	3748	13.9	12.3
Q10	3647	12.5	12.3
Q11	4856	17.8	20.6
Q12	4209	14.9	23.2
Q13	6390	20.8	23.2

Table 8.4: Number of pages access and query response time for 13 sample queries.

multi-dimensional queries via the bitmap index. Similar to the results presented in the previous chapter, the resulting error is in the range of 10 to 20% which is accurate enough for a query optimiser to chose between two access plans, namely the sequential scan or an the index scan.

Part of our future work will be to study the effect of correlated attributes and the impact on the candidate reduction factor.

8.4.4 Bitmap Compression

Before we can study the impact of bitmap compression on the query response time of the bitmap index, we will look at the compression ratios of various attributes of queries `Q8` to `Q13`. In Figure 8.3 we can see that attributes `jet1E` and `part1E` show good compressibility, whereas attributes `jet1Theta` and `jet1Phi` only show good compressibility for the edge bins.

We next performed queries `Q8` to `Q13` on the compressed bitmap index. In order to study the effect of compression, we only measured the response time for performing the index I/O rather than including the candidate I/O. In Figure 8.4 a) we plotted the response times for these six one-sided range queries including the "<"-operators. Figure 8.4 b) depicts the response times of the same queries including ">"-operators. In all cases the verbatim bitmap index performs better than the compressed one. This can be explained by the poor compressibility of the attributes `jet1Phi`, `jet2Phi`, `part1Phi`, `part2Phi` and `jet2Theta`.

Finally we carried out a set of benchmarks on attributes that show good compressibility. In particular, we have chosen ten attributes with similar compression ratios to those of attributes `jet1E` and `part1E`. In Figures 8.5 a) and 8.5 b) we again report on queries including the "<"-operator and the ">"-operator. We varied the queries in such a way that the effected bins range from 10 to 90, i.e. the number of candidate objects decreases and thus the compressibility of the effected bins increases. The results show that for queries including the "<"-operator

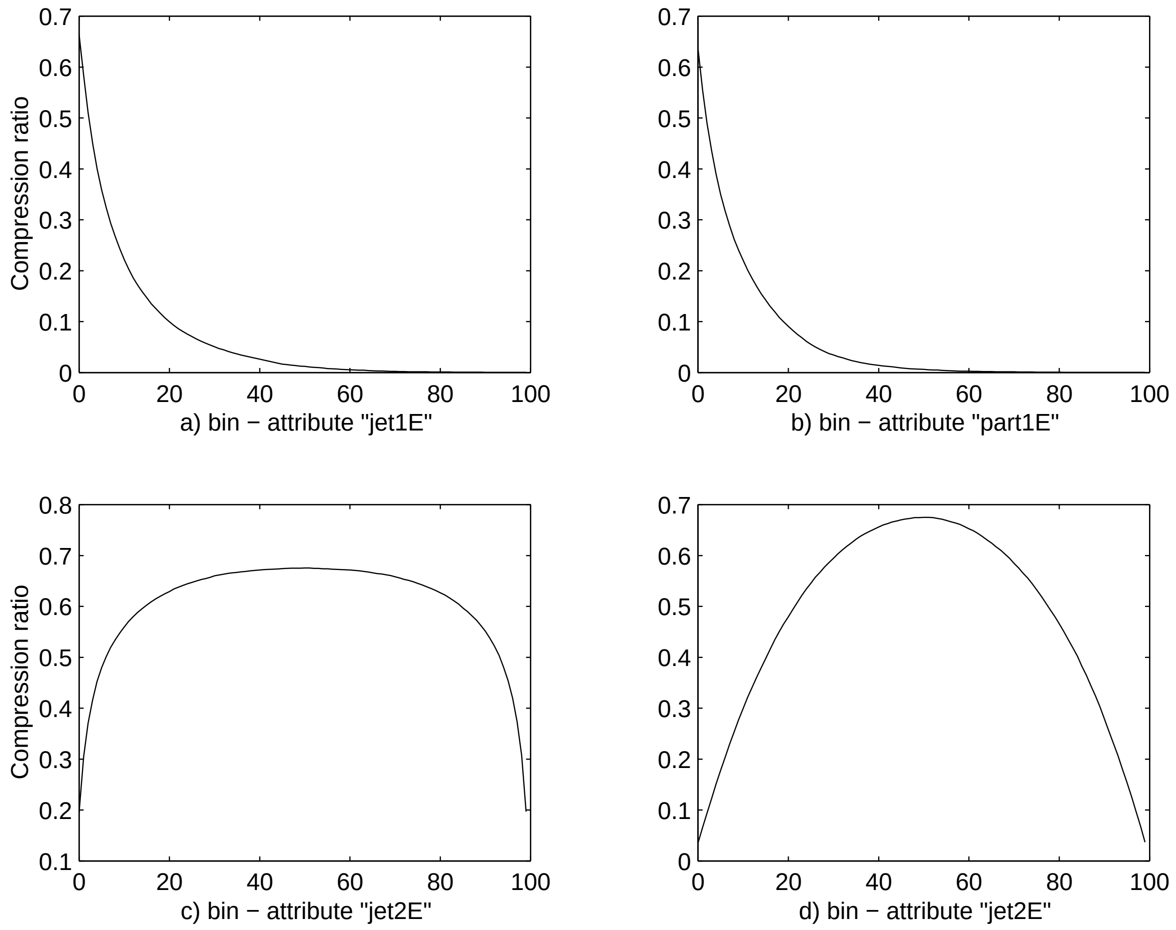

Figure 8.3: Compression ratios for various attributes based on range encoded bitmap index with 100 bins.The compression algorithm is two-sided byte-aligned bitmap compression.

the compressed bitmap index is more efficient than the verbatim one if the effected bins are above 40. For queries including the “>”-operator, the verbatim bitmap index always performs better.

8.5 Astronomy

In addition to applying bitmap indices for speeding up High Energy Physics analysis, we also evaluated the performance of bitmap indices for typical Astronomy queries of the Sloan Sky Server database [61] which is also based on Objectivity. The main part of this work was done during a research visit at the California Institute of Technology.

The Sloan Digital Sky Survey (SDSS) digitally maps about half of the Northern sky in five spectral bands from ultraviolet to the near infrared [63]. In total, some 200 million objects are

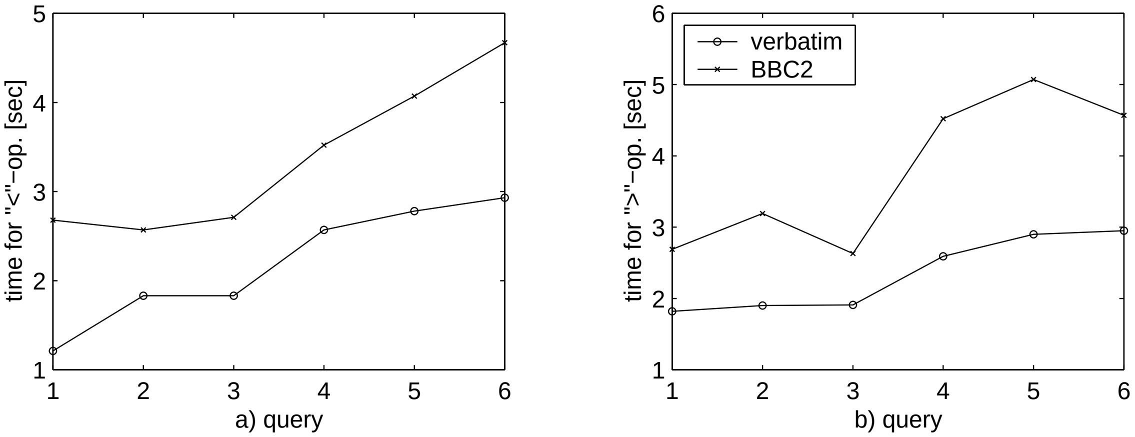

Figure 8.4: Response times for queries `Q8` to `Q13` based on verbatim vs. compressed bitmap indices.

expected to be detected. It will also measure the redshifts for the brightest million galaxies [61]. The complex archive of textual information, derived parameters, multi-band images, spectra and temporal data will allow astronomers to study the evolution of the universe in great detail. The survey is intended to serve as the standard reference for the next several decades. A typical galaxy is depicted in Figure 8.6.

The SDSS is a collaboration between major universities in the USA. Data is taken by a dedicated 2.5-metre telescope at Apache point, New Mexico, USA. The primary targets of observation are galaxies selected by magnitude and surface brightness limit in the r band [63]. The survey will span five to seven years depending on the weather influences.

8.5.1 Data Preparation

All SDSS data is stored in Objectivity/DB and can be retrieved by a special SQL-like query interface developed by the astronomers. Thus, we first ported all the astronomy data to our so-called sliced Tag, reclustered the attributes accordingly and built the indices on top of them. The base objects consist of 6,182,527 tags with 65 attributes each. The total size is 1.9 GB. We created 65 bitmap indices with 100 equi-width bins each. The total size of all indices is 5.8 GB.

We evaluated the performance of the bitmap index against queries taken from the SDSS query server logs. In particular, the logs contained 357 queries of 41 different users. 49 out of these queries are against the data set called "sxGalaxy" which we studied in more detail. We took three representative range queries and compared the performance of the Sloan Sky Server against the performance of the bitmap index. In short, due to a better clustering strategy and the use of bitmap indices, we gained a significant performance improvement of a factor of 10 to 20. The details of these performance tests are discussed in the following sections.

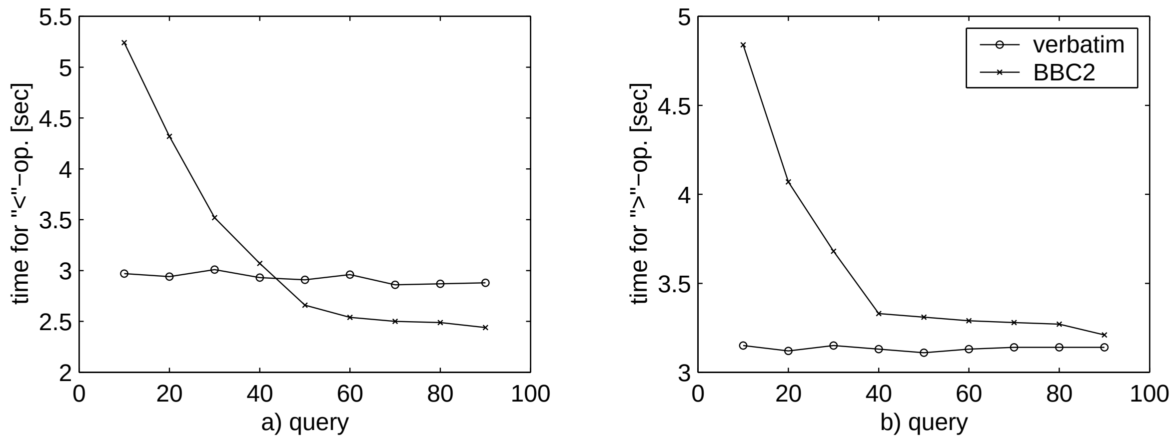

Figure 8.5: Response times for 10 dimensional queries based on verbatim vs. compressed bitmap indices.

8.5.2 "Typical" Astronomy Sample Queries

Most of the 49 queries were 5-dimensional range queries with relatively low total query selectivities. The three queries we picked for further analysis are as follows:

```
Q1:
SELECT g,r,i FROM sxGalaxy
WHERE ((RA() between 180 and 185)
&& (DEC() between 1. and 1.2)
&& (r between 10 and 18)
&& (i between 10 and 18)
&& (g between 10 and 18))

Q2:
SELECT g,r,i FROM sxGalaxy
WHERE ((g-r between 1.05 and 1.13)
&&(r-i between 0.42 and 0.51)
&& (r between 15.68 and 19.68))

Q3:
SELECT u,g,r FROM sxGalaxy
WHERE ((u-g between 0.0 and 0.75)
&& (g-r between 0.0 and 0.5)
&& (u between 18 and 23)
&& (g between 18 and 23)
&& (r between 18 and 23)
&& ((u-g)/(g-r) between 0.8 and 1.2))
```

Figure 8.6: NGC 5792, a highly inclined spiral galaxy.

The query response times for sequentially scanning 6,182,527 objects with various numbers of attributes are given in Table 8.5.2.

Dimensions	data size [MB]	response time
1	29.7	10.3
2	59.5	18.3
3	89.3	28.7
4	119.1	35.6
5	149.1	42.1

Table 8.5: Response time for sequential scan over 6,182,527 objects with various number of attributes (dimensions) - attribute-wise clustering.

The query response times for both the Sloan Sky Server and the bitmap index are as follows. The result set of query `Q1` is 107 objects. The query can be characterised as a typical *index case* since only indexed variables and no compiled ones are used. The query response time of the Sloan Sky server is 127 seconds whereas the response time of the bitmap index is 9.7 seconds. Thus, the bitmap index outperforms the Sloan Sky Server for this kind of query by a factor of nearly 13.

Query Q2 can be characterised as *indexed and compiled attribute case*. Thus, in addition to the normal index scan, also the non-indexed attributes must be considered when evaluating the query. The result set of this query contains 15,418 objects. The query response time of the Sloan Sky Server is 260 seconds and for the bitmap index 13.5, which corresponds to a speedup factor of the bitmap index in the order of 20.

Finally, query Q3 can also be characterised as *indexed and compiled attribute case*. The result set contains 30,372 objects. The response time for the Sloan Sky Server is 521 seconds as compared to 33.4 seconds for the bitmap index. The speedup factor for the bitmap index is roughly 15.

8.5.3 Analysis of the Results

We will now interpret the performance results of the bitmap index more closely by looking at the distribution of the data values and the effect of the binning. In Figure 8.7 a) and c) we plotted the number of set bits in each of the range encoded bit slices of the queried attributes and also the number of candidate objects for each bit slice (Figure 8.7 b) and d)).

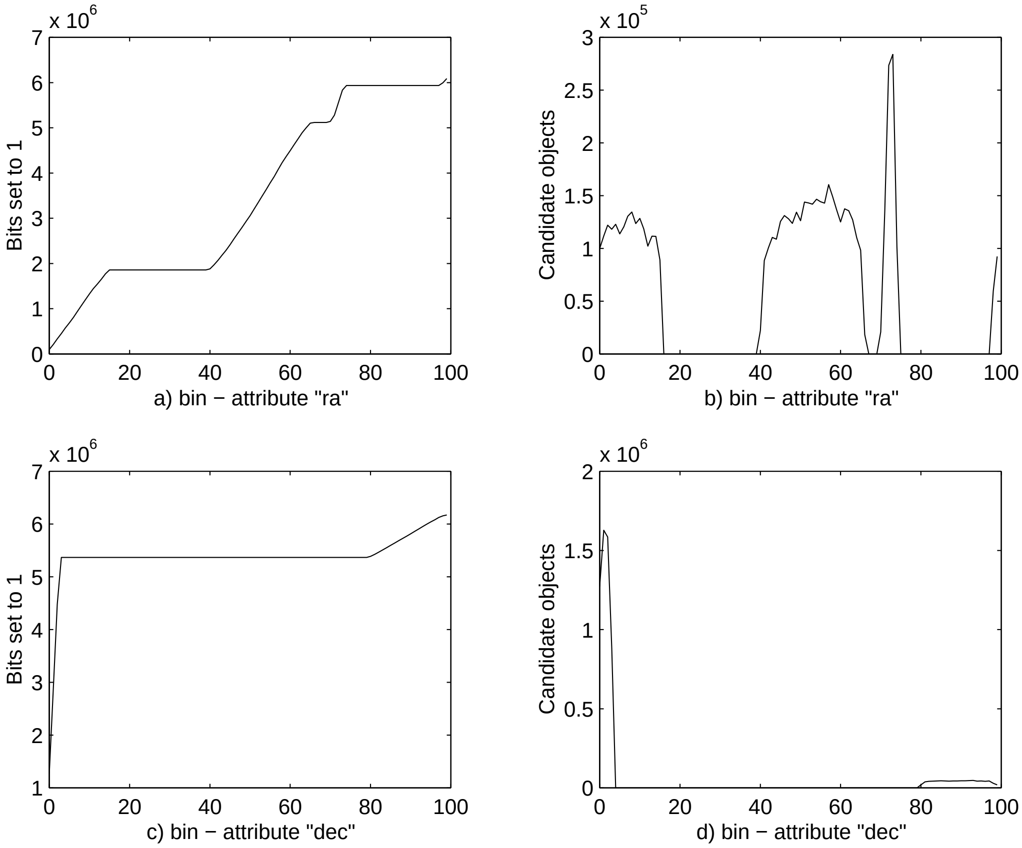

Figure 8.7: Effect of equi-width binning for the attributes ra and dec.

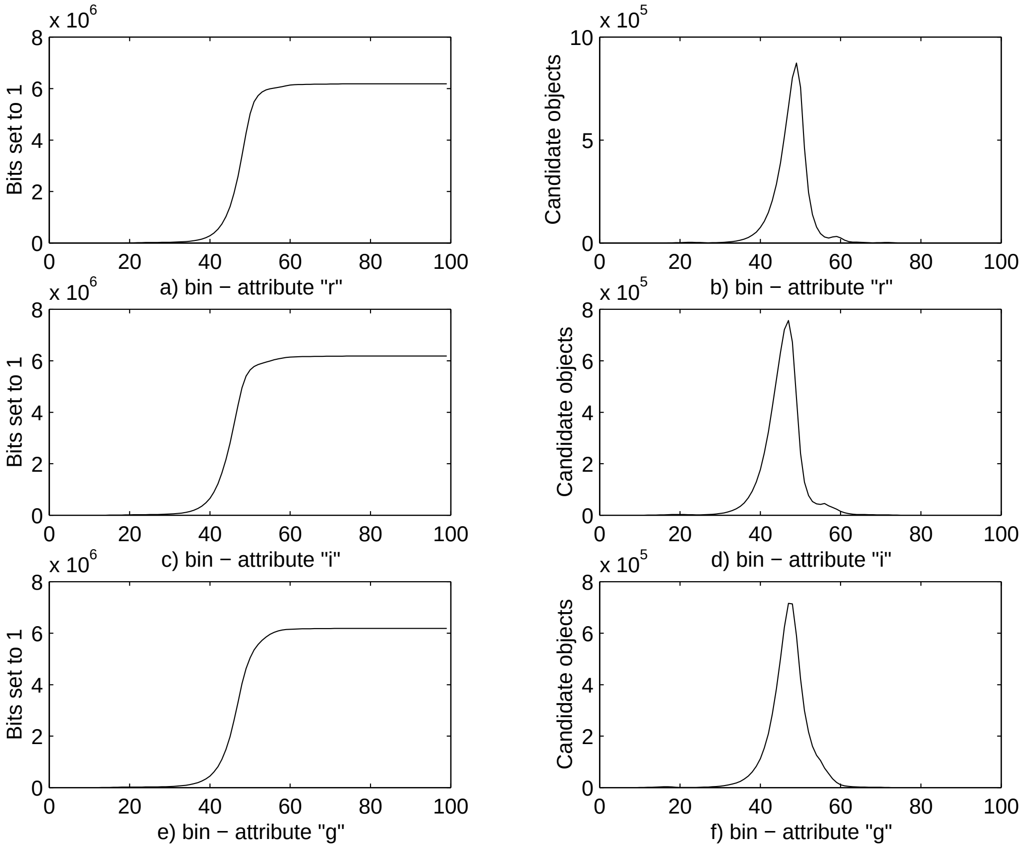

Figure 8.8: Effect of equi-width binning for the attributes `r`, `i` and `g`.

Since we used equi-width bins, the actual distribution of the data is also reflected by the graph which plots the number of candidate objects for each bit slice. For example, for the attribute `ra`, there are no candidate objects for the bins 17 to 40, 68 to 70 and 76 to 98 (Figure 8.7 b)). Attribute `dec` shows an exponential distribution with the bulk of candidate objects in the first few bins (Figure 8.7 d)). The remaining attributes (Figure 8.8 b) d) f)) follow a typical Gauss distribution. We also see that these attributes are highly correlated, whereas `ra` and `dec` do not show any correlation.

Let us now analyse query `Q1` and compute the number of candidate objects for the first three attributes. The attribute range `RA() between 180 and 185` maps to bin 50 and 51 which both contain some 150,000 candidate objects (see Figure 8.7 b)). The attribute range `DEC() between 1. and 1.2` corresponds to bin 3 which holds some 80,000 candidate objects (see Figure 8.7 d)).

The third query range `r between 10 and 18` corresponds to bins 13 and 37 respectively which is the left branch of the Gauss distribution and thus covers only a small number of candidate objects (see Figure 8.8). Since the query range and the distribution for the remaining two attributes is similar to attribute `r`, there are also only a few candidate objects involved.

From this observation we can conclude that the chosen binning, namely equi-with bins, has a positive effect on the number of candidate objects for the query ranges of the last three attributes (`r`, `g` and `i`), whereas the range of the first two attributes (`ra` and `dec`) results in a high number of candidate objects. Thus, if the access patterns of further queries are known, the bin ranges can be chosen in such a way, that the query only results in a low number of candidate objects.

8.6 Conclusions

In this chapter we demonstrated the advantage of using bitmap indices for speeding up typical queries both in High Energy Physics and Astronomy.

We first discussed the implementation of a query parser which allows evaluating any mathematical expression via the bitmap index. In particular, mathematical expressions are compiled and evaluated via the bitmap index on the result set of the queried attributes. This has the advantage that the search space can be pruned by the indexed attributes and complex expressions only need to be applied on a smaller result set.

We carried out various sample queries against real data. Next, we compared the experimental results with the results yielded from the cost model. As we have already stated in Chapter 6 the cost model predicts fairly well the performance of bitmap indices based on uniformly distributed and independent data. In this chapter we showed that given the number of candidate objects per queried attribute, the cost model describes fairly accurately the I/O complexity of the bitmap index.

In addition to verbatim bitmaps indices, we also evaluated the query performance of compressed bitmap indices based on various attributes from High Enery Physics. The compression ratios and thus also the query response times vary a lot among different attributes. However, for certain attributes bitmap compression yields a slight performance improvement.

To sum up, our experimental results, we yielded a performance improvement of bitmap indices up to a factor of two for High Enery Physics queries over ten dimensions and up to a factor of 20 for Astronomy queries over five dimensions.

Chapter 9

Outlook - Query Optimisation in a Grid Environment

9.1 Introduction

In the previous chapters we demonstrated that bitmap indices are an efficient method for speeding up so-called end user physics analysis based on tag data. In this chapter we will give an outlook about access optimisation opportunities for a wider range of data within the HEP data model. One of the tasks of the EU DataGrid [18] project is to perform physics analysis on distributed and replicated data sets all over the world. We thus present an architecture for identifying opportunities for optimising the data access in a typical Grid environment.

9.2 The EU DataGrid

The main idea of the EU DataGrid project is to demonstrate the ability to build, connect and effectively manage large general-purpose, data intensive computer clusters constructed from low-cost commodity components. Besides High Energy Physics, the projects also includes data intensive applications from Earth Observation and Bioinformatics. In short, the entire project consists of several work packages (WP) for middleware development, computing fabric and mass storage management, testbeds and applications (see Figure 9.1). The task "Grid Query Optimisation" is part of WP2 (Data Management) [35, 73] but requires close interaction with WP1 (Workload Management) [72].

9.3 Optimisation Opportunities for Analysis

We define *Grid Query Optimisation (GQO)* as the optimisation of the time and/or cost of execution of a Grid job submitted by a user while performing analysis. In the following the terms job and query will be used interchangeably.

We can consider query optimisation under various points of view [34, 7]:

- **User oriented optimisation (high performance computing).** Each physicist would

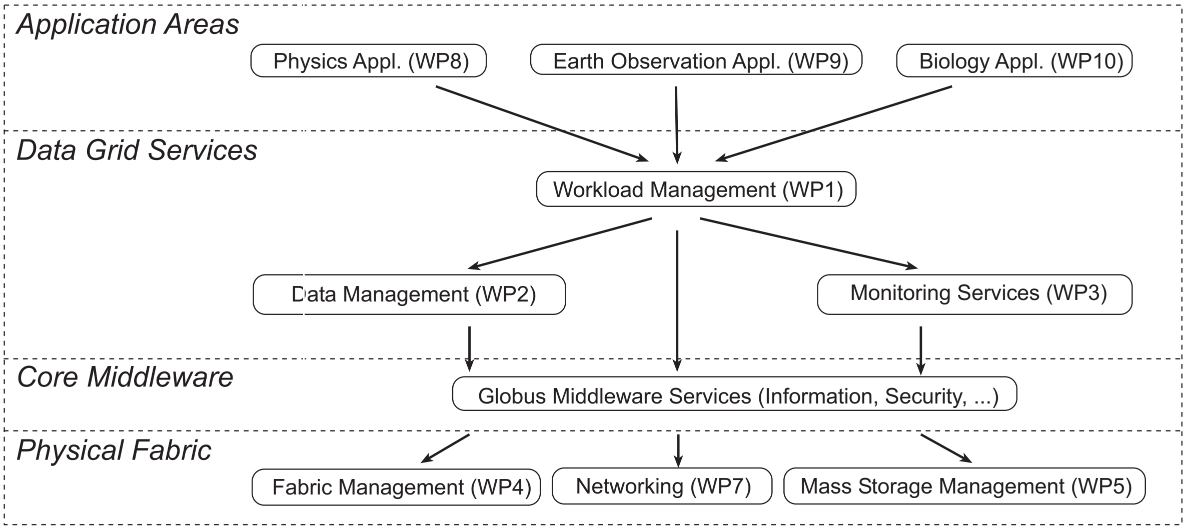

Figure 9.1: Workpackages within the EU DataGrid.

like to perform analysis on HEP events as fast as possible and possibly minimising the cost for execution of jobs that he submits to the Data Grid. The optimal situation for a physicist would be being the only user of the Data Grid, thus having all the Data Grid resources available.

- **Grid oriented optimisation (high throughput computing).** Grid designers have a different perspective on the Data Grid. Their task is to define optimisation services that guarantee certain fairness among all the users of the Data Grid, without favouring any of them (of course, different categories of users can be given different priority of use of the Data Grid). Thus, collective optimisation should be taken into consideration, trying to maximize exploitation of Grid resources while maintaining acceptable time/cost for execution of single physicist jobs.

- **Site oriented optimisation.** Administrators of Data Grid sites would like to decide upon local policies of use for resources at that site. These policies will influence the usage of the site by Grid and non-Grid jobs.

The optimisation services provided by the Data Grid should be designed to take into consideration all three of these perspectives and the right trade-off between them. We can consider two kinds of optimisation:

- **Short-term optimisation of user queries.** This optimisation can be activated whenever a query is submitted to the Data Grid and aims at minimizing the execution time and cost of the job. Optimisation is based on the information specified by the job: input data set and job code. We identify some possibilities for optimisation.

- **Job decomposition.** A job usually operates on a set of events independently. This means that a job can be decomposed as a set of sub jobs, whose code corresponds to the code (or part of the code) of the main job, and an aggregation job, that executes on the output data set of the sub jobs. Job decomposition can be performed on the basis of where the input data products are stored in the Data Grid, or on the structure of the job code.
- **Job execution time estimation.** This information could be exploited by the user in order to decide whether or not to submit a certain job to the Data Grid, according to how long he is willing to wait for job completion. Execution time execution can be based on both the location of data and/or code execution time estimation for each of the sub jobs that compose the job.
- **(Sub)Job dispatching.** Where to dispatch a (sub)job is another important issue to take into consideration for GQO. The optimal location for the execution of a job depends on the location and current status of both the required data and the required computation resources. A trade-off between data optimisation and computation optimisation is important to assure Grid oriented optimisation. As far as data optimisation is concerned, there are two important aspects
  * *Data selection.* The same input data product for a job could be replicated in several files, placed in different locations on the Grid. The most convenient file copy should be selected for use by the job.
  * *Data relocation.* The relocation and replication of files inside and between sites must be performed in an optimal manner.
- **During execution optimisation.** When a job requests a set of data that is not available on the site where the job is running, a decision has to be taken on how to optimally access the missing data. Therefore, data selection and relocation could also be performed during the execution of the job, to meet the optimisation needs that arise due to unforeseen data requirements.

- **Long-term optimisation of use of Data Grid resources.** As previously stated, the optimised use of Grid resources improves the overall performance of the Data Grid and thus, on average, the performance of jobs submitted by single physicists. Long-term optimisation can be based on statistics on the past use of the Data Grid and forecasts of its future use.
  - *Replication.* A possibility of long-tem optimisation is to set up a suitable replication policy for files stored in the various sites of the Data Grid. For example, suppose that several jobs submitted to the Data Grid from sites placed in the same area have been accessing much the same set of files. Then, an immediate optimisation is to replicate those files in a site easily accessible from the sites in that area. Analogously, if a set of jobs to be executed in the future in a certain area of the Data Grid are going to use a similar set of files, these files could be replicated in advance and stored in sites included in that area. We evaluated different replication scenarios in [19, 66].

– *Reclustering.* Input data products of a job might be sparsely spread over several large files. Reclustering them into a smaller set of files prior to their analysis could improve execution time of the job, that would fully exploit the content of the files. A possibility is to set up some *reclustering policy* for data product. For example, if several jobs are going to access almost the same set of data products in the future, it could be convenient to store these data into the same file (or a set of files).

9.4 Grid Query Optimisation for Analysis within a Typical Grid Architecture

In this section we present a brief overview of a Grid Architecture from the point of view of "query optimisation" [34]. Our aim is not to define an explicit "Grid Query Optimisation Service" per se, but rather to discover which services will be required in order to optimise data access in a Grid environment. In particular we will focus on typical HEP use cases for distributed physics analysis. The work was done in collaboration with the "Query Optimisation" team of the "Data Management" Workpackage.

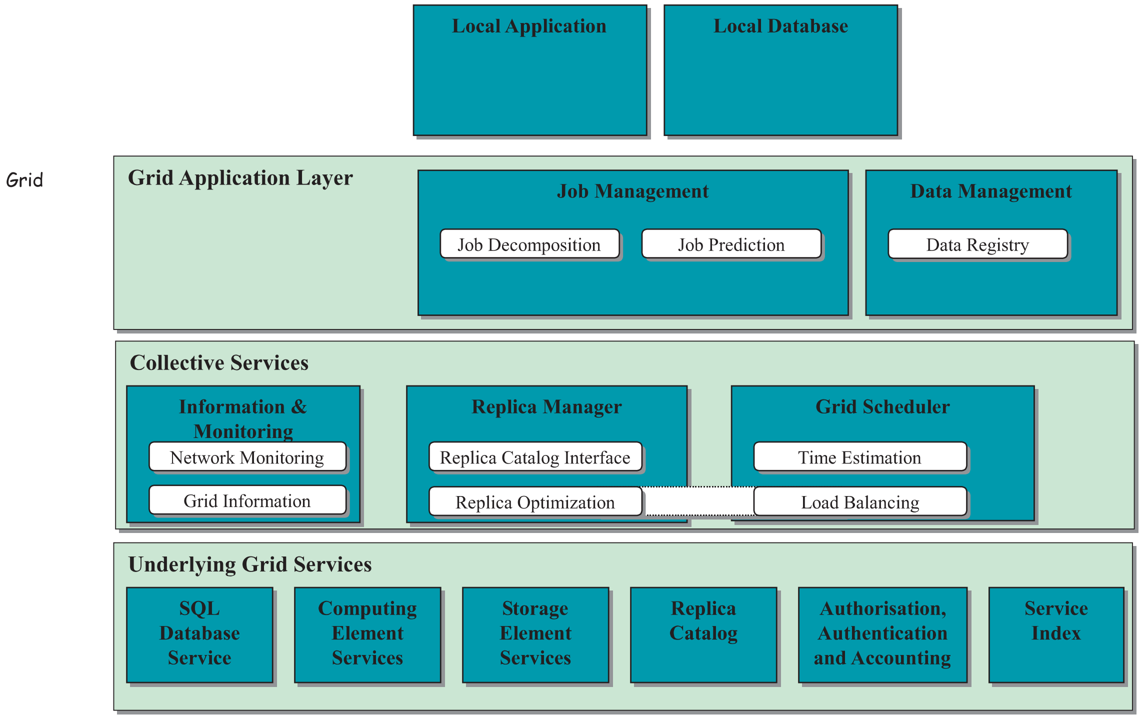

Figure 9.2: Grid Architecture from the point of view of "Grid Query Optimisation".

9.4.1 The Local Application Layer

The top layer of the architecture, the *Local Application Layer*, exists outside of the Grid infrastructure. It is in this layer that TAG data analysis will most likely be performed. TAG data will be stored on locally owned storage devices, using a locally managed database system. The optimisation of TAG data analysis is also part of the task "Grid Query Optimisation", and will be achieved through the *Bitmap Indexing* of TAG data values [67, 68]. Once the physicist has exhausted the local supply of information, he begins to analyse on lower level Grid managed data, by submitting a job to the Grid via the local Grid Client.

9.4.2 The Grid Application Layer

Job Management

This application level *Job Manager* will submit each job to the *Grid Scheduler* in the form of a well defined *Job Description Language* (JDL, see [72]). There is need, therefore, for *Job Decomposition* functionality within the *Job Management* system, so as to reformulate jobs into (graphs of) atomic computation steps ready for submission to the Grid. Only the applications themselves can provide enough information to fulfill such a decomposition task. One architecture (not necessarily advocated here) would be to define a standard interface to a *Job Decomposition* service, which some/all Grid applications could implement, such that the *Grid Scheduler* could access decomposition services at execution time. Another approach would be to decompose all jobs completely before passing them in JDL form to the *Scheduler*. The main output of the *Job Management* would thus be a logically decomposed job which could then be further optimised by the Scheduler together with the Replica Manager/Optimiser (as defined in [34]).

We also propose the need for a *Job Prediction* component within each Grid application. Such a component would be responsible for collecting statistics on the execution of Grid jobs, and for making predictions based on those statistics in terms of:

- Which logical files / data collections a (analysis) job is likely to require.
- How long a job is likely to run for.

The collection of statistics for the first prediction would require that database access to files/data collections is logged in some way, and that the log information is made available to the *Job Prediction* component. (This may be done either by the Grid intercepting calls to the database, or by the database itself logging its own activities.) Assuming the gathering of statistics is possible, why would such predictions be important, and why would the information not be known already? In fact, in many analysis type jobs, it is impossible to know a priori what data the job will access, or how long it is likely to run for. Estimations made by the *Job Prediction* unit would then be used by the *Scheduler* to optimise the execution of the job. Since the similarity between different jobs can only be assessed on a semantic level (i.e. by the application submitting the job), the *Job Prediction* unit needs to be co-located with the Grid application. The unit may also use other information, such as the physicist's user profile, or processing hints the physicists have given, to make predictions for job execution. As was the case for the *Job Decomposition* system, there are two mechanisms by which the prediction information can be passed from the Grid application to the *Scheduler*. The first option is to include such information as hints in the JDL (i.e. to generate all the prediction information a priori). The second option is to define an interface for the *Job Prediction* unit which would be used by the *Scheduler* to make predictions if and when required.

9.4.3 The Collective Services Layer

We define a few submodules as part of our discussion of optimisation. In the sections below the functionalities of these modules with respect to optimisation are discussed.

The Information and Monitoring Service

The *Information and Monitoring Service* will aggregate information provided by monitoring services such as the *Network Monitoring Service* (monitoring traffic on Virtual Private Network links between sites) and the *Fabric Monitoring Service* (monitoring the load on computational resources at the different sites).

The Replica Manager

The aim of the *Replica Optimisation* module is to automatically replicate data throughout the Grid in such a way as to minimise the total cost of data access for all the jobs executing on the Grid. This is part of what we called "long term optimisation". Within a site the module might control which files remain staged on disk and which ones are relegated to tape storage. Between sites the module would control which files are replicated and which ones are not. The question then becomes, how does the *Replica Optimiser* decide which files to replicate and which ones not to? i.e. how does the *Replica Optimiser* know which files will be in demand on the Grid and which ones will not? It could do this by:

- collecting its own statistics; (It would be too late to collect information at this point, if the mapping between data collections and files is not constant across the Grid, i.e. if internal data reclustering at each site causes data to be stored differently at each site.)
- asking the *Grid Scheduler*
- accessing standard services of the application layer *Job Prediction* module or by
- receiving hints directly from the application. (The application level *Job Prediction* unit could assign importance levels to different logical files / data collections, which the *Replica Optimiser* would then to decide where and how many replicas to create.)

A second function of the *Replica Optimiser* (a kid of short term optimisation) is to find the best replica [55, 11] of a file when the file is demanded by the *Scheduler*. In this case the *Replica Optimiser* can decide whether it should create a new replica of the file locally, create a temporary copy of the file locally, or (possibly) open the file on the remote location for remote access. In the case where it decides to create a local replica or copy of the file, the *Replica Optimiser* must use an interface provided by the *Data Registry* module of the application level *Data Management* system to import the new replica/copy into the local database implementation.

The Grid Scheduler

Other important functions of the *Replica Optimiser* are to supply the *Time Estimator* module of the *Scheduler* with time estimates for the retrieval of files, and to negotiate with the *Load Balancer* to determine the best location for executing a given job, based on both the data and computational requirements of the job. The negotiation with the *Load Balancer* may require close coupling between the two components (as shown in Figure 9.2).

The *Scheduler* provides high level scheduling services to Grid applications such that applications need not know where and how to schedule their work on the Grid, but can simply define the constraints for running a job (such as the amount of memory required, the input data collection needed, etc.) and allow the Grid to schedule it for them. The applications can then view the Grid as a single enormous computation and storage resource. One of the functionalities that needs to be provided by the *Scheduler* is that of *Time Estimation* for the execution of a Grid job. A time estimate is used by the Grid application or the physicist to decide whether or not to run a job. The *Time Estimator* uses information from the *Job Prediction* module, the *Information and Monitoring* services, and the *Replica Optimiser* to calculate an approximate time for job execution.

The *Load Balancing* unit is responsible for the actual scheduling of jobs to different sites on the Grid. It takes as input a decomposed job from the Grid application, and negotiates with the *Replica Optimiser* to discover the best location for job execution, based on the availability of both data and computational resources. (The *Replica Optimiser* uses information from the monitoring services to compare the costs for replicating data to different sites.)

9.4.4 The Task of Grid Query Optimisation within a Typical Architecture

The primary task of GQO Task is to build a major part of the Replica Optimiser module of the Replica Manager. In this section we look more closely at the required functionality of such a module. The functionality can be viewed in terms of the five "services" it offers:

- **Data Access Time Estimation.** Aid the Time Estimation unit of the Scheduler to calculate approximate data access times for jobs that might be submitted to the Grid. The Replica Optimiser accesses the Network Monitoring Service (to discover the current network bandwidth situation), a Storage Device Monitoring Service (to discover the current load on the devices), and the Replica Catalog (to discover the amount of replication of the required files), so that an estimate of the cost of data movement can be returned to the Scheduler.

- **Pre-Execution Optimisation.** Aid the Load Balancing unit of the scheduler to make scheduling decisions, (i.e. help the Load Balancer decide on which site to run a job). An optimised scheduling decision should take into account both the cost of data movement between sites and the computational load on sites. A trade-off between data optimisation and computation optimisation will be achieved through the use of a negotiation protocol between the Replica Optimiser and the Load Balancer. (This negotiation/interaction protocol is still to be defined.)

- **During Execution Optimisation.** If a job requests a set of data at a particular site, and the data is unavailable at that site (e.g the database method returns an exception, and the exception is caught by the Replica Optimiser), then the Replica Optimiser needs to make a decision on how to optimally access the missing data. The Optimiser can then choose between possibly five options:
  - Open data for remote read on a site with a fast network connection.
  - Copy the data locally, register and create a "permanent" replica of the data.
  - Copy the data locally, register a "temporary" replica and de-register it subsequently.
  - Ask the Scheduler to reschedule and restart the job on another site (e.g. a Tier 1 or 0 site).
- **Post-Execution Optimisation.** The aim of this optimisation is to automatically distribute (create replicas of) the Grid managed output files created by jobs running on the Grid. (Such output files are primarily only created by production type jobs, which are not the main focus of our work). In making decisions on to what extent to replicate the output files, the Optimiser relies on hints from the user submitting the job, as well as heuristics information on the use of similar sets of data.
- **Offline Optimisation.** The aim here is to monitor the usage of logical files / data collections on the Grid as a whole and try to match the supply of replicas to the demand for them.

As well as creating the major part of the Replica Optimiser, the GQO task involves helping the applications groups (Work Packages 8 to 10 [18]) to create the Grid Application Layer services they require in order to make optimal use of the Grid. The services of interest include the Job Decomposition, Job Prediction and Data Registering service. The assistance would be in terms of defining standard interfaces to these components and possibly helping to create generic code for use in each application's implementation of the components.

9.5 Interaction of Services for a Particular HEP Use Case

We now look at a particular HEP use case described previously in the context of the architecture described above.

At the start of the use case, the physicist performs "cuts" on the entire data set, by specifying that he is only interested in those events for which certain conditions on TAG attribute values hold. The local application sends these "cut predicates" to a local (bitmap) indexing system, which returns a list of events adhering to the selection. The physicist then performs some sort of statistical analysis on the events returned by the indexing system, studies the results and repeats the process. Since TAG data is mostly stored locally, the Grid is probably unaware of the analysis being performed by the physicist. After exhausting the information available in the TAG data, the user writes analysis code and submits it as a job to the Grid. The local application sends this code, along with the names of the AOD objects of interest and an output location for the results of the computation, to the *Grid Application Layer*.

The Grid level application reformulates the request into a job capable of execution on the Grid. It does this by mapping all of the requested AOD objects to a set of logical files or to a data collection description. It may also decompose the job into smaller subjobs via the *Job Decomposition* service. Having reformulated the request, the application then submits it to the Grid for an estimation of execution time and cost, by sending a request to the *Scheduler* which provides an interface to the services of the *Time Estimator*. The Time Estimator requests information from the *Job Prediction* module (data and time requirements of the job), the *Monitoring Services* (current computational load on the Grid), and the *Replica Optimiser* (cost of data retrieval) to calculate an approximate time for job execution. When the application receives the execution time estimate, it uses that information to schedule the execution of the job on an application level, based on the cost of the job, the user submitting the job, the status of its job queue, etc. Of course the Grid application could also decide to refuse to schedule the job, or just to send the time/cost estimate back to the local application for approval. For more precise estimates, the *Time Estimator* could also contact the *Load Balancer* (see below).

In order to decide on which site to schedule the job, the *Load Balancer* enters into a negotiation process with the *Replica Optimiser*. The *Load Balancer* first uses the constraints given in the JDL job description (such as memory requirements, software library availability, etc.) to select a set of possible sites for job execution. It then requests information from the *Fabric Monitoring* service and the *Job Prediction* service in order to calculate the computation cost for job execution on each of these possible sites. It also sends the list of possible sites to the *Replica Optimiser*, so that the optimiser can use information from the *Network Monitoring* service and the *Replica Catalog* to calculate the minimum cost of staging/using/creating the required data at each of the possible sites. The *Replica Optimiser* then provides this information to the *Load Balancer*, so that the Balancer can schedule the job at the site with the lowest overall cost. (The "negotiation" between the *Load Balancer* and the *Replica Optimiser* given above is highly simplified, and implies a full search of the available search space, to achieve a global minimum. Such an exhaustive search may not be possible or be simply inefficient, in which case more complicated forms of negotiation may be required.)

The *Load Balancer* then asks the *Replica Optimiser* to stage all required files to the site selected for execution. Once the files have been staged, the *Load Balancer* dispatches the job for execution to the selected site, using the *Computing Element* service.

During job execution, the navigational access within the job causes it to demand a set of data which is not available in the local database. The *Replica Optimiser* catches the exception thrown by the database, and remedies the situation by either opening the data for WAN access on a remote site, copying the data locally to create a temporary/permanent replica, or asking the scheduler to stop the job and restart it (from the beginning) on another site. (The latter might be the case if a job on a Tier 2 site say, wants to start accessing Raw data, in which case restarting the job on a Tier 0 or 1 site, would probably be more advisable than moving Raw data to the Tier 2 site.) . Certainly the "navigational" case can also be handled by the *Replica Manager*. However, since the input data set is not known in advance, no explicit optimisation techniques concerning the optimal replica selection can be applied. Also a possible estimation about the access time for this job cannot be made. The only possibility for optimisation would be to deliver the requested files from a "cache".

Once the job has reached completion, the *Replica Optimiser* is responsible for deleting any temporary replicas created for the execution of the job. (Deletion also implies the "deregistering"; of the data from the local database).

9.6 Conclusions and Future Work

Our aim now is to simulate the Grid environment, and to attempt to optimise data access within such an environment. More specifically, the simulator has the following goals [34]:

- To build a system that can realistically simulate a Grid environment, in which multiple autonomous resources must be managed coherently. The model will include simulations of the main parts of the architecture components discussed above. We will simulate a "simple" application requesting a set of logical files. We will also simulate major parts of the Replica Manager and the Grid Scheduler to study and optimise the complex "negotiation process" between these components. In particular, we plan to simulate optimal selection of replicas [66]. Based on access patterns, a further goal is to study the impact of "automatically" creating data replicas between sites.

- The simulator will help testing different algorithms and heuristics for making such replication decisions, based on their ability for optimising globally the use of data resources (disk arrays and tape pools) on the Grid.

Additional goals in building the simulator include:

- To study the effects of local policy decisions on the overall working of the Grid system. For instance, what is the impact of reducing the disk quota for a certain user community? How shall a user community with a high priority be handled?

- To validate the design and the interfaces of the different Grid components.

A final goal is that the Replica Optimiser - initially a simulator only - is a software component which is part of the Replica Manager. This additional software component uses monitoring and performance information and optimises replica selection based on current performance parameters and predictions in the Data Grid.

Chapter 10

Conclusions

Efficient query processing in high-dimensional search spaces is an important requirement for many analysis tools. In the literature on index data structures one can find a wide range of methods for optimising database access. In particular, bitmap indices have recently gained substantial popularity in data warehouse applications with large amounts of read mostly data. Bitmap indices are implemented in various commercial database products and are used for querying typical business applications. However, scientific data that is mostly characterised by non-discrete attribute values cannot be queried efficiently by the techniques currently supported.

In this thesis we proposed a novel access method based on bitmap indices that efficiently handles multi-dimensional queries against typical scientific data. The algorithm is called `GenericRangeEval` and is an extension of a bitmap index for discrete attribute values. By means of a cost model we studied the performance of queries with various selectivities against uniformly distributed and independent data values. Experimentally we verified our analytical findings and demonstrated that for certain query selectivities the proposed bitmap index shows a significant performance improvement over traditional access methods.

Next, we studied the impact of bitmap compression on the query performance for different bitmap encoding techniques. We showed that for uniformly distributed data values, equality encoded bitmap indices show good compressibility whereas range encoded bitmap indices show bad compressibility. By comparing compressed equality encoded bitmap indices with verbatim range encoded bitmap indices we showed that the first approach performs better than the latter for range queries with very low query selectivities. However, for a majority of the queries, verbatim range encoded bitmap indices have better performance characteristics. In addition, for non-uniformly distributed data the query response time of range encoded bitmap indices can further be improved with bitmap compression.

Apart from evaluating bitmap indices for synthetic data, we also evaluated this access method based on real data taken from High Energy Physics and Astronomy applications. We thus demonstrated that our approach is not only of theoretical value but also improves the performance of practical applications. In this sense this thesis was the first successful proof that multi-dimensional access methods can significantly speed up typical end-user analysis.

Finally, we discussed the problem of access optimisation for distributed analysis in a Grid environment with data replicated all around the globe. Again, our proposed bitmap indices can be used during end user analysis filtering out the most important physics properties in a highly efficient way. Based on these results, scientists can even further drill down into different kinds of replicated data.

We want to conclude this thesis with a citation of Alexander Nikitenko, a physicist from the CMS experiment, "One of the crucial points is the fast access to the data. Who accesses it faster, makes physics discoveries first."

Chapter 11

Acknowledgements

My first thank you goes to my "doctor father" Prof. Erich Schikuta who encouraged me to do a Ph.D. at CERN, the European Organization for Nuclear Research and was thus the "main trigger" for this work. I also want to thank Dirk Duellmann, my Ph.D. supervisor at CERN, for many hours of very useful discussions. It was a pleasure to get used to the physics terminology and discover the sometimes very different "language" to what I was used to from the "computer science world".

I also want to thank my second "doctor father" Prof. Quirchmayr, my "boss" and group leader at CERN, Jamie Shiers, Ben Segal, one of the Grid pioneers at CERN, and Prof. Harvey Newman who hosted me for three very interesting and challenging months at the California Institute of Technology.

Warmest thanks go to my colleagues and friends Wolfgang Hoschek and Koen Holtman with whom I spent endless hours in the cantine of CERN to discuss sometimes very emotionally all kinds of computing issues.

Special thanks go to my friend Veronique Lefebure who introduced me to the exciting field of physics analysis. Reading her physics thesis helped a lot in understanding some of the problems of CERN's hunt for the Higgs boson.

Last but not least I want to thank my twin brother Heinz who always patiently listened to my questions I addressed over the phone.

Bibliography

[1] G. Antoshenkov, Byte-Aligned Bitmap Compression, *Technical Report, Oracle Corp., 1994*

[2] S. Amer-Yahia, T. Johnson, Optimizing Queries On Compressed Bitmaps, *Proceedings of 26th International Conference on Very Large Data Bases*, Cairo, Egypt, Sept. 2000, Morgan Kaufmann.

[3] R. Bayer, E. M. McCreight, Organization and Maintenance of Large Ordered Indices, *Acta Inf. 1,3, 173-189*, 1972.

[4] J. L. Bentley, Mulidimensional Binary Search Trees, *Communications of the ACM 18, 9, 509-517,* 1975.

[5] S. Berchtold, C. Boehm, H.-P. Kriegel, The Pyramid-Tree: Breaking the Curse of D imensionality, SIGMOD 1998, *Proceedings ACM SIGMOD International Conference on Management of Data*, Seattle, Washington, USA, June 1998.

[6] P. Busetta, M. Carman, M. Nori, L. Serafini, F. Zini, Using BDI Agents for a DataGrid Simulator, *http://sra.itc.it/events/agents4Grid/program.html*

[7] P. Busetta, M. Carman, L. Serafini, K. Stockinger, F. Zini, Grid Query Optimisation in the Data Grid, *Technical Report, TR-01 09-01*, IRST, Trento, Italy, Sept. 2001.

[8] R. Bellman, Adaptive Control Processes: A Guided Tour, 1961, *Princeton University Press.*

[9] S. Berchtold, D. Keim, H.-P. Kriegel, The X-tree: An Index Structure for High-Dimensional Data, *In Proceedings of the International Conference on Very Large Databases*, Mumbai (Bombay), India, September 1996.

[10] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The R*-tree: An Efficient and Robust Access Method for Points and Rectangles, *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, Atlantic City, New Jersey, May 1990.

[11] M. Carman, F. Zini, L. Serafini, K. Stockinger, Towards an Economy-Based Optimisation of File Access and Replication on a Data Grid, *submitted for publication.*

[12] European Organization for Nuclear Research. *http://www.cern.ch*

[13] S. Chaudhuri and U. Dayal, An Overview of Data Warehousing and OLAP Technology, *ACM SIGMOD Record 26(1)*, March 1997.

[14] C. Chan, Y.E. Ioannidis, Bitmap Index Design and Evaluation, *In Proceedings ACM SIGMOD International Conference on Management of Data*, Seattle, Washington, USA, June 1998.

[15] C. Chan, Y.E. Ioannidis, An Efficient Bitmap Encoding Scheme for Selection Queries, *Proceedings ACM SIGMOD International Conference on Management of Data*, Philadephia, Pennsylvania, USA, June 1999.

[16] CMS - Compact Muon Solenoid. http://cmsinfo.cern.ch/

[17] D. Comer, The Ubiquitous B-tree, *ACM Computing Survey 11, 2, 121-138*, 1979.

[18] European DataGrid Project, *http://www.eu-datagrid.org*

[19] D. Düllmann, W. Hoschek, J. Jaen-Martinez, A. Samar, B. Segal, H. Stockinger, K. Stockinger. Models for Replica Synchronisation and Consistency in a Data Grid. *10th IEEE International Symposium on High Performance and Distributed Computing (HPDC2001)*, San Francisco, California, August 7-9, 2001.

[20] I. Foster and C. Kesselman, The Grid: Blueprint for a New Computing Infrastructure, *ISBN 0-97028-467-5*, July 1998.

[21] R. Fagin, J. Nievergelt, N. Pippenger, R. Strong, Extendible Hashing: A Fast Access Method for Dynamic Files, *ACM Transactions on Database Systems 4, 3, 315-344*, 1979.

[22] C. Faloutsos, S. Roseman, Fractals for Secondary Key Retrieval, *Proceedings of ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Databases*, 1989.

[23] M. Freeston, A General Solution of the n-dimensional B-tree problem, *In Proceedings of the ACM SIGMOD International Conference on Management of Data, San Jose, California, May 1995.*

[24] J.-L. Gailly, M. Adler, Zlib home page. *http://quest.jpl.nasa.gov/zlib/.*

[25] V. Gaeda, O. Guenther, Multidimensional Access Methods, *ACM Computing Surveys 30, September 1998.*

[26] J. Gillies, R. Cailliau, How the Web Was Born - The Story of the World Wide Web, *ISBN 0-19-286207-3, Sept. 2000, Oxford University Press.*

[27] A. Guttman, R-trees: A Dynamic Index Structure for Spatial Searching, *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, Boston, Massachusetts, June 1984.

[28] A. Henrich, H.-W. Six, P. Widmayer, The LSD tree: Spacial Access to Multidimensional Point and Non-Point Objects, *In Proceedings of International Conference on Very Large Databases*, Amsterdam, The Netherlands, August 1989.

[29] HepODBMS, *http://wwwinfo.cern.ch/db/objectivity/docs/hepodbms*

[30] K. Hinrichs, Implementation of the Grid File: Design Concepts and Experiences, *BIT 25, 569-592*, 1985.

[31] K. Holtman. Prototyping of CMS Storage Management, *Ph.D. thesis (proefontwerp), Eindhoven University of Technology, ISBN 90-386-0771-7*, May 2000.

[32] K. Holtman, P. van der Stok, I. Willers. A Cache Filtering Optimisation for Queries to Massive Datasets on Tertiary Storage,*Proc. of DOLAP'99*, Kansas City, USA, November 1999.

[33] K. Holtman, P. van der Stok, I. Willers. Automatic Reclustering of Objects in Very Large Databases for High Energy Physics, *Proc. of IDEAS*, Cardiff, UK, 1998.

[34] W. Hoschek, J. Jaen-Martinez, P. Kunszt, B. Segal, H. Stockinger, K. Stockinger, B. Tierney, Data Management (WP2) Architecture Report: Design, Requirments and Evoluation Criteria, *http://grid-data-management.web.cern.ch/grid-data-management/meetings.html*

[35] W. Hoschek, J. Jean-Martinez, A. Samar, H. Stockinger, K. Stockinger. Data Management in an International Data Grid Project. *1st IEEE/ACM International Workshop on Grid Computing (Grid'2000).* Bangalore, India, Dec 2000.

[36] A. Hutsfles, Twin Grid Files: Space Optimizing Access Schemes, *In Proceedings of ACM SIGMOD International Conference on Management of Data*, Chicago, Illinois, June 1988.

[37] A. Hutflesz, H.-W. Six, P. Widmayer, The R-file: An Efficient Access Structure for Proximity Queries, *In Proceedings of Interational Conference on Data Engineering*, 1990.

[38] M. Juergens, H.-j. Lenz, Tree Based Indexes vs. Bitmap Indexes: A Performance Study, *International Workshop on Design and Management of Data Warehouses*, Heidelberg, Germany, June 1999.

[39] T. Johnson, Performance Measurements of Compressed Bitmap Indices, *Proceedings of 25th International Conference on Very Large Data Bases*, Edinburgh, Scotland, UK, September 1999, Morgan Kaufmann.

[40] N. Koudas, Space Efficient Bitmap Indexing, *International Conference on Information and Knowledge Management*, McLean, VA, USA, November 2000.

[41] I. Kamel, C. Faloutsos, Hilbert R-tree: An Improved R-tree Using Fractals, *Proceedings of International Conference on Very Large Data Bases*, Santiago de Chile, Chile, September 1994.

[42] Monarc Project: Models of Networked Analysis at Regional Centres for LHC Experiments,*http://monarc.web.cern.ch/MONARC, January 2000.*

[43] T.A. Mueck, Vorlesung aus Algorithmen und Datenstrukturen, *University of Vienna*, 1996.

[44] A. Moffat, J. Zobel, Parameterized Compression of Sparse Bitmaps, *Proceedings SIGIR Conference on Information Retrieval*, 1992.

[45] J. Nievergelt, The Grid File: An Adaptable, Symmetric Multikey File Structure, *In Proceedings of the Third ECI Conference*, 1981, Springer Verlag.

[46] P.A. Larson, Linear Hashing with Partial Expansions, *In Proceedings of Sixth International Conference on Very Large Databases*, Montreal, Quebec, Canada, October 1980.

[47] D. B. Lomet, B. Salzberg, The hB-tree: A Multiattribute Search Structure, *In Proceedings of International Conference on Data Engineering*, 1989.

[48] Objectivity/DB, *http://www.objectivity.com*

[49] Object Data Management Group, *http://www.odmg.org*

[50] P. O'Neil, Model 204 Architecture and Performance, *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, Asilomar, CA, USA, 1987.

[51] P. O'Neil, Informix and Indexing Support for Data Warehouses, Database and Programming Design, February 1997.

[52] P. O'Neil, D. Quass, Improved Query Performance with Variant Indexes, *Proceedings ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, USA, May 1997.

[53] Physics Analysis Workstation - PAW, *http://www.cern.ch/paw*

[54] RD45 - A Persistent Object Manager for HEP,
http://wwwinfo.cern.ch/asd/rd45/index.html

[55] K. Ranganathan and I. Foster. Identifying Dynamic Replication Strategies for a High Performance Data Grid. *In Proc. of the International Grid Computing Workshop*, Denver, CO, November 2001.

[56] J. T. Robinson, The k-d-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Ann Arbor, Michigan, April, 1981.

[57] H. Samet, The Quadtree and Related Hierarchical Data Structure, *ACM Computing Survey 16, 2, 187-260*, 1984.

[58] A. Shoshani, L.M. Bernardo, H. Nordberg, D. Rotem, A. Sim, Multidimensional Indexing and Query Coordination for Tertiary Storage Management, *11th International Conference on Scientific and Statistical Database Management*, Cleveland, Ohio, USA, July 1999.

[59] E. Schikuta. Grid-clustering: An efficient hierarchical clustering method for very large data sets. *In Proc. 13th Int. Conf. on Pattern Recognition*, Vienna, Austria, October 1996. IEEE Computer Society Press.

[60] L. Serafini, H. Stockinger, K. Stockinger, F. Zini. Agent-Based Query Optimisation in a Grid Environment, *IASTED International Conference on Applied Informatics (AI2001)*, Innsbruck, Austria, February 2001.

[61] Sloan Digital Sky Survey, *http://www.sdss.org*

[62] B. Seeger, H.-P. Kriegel, The buddy-tree: An Efficient and Robust Access Method for Spatial Database Systems, *In Proceedings of International Conference on Very Large Databases*, Brisbane, Queensland, Australia, 1990.

[63] A.Szalay, P. Kunszt, A. Thakar, J. Gray, D. Slutz, Designing and Mining Multi-Terabyte Astronomy Archives: The Sloan Digital Sky Survey, *International Conference on Management of Data*, Philadephia, Pennsylvania, USA, June 1999.

[64] T. Sellis, N. Roussopoulos, C. Faloutsos, The R+-Tree: A Dynamic Index for Multi-Dimensional Objecs, *Proceedings of International Conference on Very Large Databases*, Brighton, England, September 1987.

[65] H. Six, P. Widmayer, Spacial Searching in Geometric Databases, *In Proceedings of International Conference on Data Engineering*, 1988.

[66] H. Stockinger, K. Stockinger, E. Schikuta, I. Willers. Towards a Cost Model for Distributed and Replicated Data Stores, *9th Euromicro Workshop on Parallel and Distributed Processing (PDP 2001)*, Mantova, Italy, February 7-9, 2001, IEEE Computer Society Press.

[67] K. Stockinger, D. Duellmann, W. Hoschek, E. Schikuta. Improving the Performance of High Energy Physics Analysis through Bitmap Indices. *In International Conference on Database and Expert Systems Applications*, London - Greenwich, UK, Sept. 2000. Springer-Verlag.

[68] K. Stockinger, Design and Implementation of Bitmap Indices for Scientific Data, *International Database Engineering & Applications Symposium*, Grenoble, France, July 2001, IEEE Computer Society Press.

[69] K. Stockinger, Performance Analysis of Generic vs. Sliced Tags in HepODBMS, *International Conference on Computing in High Energy and Nuclear Physics*, Beijing, China, September, 2001.

[70] M. Tamminen, The Extendible Cell Method for Closest Point Problems, *BIT 22, 27-41*, 1982.

[71] M. Wu, A.P. Buchmann, Encoded Bitmap Indexing for Data Warehouses, *Proceedings of the Fourteenth International Conference on Data Engineering*, Orlando, Florida, USA, February 1998.

[72] WP1 - Workload Management Work Package, *http://www.infn.it/workload-grid*

[73] WP2 - Data Management Work Package, *http://grid-data-management.web.cern.ch/grid-data-management*

[74] M. Wu, Query Optimization for Selections Using Bitmaps, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data*, Philadephia, Pennsylvania, USA, June 1999.

[75] K. Wu, P.S. Yu, Range-Based Bitmap Indexing for High-Cardinality Attributes with Skew, *Technical Report, IBM Watson Research Center*, May 1996.

[76] J. Ziv, A. Lempel, Compression of Individual Sequences via Variable-Rate Coding, *IEEE Transactions on Information Theory IT-24(5)*, Sept. 1978.