Efficient Bitmap Indexing Techniques for Data Warehouses and Scientific Databases

THESIS

Submitted in partial fulfilment

of the requirements for the degree of

DOCTOR OF PHILOSOPHY

by

Yashvardhan Sharma

Under the Supervision of

Prof. Navneet Goyal

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE

PILANI (RAJASTHAN) INDIA

2008

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
PILANI (RAJASTHAN) INDIA

C E R T I F I C A T E

This is to certify that the thesis entitled "**Efficient Bitmap Indexing Techniques for Data Warehouses and Scientific Databases**" submitted by **Yashvardhan Sharma,** ID. No. **2001PHXF417** for award of **Ph.D. Degree** of the Institute embodies original work done by him under my supervision.

(Signature in full of the supervisor)
DR. NAVNEET GOYAL
Associate Professor
Birla Institute of Technology and science
Pilani – 333 031 (Rajasthan) INDIA

Date:
Place: Pilani

ABSTRACT

Data Warehouses and Scientific Databases pose a great challenge to the database community to improve query performance as they contain huge volumes of dimensional data. Moreover, the queries in such systems are, complex, multidimensional, and have large foot prints, often requiring millions of records to be answered. Many different kinds of techniques have been proposed to improve the query performance in such environments to provide interactive response time to users.

Indexing techniques play a major role in improving query performance in Data Warehouses and Scientific Databases. Conventional hash-based and tree-based one-dimensional indexing techniques like linear hashing, extensible hashing, B^+-tree are found lacking because of the multi-dimensional nature of the queries. In most cases it is found that doing a complete table scan is much cheaper than using an index.

Multi-dimensional index data structures are an important optimization technique for querying high-dimensional search spaces in read-mostly environments and are supported by major commercial database systems. Bitmap indexes are efficient multi-dimensional index data structures for high-dimensional data arising in data warehousing, decision support systems, and in some scientific databases. The main advantages of using bitmap indexes is that they are highly amenable to compression and encoding and bitmap manipulations using bit-wise operators AND, OR, XOR, NOT are very efficiently supported by hardware.

One of the major issues with bitmap indexes is the associated space overheads especially for high-dimensional and high-cardinality data. In recent past, a number of approaches have been proposed to reduce the index size and improve the performance of the bitmap indexes. These approaches include encoding, compression, and binning. In the thesis, an attempt has been made to develop more efficient encoding, compression, and binning and techniques for bitmap indexes.

Byte-aligned bitmap code (BBC) and word-aligned hybrid code (WAH) are two especially designed compression schemes for bitmap indexes. Some data preprocessing methods have been proposed to make both BBC and WAH more space and time efficient in answering equality and range queries.

Data reorganization, mainly tuple reodering, is a technique to improve the compression ratios achieved by BBC and WAH. Multi-component encoding has been used as a preprocessing technique to improve the compression ratio achieved by Gray code ordering algorithm used for tuple reordering.

Bitmap indexes are suitable for low cardinality attributes. A number of scientific data analysis applications have attributes with cardinality in millions. High cardinality attributes pose unique challenges in terms of keeping the space requirements within manageable limits and at the same time maintaining acceptable response time for queries. Binning is a common technique used to reduce the size of the bitmap index. Although, binning leads to considerable space savings, it gives rise to the candidate check problem which can require a lot of additional disk I/Os thereby affecting query performance adversely. A new binning strategy is proposed for high cardinality attributes which attempts to minimize the number of candidate checks, for a given set of queries, at the expense of space. Some optimization techniques for performing candidate checks have also been developed.

ACKNOWLEDGEMENTS

First of all I would like to thank my supervisor Prof. Navneet Goyal for his valuable guidance, encouragement and moral support. It has been a great pleasure to be associated with him on this work.

Next I want to thank Prof. R. K. Mittal, who introduced me to the field of databases. He has been a role model and a great source of encouragement to me.

Special thanks are due to Prof. L. K. Maheshwari, Vice Chancellor, Prof. K. E. Raman, Deputy Director (Administration), Prof. G. Raghurama Deputy Director (Academic), Prof. Ravi Prakash, Dean, Research and Consultancy Division, for giving me an opportunity and encouragement for completing this thesis. I would like to express my sincere gratitude to Prof. S. C. Sivasubramanian, for his constant encouragement and support. Thanks are also due to Dr. Sudeept Mohan for providing valuable comments while going through initial drafts of the thesis. I sincerely acknowledge the help received from Prof. J. P. Mishra, Unit Chief, Information Processing center Unit. He helped me with suggestions on trying out various new binning strategies. I sincerely thank members of the doctoral advisory committee, Prof. Rahul Banerjee and Dr. Mukesh Rohil, for their critical remarks and help in improving the work.

I would also like to thank Prof. S. Balasubramanium, Group Leader, CS & IS for his constant support and guidance.

I received encouragements and support from my colleagues, Dr. T S B Sudershan, Dr. Chitranjan Hota, Mr. Pankaj Vyas and Mr. Biju Ravindran. I express my sincere thanks to them. Lastly, I also thank my ex-colleagues and friends Mr. Dinesh Sharma, Mr. Girish Pujari and Mr. S. Jagdish for their advice and motivation.

Last but not the least the completion of this thesis would not be possible without the love and support from my family. I am grateful to them for the continuous encouragement and support during the years of my research.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	Details or Expanded Form
BBC	Byte-Aligned Bitmap Compression
BMI	Bit Map Index
BSI	Bit Sliced Index
DBEC	Dynamic Bucket Expansion and Contraction
DBMS	Data Base Management System
DML	Data Manipulation Language
DW	Data Warehouse
EBI	Encoded Bitmap Indexes
ETL	Extraction Transformation and Loading
EVI	Encoded Vector Indexes
HEP	High Energy Physics
LSB	Least Significant Bit
MSB	Most Significant Bit
OLAP	On-Line Analytical Processing
OLTP	On-Line Transactions Processing
RID	Row Identifier
SBI	Simple Bitmap Indexes
SNAP	Super Nova Acceleration Probe
TPC	Transaction Processing Performance Council
VLDB	Very Large Data Bases
WAH	Word-Aligned Hybrid Compression

LIST OF MATHEMATICAL NOTATIONS

Notation	Details
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
$B_{d, s}$	sth bit slice of the bitmap index for the dth dimension
Op_d	predicate operator for dimension d where Op_d belongs to (<, <=, >, >=)
q_d	query range of dimension d
$s (q_d)$	lower limit($(q_d-l_d)/w_d$) where l_d is the absolute lower bound of the search space
P	Total number of pages on disk for values of an attribute
P_b	The expected number of disk pages that contain data values that fall into bin b
$q = [l_q, u_q)$	A range query q with endpoints l_q and u_q, open on the right
Q	A set of range queries
x_i	A bin boundary point
$b_i = [x_{i-1}, x_i)$	A bin defines a sub-range open on the right
$B=< b_1, b_2, \ldots ,b_k >$	A partitioning into *k* bins
E(b)	The set of queries having bin *b* as an edge bin
Cost(Q,B)	Candidate check cost associated with binning B and query set Q
e_j	The jth smallest query endpoint
r	Number of distinct query endpoints
$EP (Q) = (e_1 \ldots e_r)$	Ordered set of distinct query endpoints
$\Pi(k, n)$	All possible binnings of the range 1 to *n* into *k* bins
$w_i = \sum q \in E(b_i)\, p_q$	Weight of bin b_i, the sum of probabilities of all queries in E(b)
R(Q, *j*)	The set of queries in Q with a right endpoint on the right of e_j
$B_{opt}(e_j , l)$	Optimal binning of the sub-region from e_j to *n* using *l* bins
$b_{i,j}$	A bin defined over the range between query endpoints e_i and e_j ,i.e., $b_{i,j} = [e_i, e_j)$

B_{opt}	Optimal binning
N_b	Number of Bins
x_u	Upper limit of bin
x_l	lower limit of bin
f	query frequency
b	Bin containing the result
b_f	Bitmap equal to size of one bin
R_i	Number of records in i^{th} bin
b_l	Lower Bin
b_h	Higher Bin
R	Number of records in bin
R_m	Minimum number of records
x	Value to be searched for equality queries

Chapter 1: Introduction

Widespread adoption of database systems in many areas of human activity witnessed in recent years led to rapid increase in the volume of data gathered by computers. Common employment of bar code readers, automated call centers, Web application interfaces, and other facilities induces an unfaltering stream of raw data loaded continuously into databases. The ever increasing volume of the gathered data makes manual processing of the data impossible, necessitating the development of efficient techniques and technologies for extraction of information/knowledge from vast data repositories.

Data is one of the most valuable assets of an organization, and when used properly, can assist in decision making that can in turn significantly improve the functioning and profitability of an organization. Data Warehousing is a technology that allows information to be easily and efficiently accessed for decision making activities by collecting data from many operational, legacy, and heterogeneous source systems. Data in data warehouses are accessed/analyzed through On-Line Analytical Processing (OLAP) tools which well-suited for complex data analysis, such as multi-dimensional data analysis, and assist in decision support activities. Many organizations have either invested heavily or are planning to invest in the data warehousing technology for fulfilling their decision support needs.

A data warehouses is a huge repository of data that is available for downstream data analytics applications. OLAP tools provide data analysis functionality which in turn, helps in the decision making process. Data warehouses and OLAP applications differ significantly from the traditional database applications. Online Transactional Processing (OLTP) or operational systems carry out the day to day operations of a business and are highly optimized for answering repetitive and narrow queries. Data warehouses and OLAP provide a different context in which huge amounts of data must be processed efficiently and queries are often complex and ad-hoc, but still require interactive response times. In data warehouse environments, the data is used for decision support and large sets of data are read and analyzed in an *ad-hoc* manner. Data warehouses tend to be extremely large, for example, the data warehouse of General Motors, exceeds 1.5

terabytes in size, and contains a fact table with more than 2 billion rows in which queries can take hours to complete.

Data warehousing refers to “a collection of decision support technologies aimed at enabling the knowledge worker (executives, managers, and analysts) to make better and faster decisions” [Choudhari and Dayal, 1997]. In simple terms, a data warehouse is a “very large” repository of historical data pertaining to an organization. The data warehouse can also be defined as “*a repository of data that has been extracted and integrated from heterogeneous and autonomous distributed sources*” [Kimball, 1996].

The significance of data warehousing is evidenced by the recent growth in the number of related products and services offered in the market for data warehousing, including hardware, database software, and specialized tools. These technologies are gaining widespread acceptance in a multiple fields including retail sales, telecommunications and financial services.

OLAP refers to the technique of performing complex analysis over the data stored in a data warehouse. Data warehouses are large, special-purpose databases containing data from a number of independent sources, supporting trend and anomaly analysis. The information stored in a data warehouse is clean, static, integrated, and time varying [Inmon, 1993]. The process of analysis is usually performed with queries that aggregate, filter, and group the data in a variety of ways. As the queries are often complex and the warehouse database is often very large, processing the queries efficiently is a critical issue in the data warehousing environment.

Data warehouses are typically updated periodically, in a batch fashion. The batch update process sometimes *reorganizes* data and indexes to a new optimal clustered form. As during this process the warehouse is unavailable for querying, it is possible to create specialized indexes and materialized aggregate views (called *summary tables* in data warehousing literature) which in turn help in evaluating queries efficiently. Relational DBMS (RDBMS) technology is the best understood technique to deal with large data sets. However they were not primarily designed keeping in mind the data warehousing and OLAP requirements. A host of techniques used in the relational environment for improving query performance in data warehouses are discussed in section 1.1.

Data warehouses tend to grow rapidly. To handle data explosion and provide interactive response time, highly scalable architecture is vital in a data warehouse environment. Data Marts play a vital role in both top-down and bottom-up approaches for building data warehouses to address scalability and query response time issues. Most of today's OLAP tools require data warehouses with a centralized structure where a single database contains all the data. However, the centralized data warehouse is expensive to setup and lacks structural flexibility. More importantly, "the world is distributed", world-wide enterprises operate in a global manner and do not fit in a centralized structure. Thus, a new paradigm is necessary. The first step in a new direction was the recent introduction of data marts, "small data warehouses" containing only data on specific subjects, business processes or departments [Informatica, 1997, HP, 1997]. But this approach doesn't solve the problems of space and performance. Data marts provide more flexibility in the distribution of data but they still consist of static, self-contained units with fixed locations. By distributing small static portions of data to fixed locations, the system becomes more flexible, but on the other hand new problems arise, related to intra data mart communication, especially in what concerns the processing of queries. Many of today's data marts are basically standalone, because of the unsophisticated and rudimentary integration in the global data warehouse context. In spite of the potential advantages of data marts, especially when the organization has a clear distributed nature, these systems are always very complex and pose difficult global management challenges [Albrecht et. al, 1998].

There are three major differences between transaction-oriented operational systems and data warehouse systems:

- **Size of the data:** Fast access to GB (10^9 bytes) or TB (10^{12} bytes) of data is crucial in providing interactive decision support.
- **Dynamics of data:** In a typical data warehouse, data is inserted, but exceptionally updated or deleted. Furthermore, insertion only takes place at certain time windows when the system is not accessible to the analysts. Outside these time windows, analysts use the system only for reading data. This strategy is typical for *read-mostly* environments.

- **Type of queries:** Typical queries in an operational system access data on a very detailed level, such as the *balance of a specific bank account*. Typical queries in data warehouse environments calculate aggregated data over large sets of data, such as *sum of sales on product groups for some time period*. Therefore, the access to aggregated data over large sets of data has to be supported efficiently.

Data warehousing/OLAP systems are best understood by comparing them to OLTP systems. OLTP systems are designed to automate data processing tasks (e.g., order entry), which are structured and repetitive, tasks that operate on detailed data. The emphasis in such systems is placed on maximizing transaction throughput. In contrast to OLTP, data warehouses are designed for decision support purposes and contain historical data of many years. For this reason, data warehouses tend to be extremely large containing hundreds of gigabytes to terabytes of data. OLAP applications are characterized by the rendering of enterprise data into multidimensional perspectives, which is achieved through complex, ad-hoc queries that frequently aggregate and consolidate data. Thus, OLAP environments are query-intensive, where aggregated and summarized data are much more important than detailed individual records. Typical OLAP queries require computationally expensive operations such as joins and aggregation. All such queries are performed on tables having millions of records but still interactive response time is expected. Given these characteristics, it is clear that the emphasis in OLAP environments is on efficient query processing. This area has caught the fancy of database researchers. A number of "conventional" relational query processing approaches have been applied to or extended for answering OLAP queries with acceptable response times.

Many scientific applications such as high-energy physics, climate modeling, bioinformatics , astrophysics etc., coupled with advances in technology have resulted into generation of massive volumes of data through observations or computer simulations, bringing up the need for effective techniques for efficient storage and retrieval of scientific data. Unlike conventional databases, scientific databases are mostly read-only and its volume can reach to the order of petabytes, thus making a compact index structure a vital necessity. In computational high-energy physics, simulations are continuously run, and events that are notable for physicists are stored with all the details. The number of events that need to be stored in one year are of the order of several millions [SciDAC, 2002]. In astrophysics, technological advances enabled devoting several telescopes for observations, results of which need to be stored for later query processing [SNAP, 2004]. Genomic and proteomic technologies are now capable of generating terabytes of data in a single day's experimentation [Zaki and Wang, 2003]. These new data sets and the associated queries are significantly different from those of the traditional database systems, mainly due to their enormous size and high-dimensionality (more than 500 attributes in high-energy physics experiments). This poses a new challenge for efficient storage and retrieval of data.

Most of the scientific databases of practical interest are read-only, just like the data warehouses. Various types of queries, such as partial match and range queries are executed on these large data sets to retrieve useful information for scientific discovery. As an example, a user can pose a range query to retrieve all events with energy less than 15 GeV, and the number of particles less than 13. When the data is large and read-only, as in the case of scientific databases, conventional techniques are not effective for improving the performance of querying and data analysis. Thus developing new query performance enhancing techniques tailored for scientific databases is crucial to effectively exploring such data.

1.1. Query Performance Enhancing Techniques

There are several strategies to improve query response time in the data warehouse context: indexing techniques, materialized views, parallelism, and partitioning of data. These techniques are briefly discussed below:

1.1.1. Indexing Techniques

A commonly used technique to improve the performance of queries is the use of *index structures*. Index structures avoid full table scans for answering narrow queries (queries which require only a small fraction of the total tuples), thereby considerably reducing the response time. Different indexing techniques have been investigated in much detail for operational databases during the last few decades, but very little work has been done for finding suitable indexing structures for data warehouse systems as they pose different kind of challenges than operational systems. Most of the queries on a data warehouse involve joining of large tables. Aggregate functions are also very commonly used in these queries. Such complex queries could take several hours or days to process large amount of data. A majority of requests for information from a data warehouse involve dynamic ad-hoc queries [TPC, 1998, APB, 1998]; users can pose any business query at any time for any reason on the data warehouse data. If the right index structures are created, the performance of queries, especially ad-hoc queries are greatly enhanced. The indexing requirements of OLAP systems are:

- **Symmetric partial match queries**: Most of the OLAP queries can be expressed as a partial range query or continuous range query, i.e., a query like "list total sales from January 2005 to December 2007". As queries can ask for ranges for any dimension, all the dimensions of the data cube should be *symmetrically* indexed, such that they can be searched simultaneously.
- **Indexing at multiple levels of aggregation**: It is typical that OLAP systems pre-compute data at different levels of aggregation, in order to speed up queries. These are called materialized views. These views must be indexed in the same way that the non-aggregated data.
- **Efficient batch update:** We have already said that updates are not so critical in OLAP systems, allowing more columns to be indexed. However, sometimes, the update window is not enough for data updating, which must be taken into account while designing the indexing schema.
- **Sparse data:** About 20% of the data in an OLAP system are non-zero. Indexing must be able to deal efficiently with sparse and non-sparse data. Modern indexing techniques and query processing strategies attempt to fulfill these requirements.

Due to the scale and high dimensionality of data warehouses and scientific databases, simple extensions of traditional indexing strategies are inadequate: R-trees and its variants are well-known to lose effectiveness for high dimensions; hashing-based indexes lack storage efficiency; and transformation based approaches are not effective for partial match and range queries. Furthermore, most of the indexing approaches do not focus on the size of the index structure itself. However, due to the huge data volume in a data warehouses and scientific databases, the size of the indexing structure becomes as important as other parameters and must be taken into account. Focusing on the major characteristics of business and scientific data, such as being read-only, having special access patterns and numerical attributes, researchers have managed to develop indexing techniques that are feasible for high dimensional databases. Many indexing techniques have been created to reach this goal in read-mostly environments.

Indexing techniques are among the first areas on which a database administrator will focus when good query performance in a read intensive environment is critical. Specialized indexing structures offer the optimizer alternatives access strategies for the time consuming full table scans. One of the most popular index structures is the B-tree and its derivatives [Comer, 1979]. B^+ tree indexes are the most commonly supported structures in RDBMS, but it is a well-known fact that tree structures have limitations when the cardinality of the attribute is small.

Another class of index structures, the bitmap indexes, attempts to overcome the problem by using a bit structure to indicate the rows containing specific values of the indexed attribute [O’Neil, 1997]. Although essential for the right tuning of the database engine, the performance of index structures depends on many different parameters such as the number of stored rows, the cardinality of the data space, block size of the system, bandwidth of disks and latency time, only to mention some [Jurgens, 1999].

1.1.2. View Materialization

View materialization is a technique in which pre-computed results are stored in the database. Most commercial database systems support materialized views. In materialized views, we generally store aggregated data (summary tables or summary indexes) or joins or both. The use of materialized views is probably the most effective way to speed up a specific set of queries in a data warehouse environment. Materialized views pre-compute and store (materialize) aggregates and joins, the two most commonly used operations in OLAP queries [Chaudhuri, 1997]. The data is grouped using categories from the dimensions tables, which corresponds to the subjects of interest (dimensions) of the organization. Storing all possible aggregates poses space problems and increases the maintenance cost, since all stored aggregates need to be refreshed as and when the data warehouse is refreshed. Many algorithms have been proposed for selecting a representative subset of the possible views for materialization [Harinarayan, 1996, Meredith, 1996, Ezeife, 1997], corresponding to the most usual query patterns. But the main problems associated with materialized views are the difficulty to know in advance the expected set of queries, the problems of updating materialized views to reflect changes made to base relations and the large amount of space required to store the materialized views. There are limitations to the concept of materializing views at the data warehouse. Pre-computation of queries in materialized views can give answers quickly but the number of views that should be materialized at the warehouse needs to be controlled, otherwise this can result in data explosion. Selection of views to be materialized at the data warehouse is one of the important issues related to view materialization. Another challenge in data warehousing is how to maintain the materialized views. When there is a change in the data at any source, the materialized views at the data warehouse need to be updated accordingly. The process of keeping the views up-to-date in response to the changes in the source data is referred to as view maintenance. For efficiency reasons, incremental techniques are preferred over re-computing the view from scratch, for view maintenance. In data warehousing, the view maintenance has branched into a number of sub-problems such as self maintenance, consistency maintenance, update filtering, and online view maintenance.

The technique of view materialization is hampered by the fact that one needs to anticipate the queries to materialize at the warehouse. The queries issued at the data warehouse are mostly ad-hoc and cannot be effectively anticipated at all times. The performance when using summary tables for predetermined queries is good. However when an unpredicted query arises, the system must scan, fetch, and sort the actual data, resulting in performance degradation. Whenever the base table changes, the summary tables have to be recomputed. Also building summary tables often supports only known frequent queries, and requires more time and more space than the original data. Because we cannot build all possible summary tables, choosing which ones to be built is a difficult job. Moreover, summarized data hide valuable information. For example, we cannot know the effectiveness of the promotion on Monday by querying weekly summary. Indexing is the key to achieve this objective without adding additional hardware. It is worth noting that the techniques mentioned above (indexes and materialized views) are general techniques that can (and should) be used in the data warehouse approach.

1.1.3. Parallel Processing

A large body of work exists in applying parallel processing techniques to relational database systems with the purpose of accelerating query processing [Lu et al., 1994, DeWitt and Gray, 1992]. The basic idea behind parallel databases is to carry out evaluation steps in parallel whenever possible, in order to improve performance. The parallelism is used to improve performance through parallel implementation of various operations such as loading data, building indexes and evaluating queries. One of the first works to propose a parallel physical design for the data warehouse was [Datta et. al, 1998]. In their work they suggest a vertical partitioning of the star schema including algorithms but without quantifying potential gains.

1.1.4. Data Partitioning

Partitioning a large data set across several disks is another way to exploit the I/O bandwidth of the disks by reading and writing them in a parallel fashion. User queries have long been adopted for fragmenting a database in the relational, object-oriented and deductive database models [Ezeife, 1995, Lim and Ng, 1996]. The set of user queries of a database is indicative of how often the database is accessed and of the portion of the accessed database to answer the queries. There are several ways to horizontally partition a relation, namely, *round-robin partitioning*, *hash partitioning*, and *range partitioning*. Partitioning helps in retrieving data faster from the partitions which are much smaller in size than the partitioned table. Horizontal partitioning with respect to time has several additional benefits like ease of maintenance, data purging, and incremental backups. One of the major advantages of data partitioning is that it allows the use of parallel architectures for performing different data warehousing tasks like loading and querying.

1.2. Indexing Techniques for Multi-Dimensional Queries

The indexing methods which can handle the requirements described in section 1.1 are classified in the following categories:

- Multidimensional array-based methods
- Bitmap indexes and their variations
- Hierarchical indexing methods

- Multidimensional Indexes
- Join Indexes

Multidimensional Array-based Methods [Bayer, 1997] are used in OLAP systems which do not adopt the relational approach, storing data in proprietary array structures. Thus, indexing here is closely related with matrix arithmetic. The problem is that, as the multidimensional arrays to be stored are usually sparse, some techniques must be used in order to save space without losing the array model advantages.

Bitmapped Indexes are a good way to handle sparsity, and present some other good features. The bitmap representation gives an alternate method of the row ids (RIDs) representation. The bitmap is simpler and CPU efficient than row ids when the number of distinct values of the indexed column is low. Most relational OLAP vendors use some of its variations. Hierarchical Indexing Methods attempt to index aggregate data in a different way than data which is stored at the finest granularity level [Markl and Bayer, 2000]. The other methods index everything in the same way. Suppose we want to index data aggregated by <product> and by <product, store>. These methods would first build an index on the *product* dimension, and store summaries at the *product* level. Each *product* value contains a separate index at the store level, and stores summaries at the *product-store* level, and so on. Summaries at the *store* level are kept in a separate index on *store*. Their main drawback is the space overhead.

Multidimensional Indexes apply indexing methods originally devised for spatial data structures, mainly *Grid Files* and *R-trees* [Guttmann, 1984].

Join Indexes are methods specifically suited for to perform large table joins, and can be viewed as some kind of pre-computed Join, being references to those rows in two or more tables, which satisfy the join condition [O'Neil and Graefe, 1995].

Traditional Value List Indexes, B-tree indexes are used most commonly in the database systems where we are required to get rows of a table with given values having one or more columns. The leaf level of the B-tree index consists of a sequence of entries for index key values. Each key value reflects the value of the indexed column or columns in one or more rows in the table and each key value entry references the set of rows with that value. Traditionally, Value-List (B-tree) indexes have referenced each row individually as a RID, a Row IDentifier, specifying the disk position of the row. A

sequence of RIDs, known as a RID-list, is held in each distinct key value entry in the B-tree [O'Neil and Quass, 1997].
Assume that C is a column of a table T; then the Projection index on C consists of a stored sequence of column values from C, in order by the row number in T from which the values are extracted. If the column C is 4 bytes in length, then we can fit 1000 values from C on each 4 K Byte disk page (assuming no holes), and continue to do this for successive column values, until we have constructed the Projection index. Now for a given row number n = m(r) in the table, we can access the proper disk page, p, and slot, s, to retrieve the appropriate C value with a simple calculation: p=n/1000 and s = n%1000. Furthermore, given a C value in a given position of the Projection index, we can calculate the row number easily n = 1000*p + s. The columns which are frequently used are held in the projection index which helps in the faster access for such type of columns. They are like cache which stores the more frequently used items.

There are lots of indexing techniques that are in use today; however the right choice of a proper index depends on many parameters such as the cardinality data, distribution, and value range. The read-mostly environment of data warehousing makes it possible to use more complex indexes to speed up queries than in situations where concurrent updates are present.
We need to determine which indexing technique should be built on a Column. A column has its own characteristics which we can use to choose a proper index. These characteristics are given below:

- ***Cardinality data*****:** The cardinality data of a column is the number of distinct values in the column. The efficiency of indexing technique is dependent on degree of cardinality of a column.
- ***Distribution*****:** The distribution of a column is the occurrence frequency of each distinct value of the column. The column distribution helps in determining type of indexing technique to use.
- ***Value range***: The range of values of and indexed column guides us to select an appropriate index type. For example, if the range of a high cardinality column is small, an indexing technique based on bitmap should be used. Without knowing

this information, we might use a B-Tree resulting into system performance degradation.

The following are the characteristics that we have to be concerned with when developing a new indexing technique:

a) The index should be small and utilize space efficiently.

b) The index should be able to operate with other indexes to filter out the records before accessing raw data.

c) The index should support ad hoc and complex queries and speed up join operations.

d) The index should be easy to build (easily dynamically generate), implement and maintain.

In the literature on multi-dimensional index data structures, one will often encounter the words “curse of dimensionality”. In the fields of multi-dimensional access methods these words refer to the degeneration of conventional access methods in multidimensional 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.

1.3. Bitmap Indexes

Bitmap Indexes were first introduced by O’Neil and implemented in the Model 204 DBMS [O’Neil, 1987]. This indexing technique is mostly used for typical data warehouse applications, which are mainly characterized by complex query types and read-mostly environments that are more or less static. In data warehouse environments insert, delete or update operations are not very common and, therefore, it is better to build an index which optimizes the query performance rather than the dynamic features.

Bitmap indexing technology is used by many database vendors to improve performance in query-heavy environments. Bitmap indexing is based on using a single bit (instead of multiple bytes of data) to indicate that a specific column value can be found in a particular row of the database table. The basic bitmap index scheme builds one bitmap for each distinct value of the attribute indexed, and each bitmap has as many bits as the number of tuples. The relative position of the bit within an array of bits (bit map) is used to identify the row of the database table that contains the value in question. Each distinct value for the column being queried requires its own bitmap array. This technology normally provides a smaller, compact structure requiring less system resources to search and process in comparison to a full index. In addition, multiple bitmap indexes can be combined into a single bitmap index utilizing AND/OR logic as dictated by the query search criteria. Being able to leverage and combine existing indexes is crucial in ad hoc query environments where the optimal index is not already available. The query optimizer dynamically builds bitmap indexes as needed to include or eliminate records for selection in the most economical way. Bitmap indexing technology works well with a small number of distinct key values (e.g. state codes), but these indexes can become very large when there are many distinct values for the selected column and many rows in the table.

The simple bitmap index is a collection of number of bitmap vectors for each of the number of distinct value of the indexed column. The basic idea is to use a bit (0 or 1) in a bitmap vector (sequence of bits) to indicate whether an attribute in a table is equal to a specific value or not. The i^{th} bit in a bitmap vector (B_v) is set (1) for a specific value v if and only if the value of indexed column of the record i is v otherwise (0). The number of bitmap vectors in a bitmap index of an attribute is equal to the number of distinct values that the attribute can take. The number of bits in each of the bitmap vectors is equal to the number of records in the table.

RID	*A*	B_a	B_b	B_c	B_d
0	a	1	0	0	0
1	b	0	1	0	0
2	a	1	0	0	0
3	d	0	0	0	1
4	a	1	0	0	0
5	c	0	0	1	0
6	b	0	1	0	0
7	d	0	0	0	1
8	a	1	0	0	0
9	b	0	1	0	0

Figure 1.1: Example of simple bitmap index

Figure 1.1 shows a simple bitmap index on a table with ten rows, where the column A to be indexed has character values ranging from ‘a’ to‘d’. The bitmap index for A column consists of four bitmaps, shown as B_a, B_b, B_c, B_d, with subscripts corresponding to the value represented. In Figure 1.1, the second bit of B_b is 1 because the second row of A has the value 'b', while corresponding bits of B_a, B_c and B_d are all 0.

To answer a query such as "select * from table where A = 'b'", we access B_b and identify the bits equal to 1, corresponding RID represent the result set of the query. Similarly for another query with A='a' or A='b', we perform bitwise OR (|) operations between bitmap vectors B_a and B_b, resulting in a new bitmap from which we identify the sets bits and their corresponding RID's represents the result set of the query. Since bitwise logical operations such as OR (|), AND (&) and NOT (~) are very well-supported by computer hardware, a bitmap index enabled DBMS could evaluate query predicates in extremely fast manner.

Consider following Relation R(age, salary)

Age	***salary***
25	60
22	55
30	70
22	55
23	55
25	100
23	45
30	45

We have following bitmap index:

For Field AGE

22	0 1 0 1 0 0 0 0
23	0 0 0 0 1 0 1 0
25	1 0 0 0 0 1 0 0
30	0 0 1 0 0 0 0 1

For Field SALARY

45	0 0 0 0 0 0 1 1
60	1 0 0 0 0 0 0 0
55	0 1 0 1 1 0 0 0
70	0 0 1 0 0 0 0 0
100	0 0 0 0 0 1 0 0

Consider the following Query:

SELECT *

FROM R

WHERE 23 <= age <= 25

and 50 <= salary <= 70

First, find bitmap index of field "AGE" satisfying the requirement and perform OR Operation:

```
23        0 0 0 0 1 0 1 0
25   OR   1 0 0 0 0 1 0 0
          1 0 0 0 1 1 1 0
```

Second, find bitmap index of field "Salary" satisfying the requirement and perform OR operation

```
60          1 0 0 0 0 0 0 0
55          0 1 0 1 1 0 0 0
70  OR      0 0 1 0 0 0 0 0
            1 1 1 1 1 0 0 0
```

Then, perform AND operation.

```
          1 0 0 0 1 1 1 0
   AND    1 1 1 1 1 0 0 0
          1 0 0 0 1 0 0 0
```

So, the answer to this query is tuple #1 and tuple #5

Bitmaps indexes 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 optimized for typical business applications which are characterized by discrete attribute values. However, scientific data which is mostly characterized by non-discrete attribute values, cannot be handled efficiently by these kind of data structures.

Definition (Simple Bitmap Index)

Given a table $T = \{t_1,...,t_n\}$, where t_j is a tuple of $T(j = 1,...,n)$, let A be an attribute of T, denoted by T.A, and the domain of A be $\{a_a,...,a_m\}$. Then, a simple bitmap index on T.A, B^A, is a set of bitmap vectors $\{B_1,...,B_m\}$, such that $\forall\ B_i(i = 1,...,m), t_j(j = 1,...,n), \ni B_i[j] = 1$, if $t_j.A = a_i$, else $B_i[j] = 0$, where $B_i[j]$ denotes the j-th bit of B_i

Bitmap indexes are widely used in data warehousing environments. The environments typically have large amounts of data and ad hoc queries, but a low level of concurrent DML transactions. For such applications, bitmap indexing provides:

- Reduced response time for large classes of ad hoc queries
- Reduced storage requirements compared to other indexing techniques
- Dramatic performance gains even on hardware with a relatively small number of CPUs or a small amount of memory
- Efficient maintenance during parallel DML and loads

Good properties of Bitmap indexes are cooperativity of different bitmap vectors, low cost of construction, maintenance and processing. Bitmap indexing has been successfully applied to scientific databases by exploiting the fact that scientific data are enumerated or numerical.

1.3.1. Advantages and Challenges of Using Bitmap Indexes

The main advantage of bitmap indexes is that logical operations are very well supported by hardware and, thus, the operations are executed quite fast. The cost for constructing bitmap indexes as well as the processing costs is also very low. Another advantage of bitmap indexes is that they are highly amenable to compression. However, simple bitmap indexes are only efficient for attributes with a low number of distinct values. In other words, if the cardinality of the indexed attribute is low, less number of bit vectors are 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. According to the conventional wisdom, bitmap indexes are only efficient for low-cardinality attributes. However, bitmap indexes with proper encoding, compression, and binning techniques can be made efficient even for high-cardinality attributes.

Bitmap indexes offer several advantages over conventional indexing structures in the read-mostly data warehouse and scientific database environments. Despite of these advantages, bitmap indexes pose several challenges as described below:

1.3.1.1. Space Complexity

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:

$$\text{size of bitmap} = \frac{|T| \times |A|}{8}$$

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:

$$\text{size of B+-tree} = \frac{1.44 \times |T|}{M} \times p$$

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.

1.3.1.2. Time Complexity

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

$$O(|T| \times |A|)$$

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

$$O(|T| \times \log_{\frac{M}{2}}(|A|)) + O(|T| \times \log_2(\frac{p}{4}))$$

where p is the page size and 4 bytes is the size of RID. 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.

1.4. Summary and Outline

Indexes are used to speed up the evaluation of selection conditions followed by the retrieval of desired data. If no pipelining or parallelism is applied, the query response time can be expressed by the sum of the time of index processing plus the time of data retrieval. If the selectivity of a query, which is defined as the ratio of the cardinality of the final result to that of the base table, is high, the time of data retrieval may close in on the time of a costly table scan. For example, for selectivity about 35%, over 99.8% data pages of the underlying table will be hit. For such cases, using indexes has negative effects on query performance. Even for low selectivities, if the time of index processing is high, the total time spent on index processing and data retrieval may be longer than that of a table scan. Query optimization techniques reduce the index processing time, contribute to a better query performance at low selectivities, and also extend the feasibility of bitmap indexes at medium selectivities.

The bitmap representation is an alternate method of the row ids representation. Bitmap index offers better space and time complexity as compared to row id implementation for low cardinality attributes. The indexes improve complex query performance by applying low-cost Boolean operations such as OR, AND, and NOT in the selection predicate on multiple indexes at one time to reduce search space before going to the primary source data. Many techniques have been applied on bitmap indexes, aiming to reduce space requirement as well as improve query performance. Bitmap indexes are primarily intended for data warehousing applications where users query the data rather than update it. Bitmap indexes are most effective for queries that contain multiple conditions in the WHERE clause. In ad hoc queries and similar situations, bitmap indexes can dramatically improve query performance. AND and OR conditions in the WHERE clause of a query can be resolved quickly by performing the corresponding Boolean operations directly on the bitmaps before converting the resulting bitmap to row ids. If the resulting number of rows is small, the query can be answered quickly without resorting to a full table scan. Rows that satisfy some, but not all, conditions are filtered out before the table itself is accessed. This improves response time, often dramatically. The advantages of using bitmap indexes are greatest for columns in which the ratio of the number of distinct values to the number of rows in the table is under 1%. We refer to this ratio as the degree of cardinality**.** A gender column which has only two distinct values (male and female) is ideal for a bitmap index.

The space requirement of a simple bitmap index is a linear function of the cardinality of the indexed attribute and of the indexed table, and the index processing time for a single value selection is a linear function of the length of bitmaps. The sparsity of the bit vectors increases with the cardinality resulting in poor space utilization and high processing cost. Many variations of bitmap indexing have been proposed to solve the sparsity problems. Two common objectives of the proposed methods are (1) reducing the space complexity of the index and (2) improving the performance of index processing. Solutions include compressing bitmaps, e.g., through run-length encoding, and transforming bitmap representation to tuple-id lists. Although these two methods are quite efficient in reducing the space requirements of bitmap indexes, they sacrifice the advantages of bitmap indexing in query processing namely, the low-cost bitwise operations in index processing and the capability of multiple index scans. The size of bitmap index can be very large for a high cardinality attribute where there are thousands or even millions of distinct values. Many strategies have been devised to reduce the index sizes, such as, more compact encoding strategies, binning and compression. Binary and other index types used for high cardinality data: The use of value-based bitmaps to improve performance is not new, early database systems, such as Model 204, employed similar techniques. The problem with early bitmap indexing approaches, however, was that as the number of records in the database grew, so usually did the number of unique data values in a column (the cardinality). Once the cardinality grew beyond a certain point, the benefits of bitmap indexing were diminished because of the amount of disk space required to maintain the bitmap. Bitmaps are also not well suited for joining tables or aggregating data. In addition to above mentioned issues , some of the unresolved issues in traditional database products are index compression (which raises the threshold at which high cardinality becomes an issue); support for the low-cardinality value-based bitmaps found in earlier systems, and support for high-cardinality indexes that can represent both numeric and textual information in binary form . High-cardinality binary indexes can be used in conjunction with low-cardinality, value-based bitmap indexes. Records can be filtered by performing logical AND/OR operations on the arrays of bits.

There are various indexing techniques to achieve our objective of which we mainly concentrate on bitmap indexing techniques. Detailed study and implementation of the various existing and proposed bitmap indexing techniques and different optimization algorithms have been carried out. The work presented in this thesis is based on file implementation rather than a commercial database or data ware house. The code for file based implementations is given for reference in Appendix A. We also discuss techniques which can specifically help in improving query response related to scientific data. Various parameters that are to be considered for building and efficient indexing technique and also the desirable characteristics of such indexes are also given in detail.

The rest of this thesis is organized as follows. Chapter 2 highlights complete related work and necessary background, and gives a summary of current techniques to improve the performance of bitmap indexes. Chapter 3 presents our solution as a new strategy to improve the performance of compression algorithms. We also discuss a new encoding technique which reduces the space requirements for bitmap indexes to a very large extent. Chapter 3 also gives an experimental evaluation of our new compression strategy and encoding technique. In Chapter 4, we present our solution to Tuple Reordering Problem with Multi-Component Indexes and Data Reorganization. Our experimental results with real scientific databases have shown improvement in the compression ratio. Chapter 5 describes our new binning strategies and provides algorithms to solve different types of queries. Overlapping Binning Strategy proposed by us out performs all existing strategies and improves query performance to a large extent. We conclude in Chapter 6 by summary and some recommendations for future work.

Parts of this thesis have been published in conferences and journals. Improved Compression Strategy has been published in International Conference on Information Technology by IEEE Computer Society. New Approach to address complexity issues with bitmap indexes has been published in International Conference proceeding published by Springer. Multi-Component Encoding with Data Reorganization work has published in Information Technology Journal by Asian Network of Scientific Information.

Chapter 2: Bitmap Indexing - Literature Review

2.1. Introduction

Querying large data sets to locate some selected records is a common task in data warehousing applications. However, answering these queries efficiently is often difficult due to the complex nature of both the data and the queries. The most straightforward way of evaluating a query is to sequentially scan all data records to determine whether each record satisfies the specified conditions. A typical query condition is as follows: "Count the number of cars sold by producer *P* in the time interval *T*". This search procedure could usually be accelerated by *indexes*, such as variations of *B-Trees* or *kd-Trees* [Comer, 1979, Gaede & Guenther, 1998]. Generally, as the number of attributes in a data set increases, the number of possible indexes combinations increases as well. To answer multi-dimensional queries efficiently, one faces a difficult choice. One possibility is to construct a separate index for each combination of attributes, which requires an impractical amount of space. Another possibility is to choose one of the multi-dimensional indexes, which is only efficient for some of the queries. In the literature, this dilemma is often referred to as the *curse of dimensionality* [Berchtold et al., 1998, Keim et al., 1999].

By far the most commonly used indexing method is the *B-Tree* [Comer, 1979]. Almost every database product has a version thereof since it is very effective for on-line transaction processing. This type of *tree-based indexing method* has nearly the same operational complexities for *searching* and *updating* the indexes. This parity is important for OLTP because searching and updating are performed with nearly the same frequencies. However, for most data warehousing applications such as on-line analytical processing, the searching operations are typically performed with a much higher frequency than that of updating operations [Chaudhuri, 1997, 2001]. This suggests that the indexing methods for OLAP must put more emphasis on searching than on updating. Among the indexing methods known in the literature, the bitmap index has the best balance between searching and updating for OLAP operations. Frequently, in OLAP operations each query involves a number of attributes. Furthermore, each new query often involves a different set of attributes than the previous one. Using a typical multi-dimensional indexing method, a separate index is required for nearly every combination of attributes [Gaede and Guenther, 1998]. It is easy to see that the number of indexes grows exponentially with the number of attributes in a data set. For data sets with a moderate number of dimensions, a common way to cure this problem is to use one of the multi-dimensional indexing methods, such as *R-Trees* or *kd-trees*. These approaches have two notable shortcomings. Firstly, they are effective only for data sets with *modest number of dimensions*, say, < 15. Secondly, they are only efficient for *queries involving all indexed attributes*.

Bitmap indexing scheme of one kind or another have appeared in all major commercial database systems. This is a strong indication that the bitmap index technology is indeed efficient and practical. The basic bitmap index scheme builds one bitmap for each distinct value of the attribute indexed, and each bitmap has as many bits as the number of tuples. The size of this index can be very large for a high cardinality attribute where there are thousands or even millions of distinct values. The earlier forms of bitmap indexes were commonly used to implement inverted files [Knuth 1998, Wong et al., 1985]. They were first implemented in a commercial DBMS called Model 204 [O'Neil, 1987]. Improvements on this approach were discussed in [O'Neil and Quass, 1997]. The *basic bitmap index* uses each distinct value of the indexed attribute as a key, and generates one bitmap containing as many bits as the number of records in the data set for each key. The *attribute cardinality* is defined as the number of distinct values present in a data set. The size of a basic bitmap index is relatively small for low-cardinality attributes, such as "gender," "types of cars sold per month," or "airplane models produced by Airbus and Boeing." However, for high-cardinality attributes such as "temperature values in a supernova explosion," the index sizes may be too large to be of any practical use. In the literature, there are three basic strategies to reduce the sizes of bitmap indexes: (1) using more complex bitmap ***encoding*** methods to reduce the number of bitmaps or improve query efficiency [Chan and Ioannidis, 1998, 1999, O'Neil and Quass, 1997, Wong et al. 1985], (2) ***compressing*** each individual bitmap compression [Antoshenkov, 1994, Antoshenkov and Ziauddin, 1996, Wu et al., 2001, 2002], and (3) using ***binning*** or other mapping strategies to reduce the number of keys [Shoshani et al., 1999,Stockinger et al., 2002, Wu and Yu, 1996,Wu and Buchmann ,1998]. In the remaining discussions, we refer to these three strategies as encoding, compression and binning, for short. Bitmap indexes are used for speeding up complex, multidimensional queries for On-Line Analytical Processing and data warehouse [Chaudhuri and Dayal, 1997] as well as for scientific applications [Stockinger et al., 2004].However, in many applications only *some of the attributes* are used in the queries.

In many data warehouse applications, bitmap indexes perform better than tree-based schemes, such as the variants of B-tree or R-tree [Jurgens and Lenz, 1999, Chan and Ioannidis, 1998, O'Neil, 1987, Wu and Buchmann, 1998]. According to the performance model proposed by Jurgens and Lenz [1999], bitmap indexes are likely to be even more competitive in the future as disk technology improves. In addition to supporting queries on a single table as shown in this article, researchers have also demonstrated that bitmap indexes can accelerate complex queries involving multiple tables [O'Neil and Graefe, 1995]. In these cases, the conventional indexing methods are often not efficient. For ad hoc range queries, most of the known indexing methods do not perform better than the projection index [O'Neil & Quass, 1997], which can be viewed as one way to organize the base. The bitmap index, on the other hand, has excellent performance characteristics on these queries. As shown with both theoretical analyses and timing measurements, a compressed bitmap index can be very efficient in answering one-dimensional range queries [Stockinger et al., 2002, Wu et al., 2004, Wu et al., 2006]. Since answers to one-dimensional range queries can be efficiently combined to answer arbitrary multi-dimensional range queries, compressed bitmap indexes are efficient for any range query. In terms of computational complexity, one type of compressed bitmap index was shown to be theoretically optimal for one-dimensional range queries. The reason for the theoretically proven optimality is that the query response time is a linear function of the number of hits, i.e. the size of the result set. There are a number of indexing methods, including B^**-tree* and B^+*-tree* [Comer, 1979], that are theoretically optimal for one-dimensional range queries, but most of them cannot be used to efficiently answer arbitrary multi-dimensional range queries. The bitmap index in its various forms was used a long time before relational database systems or data warehousing systems were developed. Earlier on, the bitmap index was regarded as a special form of *inverted* *files* [Knuth, 1998]. The *bit-transposed file* [Wong et al., 1985] is very close to the bitmap index currently in use. The name *bitmap index* was popularized by O'Neil and colleagues [O'Neil, 1987, O'Neil & Quass, 1997]. Following the example set in the description of *Model 204*, many researchers describe bitmap indexes as a variation of the B-tree index. To respect its earlier incarnation as inverted files, a bitmap index may be regarded as a data structure consisting of keys and bitmaps. Moreover, B-tree can be looked as a way to layout the keys and bitmaps in files. Since most commercial implementations of bitmap indexes come after the product already contains an implementation of a B-tree, it is only natural for those products to take advantage of the existing B-tree software. For new developments and experimental or research codes, there is no need to couple a bitmap index with a B-tree. For example, in a research program that implements many of the bitmap indexing methods discussed later in this chapter [FastBit, 2005], the keys and the bitmaps are organized as simple arrays in a binary file. This arrangement was found to be more efficient than implementing bitmap indexes in B-trees or as layers on top of a DBMS [Stockinger et al. 2002, Wu et al. 2002].

In [Chan and Ioannidis 1998, 1999] the following bitmap encoding strategies are introduced: equality, range and interval encoding. Equality encoding is optimized for so-called exact match queries of the form $a = v$ where a is an attribute and v the value to be searched for. Range encoding, on the other hand, is optimized for one-sided range queries of the from a op v where op in {<, <=,>,>=}. Finally, interval encoding shows the best performance characteristics for two sided-range queries of the form v_1 op a op v_2. Wu and Buchmann [1998] represented attribute values in binary form that yields indexes with only $\log_2 |A|$ bitmaps, where |A| is the attribute cardinality. The advantage of this encoding scheme is that the storage overhead is even smaller than for interval encoding. However, in most cases query processing is more efficient with interval encoding since in the worst case only two bitmaps need to be read whereas with binary encoding always all bitmaps have to be read. As already mentioned before, simple bitmap indexes are efficient for low-cardinality attributes but they show a considerable storage overhead for high-cardinality attributes. One way of reducing the storage complexity is to use bitmap compression. An efficient bitmap compression scheme not only has to reduce the size of bitmaps but also has to perform bitwise Boolean operations efficiently.

Various bitmap compression schemes have been discussed in [Johnson, 1999, Amer-Yahia and Johnson, 2000]. The authors demonstrated that the scheme named Byte aligned Bitmap Code (BBC) [Antoshenkov, 1994, Antoshenkov and Ziauddin, 1996] shows the best overall performance characteristics. More recently a new compression scheme called Word-Aligned Hybrid (WAH) [Wu et al., 2004] was introduced. This compression algorithm significantly reduces the overall query processing time compared to BBC. The key reason for the efficiency of WAH is that it uses a much simpler compression algorithm. A number of empirical studies have shown that WAH compressed bitmap indexes answer queries faster than uncompressed bitmap indexes, projection indexes, and B-tree indexes, on both high and low-cardinality attributes [Wu et al,. 2001, 2002, 2004, Stockinger et al. 2002]. The authors complement the observations with rigorous analyses. Their main conclusion includes that the WAH compressed bitmap index is in fact optimal. Some of the most efficient indexing schemes such as B^+- tree indexes and B*-tree indexes have a similar optimality property [Comer, 1979, Knuth, 1998]. However, a unique advantage of compressed bitmap indexes is that the results of one-dimensional queries can be efficiently combined to answer multidimensional queries. This makes WAH compressed bitmap indexes well suited for ad hoc analyses of large high-dimensional datasets.

To compress a bitmap, a simple option is to use a text compression scheme, such as LZ77 (used in gzip) [Gailly and Adler, 1998, Ziv and Lempel 1977]. These schemes are efficient in reducing file sizes. However, performing logical operations on the compressed bitmaps is usually much slower than on the uncompressed bitmaps, since the compressed bitmaps have to be explicitly uncompressed before any operation. To illustrate the importance of efficient logical operations, assume that the attribute NumParticles can have integer values from 1 to 10,000. Its bitmap index would have 10,000 bitmaps. To answer a query involving "NumParticles > 5000," 5000 bitmaps have to be ORed together. To efficiently answer this query, it is not sufficient that the bitmaps are small; the operations on them must be fast as well. To improve the performance of bitwise logical operations, a number of specialized schemes have been proposed. Johnson and colleagues have thoroughly studied many of these schemes [Johnson, 1999, Amer-Yahia and Johnson, 2000]. From their studies we know that the logical operations with these specialized schemes are usually faster than those with LZ77. One such specialized scheme, called the *Byte-Aligned Bitmap Code* (BBC), is especially efficient [Antoshenkov, 1994, Antoshenkov and Ziauddin, 1996]. However, in the worst case, the total time required to perform a logical operation on two BBC compressed bitmaps can still be 100 times longer than on two uncompressed bitmaps. A number of compression schemes that improve the overall query response time by improving their worst-case performance have been discussed [Wu et al. 2001]. In this article, the main concentration was on the Word-Aligned Hybrid (WAH) code for two main reasons: (1) it is the easiest to analyze, which leads them to prove an important optimality about the compressed bitmap indexes, and (2) it is the fastest in their tests. In earlier tests, the authors observed that bitwise logical operations on WAH compressed bitmaps are 2 to 100 times faster than the same operations on BBC compressed bitmaps because WAH is a much simpler compression method than BBC [Wu et al., 2001, Stockinger et al., 2002].

There is a space-time tradeoff among these compression schemes. Comparing BBC with LZ77, BBC trades some space for more efficient operations. Similarly, WAH trades even more space for even faster operations. Compressing individual bitmaps is only one way to reduce the bitmap index size. An alternative strategy is to reduce the number of bitmaps, for example, by using binning or more complex encoding schemes. With binning, multiple values are grouped into a single bin and only the bins are indexed [Koudas, 2000, Shoshani et al., 1999, Wu and Yu, 1996]. Many researchers have studied the strategy of using different encoding schemes to reduce the index sizes [Chan and Ioannidis, 1998, 1999, O'Neil and Quass, 1997, Wong et al., 1985, Wu and Buchmann, 1998]. One well-known scheme is the bit-sliced index that encodes c distinct values using log_2c bits and creates a bitmap for each binary digit [O'Neil and Quass, 1997]. This is referred to as the *binary encoding scheme* elsewhere [Wong et al., 1985, Chan and Ioannidis, 1998, Wu and Buchmann, 1998]. A drawback of this scheme is that most of the bitmaps have to be accessed when answering a query. To answer a two-sided range query such as "120 < Energy < 140," most bitmaps have to be accessed twice. There are also a number of schemes that generate more bitmaps than the bit-sliced index but access fewer of them while processing a query, for example, the attribute value decomposition [Chan and Ioannidis, 1998], interval encoding [Chan and Ioannidis, 1999], and the K-of-

N encoding [Wong et al., 1985]. In all these schemes, an efficient compression scheme should improve their effectiveness. Additionally, a number of common indexing schemes such as the signature file [Furuse et al., 1995, Ishikawa et al. 1993, Lee et al., 1995] may also benefit from an efficient bitmap compression scheme. Compressed bitmaps can also be effective for purposes other than indexing. In one case, the authors demonstrated that using compressed bitmaps significantly speeds up the tracking of spatial features as they evolve in the simulation of a combustion process [Wu et al., 2003].

The bitmap indexes discussed so far encode each distinct attribute value as one bitmap vector. This technique is very efficient for integer or floating point values with low attribute cardinalities. However, scientific data is often based on floating point values with high attribute cardinalities. The work presented in [Stockinger et al., 2004] demonstrated that bitmap indexes with binning can significantly speed up multi-dimensional queries against high-cardinality attributes. A further bitmap index with binning called range-based bitmap indexing was introduced in [Wu and Yu, 1996]. The idea is to evenly distribute skewed attribute values onto various bins in order to achieve uniform search times for different queries. The bin ranges for the bitmap vectors are chosen based on a dynamic bucket expansion and contraction approach. In the initial phase, the number of entries per bucket (bin) is counted. If the number of entries per bucket grows beyond a certain threshold, the bucket is dynamically expanded into buckets with smaller ranges. Finally, multiple buckets with adjacent ranges are combined. The authors demonstrated that the algorithm efficiently redistributes highly skewed data. However, performance results about query response times were not discussed.

A methodology for building space efficient bitmap indexes is introduced for high-cardinality attributes based on binning [Koudas, 2000]. The work in [Koudas, 2000] focuses on point (equality) queries rather than range queries. An optimal dynamic programming algorithm is used for efficiently choosing bin ranges. The author raised several interesting open research issues that inspired our research, like extending the work by analyzing range queries.

2.1.1. Bitmap Indexes in Commercial Systems

The first commercial product to use the name *bitmap index* is *Model 204*. O'Neil has published a description of the indexing method in [O'Neil, 1987]. Model 204 implements the basic bitmap index. It has no binning or compression. Currently, Model 204 is marketed by Computer Corporation of America. ORACLE has a version of compressed bitmap indexes in its flagship product since version 7.3. They implemented a proprietary compression method. Based on the observed performance characteristics, it appears to use *equality encoding* without binning.

Sybase IQ implements the bit-sliced index [O'Neil & Quass, 1997]. Using the terminology defined, Sybase IQ supports unbinned, binary encoded, uncompressed bitmap indexes. In addition, it also has the basic bitmap index for low-cardinality attributes. IBM DB2 implements a variation of the binary encoded bitmap index called *Encode Vector Index*. IBM Informix products also contain some versions of bitmap indexes for queries involving one or more tables. These indexes are specifically designed to speed up join-operations and are commonly referred to as *join indexes* [O'Neil and Quass, 1997]. InterSystems Corp's Cache also has bitmap index support since version 5.0. Even though we do not have technical details on most of these commercial products, it is generally clear that they tend to use either the basic bitmap index or the bit-sliced index.

2.2. Encoding Techniques

The limitations of simple bitmap indexing (SBI) for high cardinality attributes lead to the suggestion of encoded bitmap indexing which provides the advantage of a drastic reduction in space requirements and also a corresponding performance gain. The main idea of encoded bitmap indexing (EBI) is to encode the attribute domain. We will see the following example: We assume that we have a fact table 'SALES' with N tuples and a dimension table 'PRODUCT' with 12,000 different products. If we build a simple bitmap index on 'PRODUCT', it will require 12,000 bitmap vectors of N bits in length. However, if we use encoded bitmap indexing we only need $\lceil \log_2 12000 \rceil = 14$ bitmap vectors plus a mapping table which is a very significant reduction of the space complexity.

2.2.1. Huffman Encoding

[Wu and Buchmann, 1998] proposes query optimization strategies for selections using bitmaps. Both static and dynamic query optimization strategies have been discussed with both continuous and discrete selection criteria. Static optimization strategies discussed are the optimal design of bitmaps, and algorithms based on tree and logical reduction. The dynamic optimization discussed is the approach of inclusion and exclusion for both bit-sliced indexes and encoded bitmap indexes. Figure 2.1 explains with an example how Huffman encoding used for reducing the space complexity of bitmap indexes: We assume that our attribute domain is given by the table T is {a,b,c}. The encoding schema of EBI is stored in a separate table called mapping table and simply encodes the values from a SBI by means of Huffman encoding and therefore 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 SBI but in the table of EBI the attribute value 'A' is encoded as 00.

….	*column*	….	B_a	B_b	B_c	B_1	B_0	A	00
	A		1	0	0	0	0	B	01
	B		0	1	0	0	1	C	10
	C		0	0	1	1	0		
	B		0	1	0	0	1		
	A		1	0	0	0	0		

Table T SBI EBI Mapping Table

Figure 2.1: Huffman Encoded Bitmap Index

Figure 2.2 gives an example of a Value-List index with 12 records, where each column (except the first) represents a bitmap B associated with an attribute value *v*. The column $\pi_A(R)$ represents the relation of the attribute values present in the records.

	$\pi_A(R)$	B^8	B^7	B^6	B^5	B^4	B^3	B^2	B^1	B^0
1	3	0	0	0	0	0	1	0	0	0
2	2	0	0	0	0	0	0	1	0	0
3	1	0	0	0	0	0	0	0	1	0
4	2	0	0	0	0	0	0	1	0	0
5	8	1	0	0	0	0	0	0	0	0
6	2	0	0	0	0	0	0	1	0	0
7	2	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	0	1
9	7	0	1	0	0	0	0	0	0	0
10	5	0	0	0	1	0	0	0	0	0
11	6	0	0	1	0	0	0	0	0	0
12	4	0	0	0	0	1	0	0	0	0

Figure 2.2: Example of a Value-List Index

2.2.2. Multi-Component Encoding

Many researchers have proposed strategies to find the balance between the space and time requirements [Wong et al., 1985, Chan and Ioannidis, 1999]. A method proposed by [Chan and Ioannidis, 1999] called *multi-component encoding* can be thought of as a generalization of *binary encoding*. In the binary encoding, each bitmap represents a binary digit of the attribute values; the multi-component encoding breaks the values in a more general way, where each component could have a different size. Consider an integer attribute with values ranging from 0 to *c*-1. Let b_1 and b_2 be the sizes of two components c_1 and c_2, *where* $b_1*b_2>=c$. Any value v can be expressed as $v = c_1*b_2+c_2$, where $c_1 = v / b_2$ and $c_2 = v \% b_2$, where '/' denotes the integer division and '%' denotes the modulus operation. One can use a simple bitmap encoding method to encode the values of c_1 and c_2 separately. Next, we give a more specific example to illustrate the multi-component encoding.

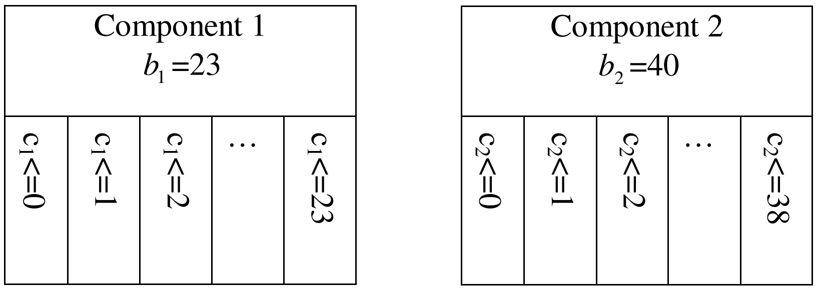


Figure 2.3: An illustration of a 2-component bitmap index.

Figure 2.3 illustrates a 2-component encoded bitmap index for an attribute with cardinality c=1000. In our example, the two components have base sizes of b_1=25 and b_2=40. Assume the attribute values are in the domain of [0; 999]. An attribute value v is decomposed into two components with $c_1 = v / 40$ and $c_2 = v \% 40$. The component c_1 can be treated as an integer attribute in the range of 0 and 24; the component c_2 can be viewed as an integer attribute in the range of 0 and 39. Two bitmap indexes can be built, one for each component, for example, c_1 with the equality encoding and c_2 with *range enco*ding. If *range encoding* is used for both components, it uses 24 bitmaps for Component 1 and 39 bitmaps for Component 2. In this case, the 2-component encoding uses 63 bitmaps, which is more than the 10 bitmaps used by *binary encoding*. To answer the same query "v < 105" using the 2-component index, the query is effectively translated to "c_1<2 OR (c_1=2 AND c_2<25)." Evaluating this expression requires three bitmaps representing "c_1<=1," "c_1<=2," and "c_2<=24." In contrast, using the binary encoded bitmap index to evaluate the same query, all 10 bitmaps are needed.

2.2.3. Equality and Range Encoding

Consider the i^{th} component of an index with attribute cardinality b_i. There are essentially two major schemes to directly encode the corresponding values v_i $(0 \le b_i \le b_i - 1)$ in bits:

Equality Encoding: There are b_i bits, one for each possible value. The representation of value v_i has all bits set to 0, except for the bit corresponding to v_i, which is set to 1. Clearly, an equality-encoded component consists of b_i bitmaps.

Range Encoding: There are b_i bits again, one for each possible value. The representation of value v_i has the v_i rightmost bits set to 0 and the remaining bits (starting from the one corresponding to v_i and to the left) set to 1. Intuitively, each bitmap $B_i^{v_i}$ has 1 in all the records whose i^{th} component value is less than or equal to v_i. Since the bitmap $B_i^{b_i-1}$ has all bits set to 1, it does not need to be stored, so a range- encoded component consists of (b_i - 1) bitmaps.

Figures 2.4(b) and (c) show the range-encoded indexes corresponding to the equality-encoded indexes in figure 2.2.

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

(a) **(b)**

Figure 2.4: Examples of range-encoded bitmap indexes. (a) Projection of indexed attribute values with duplicates preserved. (b) Single Component, base-9, range-encoded bitmap index.

B_2^1	B_2^0	B_1^1	B_1^0
1	0	1	0
1	1	1	1
1	1	1	1
1	1	1	1
0	0	0	0
1	1	1	1
1	1	1	1
1	1	1	1
0	0	0	0
1	0	1	0
0	0	0	0
1	0	1	0

(c)

Figure 2.4: (c) Base< 3, 3 > range-encoded bitmap index.

An encoded bitmap index on a column A of a table T consists of a set of bitmap vectors, lookup table and a set of retrieval Boolean function. Each distinct value of a

column is encoded using a no. of bits each of which is stored in a bitmap vector. Lookup table stores the mapping between A (column) and its encoded representation.
As of now, the only encoding scheme that we used is the one used for Value- List indexes, called Equality Encoding. But there is another scheme that actually allows for even more compact indexes: the Range Encoding scheme. With this scheme, we not only set the bit to one that contains the specific value, but also all other bits that are left of this value, in short: all bits in the bitmaps B^i to the desired value B^{v_i}. This way, we need one less column in every bitmap, because the rightmost column has then all bits set to 1 and does not need to be stored. Figure 2.4(a) and 2.4(b) shows the bitmap from the first example, now reorganized to use range encoding. Of course this type of encoding can also be used with multi-component bitmaps. This is even more effective as every single component now needs one less column to be stored, resulting in an even more compact index. As we can see in Figure 2.4(c), the index from the initial example has been reduced from 9 to only 4 columns, i.e. to less than half the original number. The downside of this method is that we can't use simple bit comparison anymore to get the desired value, because now not only the specific value has 1 in its index but also all bitmaps left of it. To overcome this disadvantage, the evaluation algorithm has to be modified. The general idea is to differ between the possible query operators $(<,\leq,>,\geq,=,\neq)$and compute each in a different way.

A key design parameter for bitmap indexes is the encoding scheme, which determines the set of attribute values “represented” by each bitmap in an index that is the attribute values that set the corresponding records’ bits in a bitmap to 1. For example, the simple design mentioned earlier, the encoding scheme is such that the bitmap associated with the value v represents v alone. Previous studies [Wong et. al., 1985, O’Neil and Quass, 1997, and Chan and Ioannidis 1999] have identified two basic encoding schemes: Equality Encoding, which is the one mentioned above and is efficient for equality queries (i.e. , queries of the form $"A = v"$), and Range Encoding, which is efficient for one-sided range queries (i.e., queries of the form $"A \leq v"$ or $"A \geq v"$). However, the space-time performance optimality of either of the encoding schemes is remain an open issue; that is, it is not known whether or not there exists an encoding scheme with strictly better space-time performance than equality encoding for equality queries or range encoding for one-sided range queries. Two sided Range queries of the form "$v_1 \le A \le v_2$" Performance of bitmap indexes for the more general class of membership queries (i.e., queries of the form"$A \in \{v_1, v_2, ..., v_k\}$").

Informally, an encoding scheme S is optimal for a query class Q if there is no other encoding scheme with strictly better space-time performance than S for Q. Consider an attribute A of a relation R, where the attribute cardinality is C. For simplicity and without loss of generality, the domain of A is assumed to be a set of consecutive integers from 0 to C-1. Let B be an individual bitmap of a bitmap index on A. For notational convenience, we overload the symbol B so that it indicates both the bitmap itself (i.e., a sequence of 0's and 1's) and the set of attribute values in A that correspond to its bits that are set to 1. This allows us to use set operators and logical operators interchangeable. The logical operators AND, OR, and XOR are denoted by $\wedge, \vee,$ and $\oplus$, respectively, while the compliment of B is denoted by $\overline{B}$

An interval query on attribute A is a query of the form "$x \le A \le y$" or "$NOT(x \le A \le y)$". An interval query is an equality query if $x = y$; it is a one-sided range query if $x = 0$ or $y = C - 1$; and it is two sided range query if $0 < x < y < C - 1$. A one-sided or two-sided range query is also called a range query. We denote the class of equality queries, one-sided range queries, two-sided range queries, and range queries by EQ, 1RQ, 2RQ and RQ, respectively. We refer to a query that belongs to the query class Q, where $Q \in \{EQ, 1RQ, 2RQ, RQ\}$, as a Q-query. Queries of the form "$A \in \{v_1, v_2, ..., v_k\}$" are membership queries.

Equality encoding is the most fundamental and common bitmap encoding scheme. It consists of C bitmaps $\varepsilon = \{E^0, E^1, ..., E^{C-1}\}$, where each bitmap $E^v = \{v\}$.

Evaluation of interval queries using an equality-encoded bitmap index proceeds as in equation (2.1):

$$"v_1 \leq A \leq v_2"= \begin{cases} \dfrac{\bigvee_{i=v_1}^{v_2} E^i}{\bigvee_{i=0}^{v_1-1} E^i \vee \bigvee_{i=v_2+1}^{C-1} E^i} & \begin{matrix} \text{if } v_2 - v_1 + 1 \leq \left\lfloor \dfrac{C}{2} \right\rfloor, \\ \\ \text{otherwise} \end{matrix} \end{cases} \tag{2.1}$$

The range encoding scheme consists of $(C-1)$ bitmaps $\Re = \{R^0, R^1, ..., R^{C-2}\}$, where each bitmap $R^v = [0, v]$.

$$"v_1 \leq A \leq v_2"= \begin{cases} R^0 & \text{if } v_1 = v_2 = 0, \\ R^{v_1} \oplus R^{v_1-1} & \text{if } 0 < v_1 = v_2 < C-1, \\ \overline{R^{C-2}} & \text{if } v_1 = v_2 = C-1, \\ \overline{R^{v_1-1}} & \text{if } 0 < v_1 < C-1, v_2 = C-1, \\ R^{v_2} & \text{if } v_1 = 0, 0 \leq v_2 < C-1, \\ R^{v_2} \oplus R^{v_1-1} & \text{otherwise.} \end{cases} \tag{2.2}$$

In a binary encoded bitmap index, each attribute value is represented in binary form (i.e., with $\lceil \log_2(C) \rceil$ bits, where C is the attribute cardinality); so there are a total of $\lceil \log_2(C) \rceil$ bitmaps in a binary-encoded index.

Theorem 2.1 states several results for the existing encoding schemes.

Theorem 2.1: The following statements hold:

1. Range encoding is optimal for EQ iff $C \leq 5$.
2. Range encoding is optimal for 1RQ for all C.
3. Range encoding is not optimal for 2RQ for any C.
4. Range encoding is optimal for RQ for all C.
5. Equality encoding is optimal for EQ for all C.
6. Equality encoding is not optimal for 1RQ, 2RQ, and RQ for any C.

Let I denote an n-component index with base $< b_n, b_{n-1}, ..., b_1 >$. Then the space and time usage of the encoding schemes can be computed with the following formulas [Chan and Ioannidis, 1998]:

For Equality Encoding:

$$Space(I) = \sum_{i=1}^{n} s_i \text{, where } s_i = \begin{cases} b_i & : \quad b_i > 2 \\ 1 & : \text{ otherwise} \end{cases} \tag{2.3}$$

$$Time(I) = \frac{1}{3}\sum_{i=1}^{n}(2t_i + 1)\text{, where}$$

$$t_i = \begin{cases} \frac{1}{b_i}\left(\left\lfloor \frac{b_i}{2} \right\rfloor^2 + (b_i - 1)\left(\left\lceil \frac{b_i}{2} \right\rceil - \frac{b_i}{2}\right)\right) & : \quad b_i > 2 \\ 1 & : \text{otherwise} \end{cases} \tag{2.4}$$

For Range Encoding:

$$Space(I) = \sum_{i=1}^{n}(b_i - 1) \tag{2.5}$$

$$Time(I) = 2\left(n - \sum_{i=1}^{n}\frac{1}{b_i} + \frac{1}{3}\left(\frac{1}{b_1} - 1\right)\right) \tag{2.6}$$

The base of the most time-efficient 2-component space-optimal index is given by $< b_2 - \delta, b_1 + \delta >$, where $b_1 = \left\lceil \sqrt{C} \right\rceil, b_2 = \left\lceil \frac{C}{b_1} \right\rceil$, and

$$\delta = \max\left\{0, \left\lfloor \frac{b_2 - b_1 + \sqrt{(b_2 + b_1)^2 - 4C}}{2} \right\rfloor\right\}.$$

The expected number of pages which are hit by selecting k tuples from a table of n pages is computed by

$$n \bullet \left(1 - \prod_{r-1}^{k} \left(pn \bullet \frac{n-1}{n} - r + 1 \right) \Big/ \left(pn - r + 1 \right) \right)$$, where each page contains p tuples. The hit rate depends, of course, highly on the value p, clustering criteria, distribution of indexed attribute, etc.

Consider an example, given are two attributes A and B of a table T. Let the domain of A, denoted by $Dom(A)$, be $\{A \mid 100 \le \mathrm{A} \le 900, \mathrm{A} \in \mathrm{Z}^+\}$ and $Dom(B) = \{a, b, c, d, e, f, t, u, v, w\}$. The cardinality of T is defined by |T| and the cardinality of an attribute is defined by the cardinality of its domain.
An encoding scheme S is optimal for a query class Q if there is no other encoding scheme with strictly better space-time performance than S for Q. An encoding function is called total-order preserving, if there exists a total order in the domain of the indexed attribute, and the same total order still exists in the encoded attribute domain. We use '+' to denote the logical operator OR and '.' to denote the logical operator AND.
A min-term of n Boolean variables is a logical conjunction of all n variables, or their negations. Both $< 5,6,6,6 >$ and $< 2,8,8,8 >$ are well-defined bases. A well-defined base, $< b_n, ..., b_1 >$, consists of finite number of components, i.e., $n \in Z^+$, such that

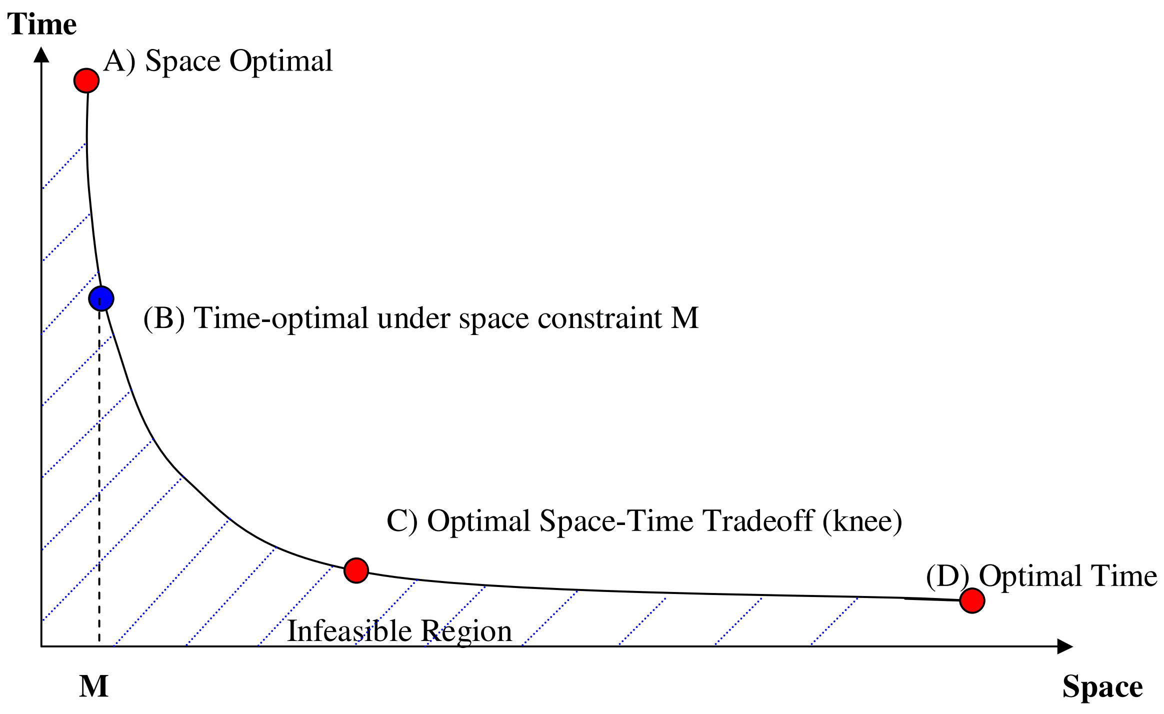
$$b_n = \left\lceil |A| \Big/ \left(\prod_{i=1}^{n-1} b_i \right) \right\rceil.$$

Space-optimum Given an integer n, the space-optimal n-component bit-sliced index is the n-component bit-sliced index with the base $< \overbrace{b-1, ..., b-1}^{n-r}, \overbrace{b, ..., b}^{r} >$,
where b=$\left\lfloor \sqrt[n]{|A|} \right\rfloor$ and r is the smallest positive integer such that $b^r (b-1)^{n-r} \ge |A|$.

Time-**optimum** Given an integer n, the time-optimal n-component bit-sliced index is the index with the base $< \underbrace{2, ..., 2}_{n-1}, \left\lceil \frac{|A|}{2^{n-1}} \right\rceil >$.

2.2.4. Comparison of Bitmap Encoding Schemes

This section compares the space-time tradeoff of the bitmap encoding schemes i.e. the best compromise between space usage and query response time, see figure 2.5.



Time
A) Space Optimal
(B) Time-optimal under space constraint M
C) Optimal Space-Time Tradeoff (knee)
(D) Optimal Time
Infeasible Region
M
Space

Figure 2.5: Space-Time Tradeoff

For multi component indexes, the more space we save with splitting the index into multiple components, the more indexes have to be scanned for a query and so the processing time rises [Chan and Ioannidis 1999]. The results show that range-encoding in general provides better space-time tradeoff than equality-encoding in most cases. This becomes clear as the graph for the range-encoded index most of the time is located under the graph for the equality-encoded index, meaning the time for using it is shorter. Encoded Bitmap indexes solves the problem of sparsity, improves the space utilization, shortens the maintenance and processing time, and also improves the performance of processing range queries. Most of all, the cardinality of the indexed attribute no longer dramatically effects the maintenance and processing costs of the encoded bitmap indexes.

The main encoding schemes for bitmap indexes are as follows:

- Attribute value decomposition
  - Convert attribute value to suitable base
- Range Encoding
  - |A| - 1 bitmaps
- Interval Encoding
  - |A|/2 -1 bitmaps
- Equality encoding
  - |A| bitmaps

|A| means Attribute cardinality. The figure 2.6 explains the differences.

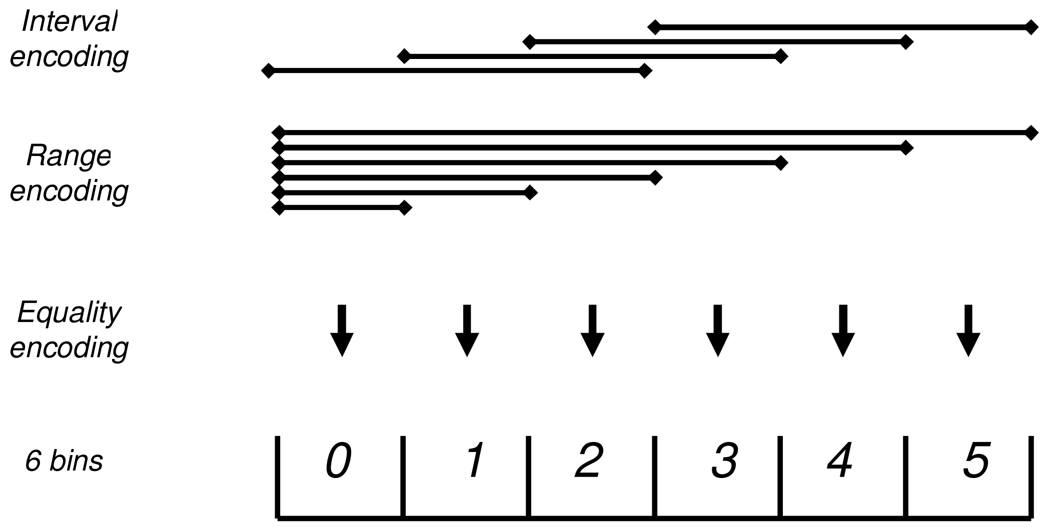


Figure 2.6: Different encoding techniques

2.2.5. Bit-Sliced Encoding

A bit-sliced index of an attribute is a bitwise projection of the attribute [O'Neil and Quass, 1997]. The number of bit vectors is equal to the length of the attribute's data type in bits, and the length of each bit vector is equal to the cardinality of the indexed table. A bit-sliced index is based on converting integer values to binary values in order to perform fast logical operations on them since the hardware directly supports 1's and 0's. We should choose an optimal number of bits per bit-vector in order to represent the whole attribute domain and to occupy minimum space.

RID	**Value C**	**Binary Representation C_2**	**B^7 2^7**	**B^6 2^6**	**B^5 2^5**	**B^4 2^4**	**B^3 2^3**	**B^2 2^2**	**B^1 2^1**	**B^0 2^0**
1	45	00101101	0	0	1	0	1	1	0	1
2	156	10011100	1	0	0	1	1	1	0	0
3	14	00001110	0	0	0	0	1	1	1	0
4	4	00000100	0	0	0	0	0	1	0	0
5	82	01010010	0	1	0	1	0	0	1	0
6	25	00011001	0	0	0	1	1	0	0	1

Table 2.1: An example of a Bit-sliced Index

In the above example, table 2.1, column C represents the actual values and C_2 represents binary form of the value and B^0 to B^7 represents the bit slices. As we look in the index, B^0 represents the bits for the numbers corresponding to the value 2^0 and B^1 represents the bits for the numbers corresponding to the value 2^1 .So B^i represents the bits for the numbers corresponding to the value 2^i where i is from 0 to m (maximum number of bit slices required is m+1).In the above example m is 6 since the largest number 156 can be represented in 7 bits. So the number of bit slices required is also 7.

For example, 23 can be written as the sum of 1.2^0 +1. 2^1 $+1.2^2$ $+0.2^3$ $+1.2^4+0.2^5+0.2^6+0.2^7$. So for the number 20 B^0 contains the value corresponding to 2^0 i.e. 0 and B^1 contains 0 and B^2 contains 1 etc. Formal definition of bit-sliced index is as follows:

Definition: A bit-sliced index B, often referred to as a *BSI,* is an ordered list of bitmaps B^S, B^{S-1}. . . B^1, B^0; the list of bitmaps is used to represent values of some column C. The bitmaps B^S, B^{S-1}, . . . , B^1, B^0 are called the bit-slices, and provide binary representations of C values for all the rows, B^0 holds the 1's bits, B^1 holds the 2's bits, B^2 holds the 4's bits, etc. More precisely, if we represent the C value of row j by C[j], and the bit for row j in bit-slice B^i by $B^i[j]$, then the values for $B^i[j]$ are chosen such that $C[j]= \sum^{s}_{i=0} B^i[j].2^i$ .So as mentioned in the previous example $23 = 1.2^0+1.2^1 +1.2^2 +0.2^3+1.2^4+0.2^5 +0.2^6+0.2^7$ where 23 represents C [4].

2.3. Compression Techniques

Wu et al., 2000, proposed to improve the effectiveness of the basic bitmap index by compression. Other ways of improving the bitmap index include binning and using different encoding. With binning, multiple values are grouped into a single bin and only the bins are indexed [Koudas, 2000, Shoshani, 1999, Wu and Yu, 1996]. This strategy reduces the number of bitmaps used but it also introduces inaccuracies. In order to accurately answer a query, one has to scan some of the attribute values after operating on the indexes. Many researchers have studied the strategy of using different encoding schemes [Chan and Ioannidis, 1998, 1999, Wong et al., 1985, Wu and Buchmann, 1998]. One well known scheme is the bit-sliced index, that encodes k distinct values using $\log_2 k$ bits and creates a bitmap for each binary digit [O'Neil and Quass, 1997]. This is related to the binary encoding scheme discussed earlier. A drawback of encoding scheme is that to answer each query, most of the bitmaps have to be accessed, and possibly multiple times. There are also a number of schemes that generate more bitmaps than the bit-sliced index but access less of them while processing a query, for examples, the attribute value decomposition [Chan and Ioannidis, 1998], interval encoding [Chan and Ioannidis, 1999] and the K-of-N encoding [Wong et al.,1985]. Compression techniques can be applied on any bitmap.

Once we have identified some efficient compression schemes, we can improve all bitmap indexes. Additionally, a number of other common indexing schemes such as the signature file [Furuse et al., 1995, Ishikawa et al., 1993, Lee, 1995] and the bit transposed files [Wong et al., 1985] may also benefit from efficient bitmap compression algorithms. Other high-dimensional indexing schemes yet to be mentioned include the projection index [O'Neil and Quass, 1997] and the UB-tree [Bayer, 1997, Markl and Bayer, 2000]. The projection index can be viewed as a different way of organizing the attribute values of a table. It can be implemented easily and efficiently by using bitmaps to store the intermediate results, and we use it as the bases for measuring the performance of our compressed bitmap index. The UB-Tree is a promising technique.

To address the performance issue, a number of special algorithms have been proposed. Johnson and colleagues have conducted extensive studies on their performances [Jhonson, 1999, Amer-Yahia and Jhonson, 2000,]. From their studies, we know that the logical operations using these specialized schemes are usually faster than those using gzip. One such specialized algorithm, called the *Byte-aligned Bitmap Code* (BBC), is known to be very efficient. It is used in a commercial database system, ORACLE [Antoshenkov, 1994, Antoshenkov and Ziauddin, 1996]. However, even with BBC, in many cases logical operations on the compressed data still can be orders of magnitudes slower than on the uncompressed data.

2.3.1. Byte aligned Bitmap Code (BBC)

Normal bitmap indexing technique doesn't compress the runs of zero and ones. As a result it is not space efficient. To overcome this disadvantage, BBC was introduced [Antoshenkov, 1994]. BBC is designed so that it can compress runs which are both short and long. As a result we can get better compression ratio (on a normal data). It is based on the basic idea of run length encoding that represents consecutive identical bits (also called a fill or a gap) by their bit value and their length. The bit value of a fill is called the fill bit. If the fill bit is 0, we call it as 0-fill otherwise for 1 called as 1-fill.

Given a bit sequence, BBC first divides it into bytes and then group bytes into runs. Run consists of a fill word followed by a tail of literal bytes. It always contains a number of whole bytes which represents the fill length. Byte Alignment property limits a fill length to be an integer multiple of bytes. This ensures that during any bitwise logical operation a tail byte is never broken into individual bits. In BBC we group the bits (which are either 0 or 1) in bytes and then compress them.

BBC can be categorized into two forms:

a) one sided (compresses only fills of 0's or 1's) and

b) two sided (compresses both fills of 0's and 1's)

BBC has got two types of runs viz. fill and literal. The fill runs are the ones where we actually store the compressed form of long runs, whereas in case of literal, the bits (which are present in the Byte) have literal meaning i.e., 0 - there is no record and 1 - there is a record in the database.

Procedure for two sided BBC:

BBC scans through records and generates a bit map vector for it. Then, it categorizes this into either of four runs, which are described below.

Type#1 run:

<header byte> <0 to 3 literal bytes>

header byte:

1 <fill bit> <fill length - 2 bit> <tail length - 4 bits>

fill bit signifies whether the runs are of 0's or 1's.

With the help of this run, we can store a run having length less than or equal to 24. (as we have 2 bit for fill length) we can store four values viz. 0 to 3, a value of 3 in fill length bits signifies a run length of 8*3 = 24 (since BBC organizes the run in two groups of 8).

After the fills we can have a maximum of 15*8=120 bits which can act as literal bits.

Tail length has 4 bits, so the maximum number possible is 15 => 15*8=120 bits.

e.g 00 8A 37 (in hexa)

here we have a runs of zero.

Its run length is 1.

so, the header bit will now become

1 0 01 0010 1000 1010 0011 0111

92 8A 37.

once we compress this bit, we can identify this run, if a number has a value >= 127. Since the MSB bit is 1, in the header byte.

Type#2 run:

<header byte>

01 <fill bit> <fill length - 2 bits> <odd bit position - 3 bits>

fill bit signifies whether the runs are of 0's or 1's.

With the help of this run, we can store a run having length less than or equal to 24. (as we have 2 bit for fill length) we can store four values viz. 0 to 3, a value of 3 in fill length bits signifies a run length of 8*3 = 24 (since BBC organizes the run in to groups of 8).

After the fills we can have a literal bit which has got only one 1 and other 7 bits are zeros. The bit which is one is stored in the header information.

e.g 80 (HEX)

01 0 00 111

47 (HEX).

Here we don't have a runs in the beginning so, the fill length is made 0. In the literal bit we have 1 in the MSB position. Hence the odd bit position is taken as 7. (7 for MSB and 0 for LSB).

e.g 00 00 00 02.

01 0 11 001

59 (HEX).

we have a fill length of 3, followed by a literal which has one in the 1st position (0th position being LSB). once we compress this bit, we can identify this run, if a number has a value >= 64 and < 127.

Type#3 run:

<header> <multi byte counter> <literal>

001 <fill bit> <tail length - 4 bit>

The disadvantage of Type 1 run is that it can accomodate run which are less than or equal to 120. To overcome this we have Type 3 run, where in we can store runs which are even larger.

fill bit represent the type of run.

The length of fill bits is represented using the multibyte counter.

If MSB of multibyte counter is 1, it signifies that there is another multibyte counter following it.

If MSB of multibyte counter is 0, it signifies that there is a no multibyte counter following it.

no.of.fill bits = sum of lower 7 bits in the multi byte counter + 4

we add 4 because, initially we knew that we have started from type 1 run, which has accounted for 3 runs. So, if we are working on type 3 run it means we will be having runs which is >=4.

Tail holds the literal bits.

e.g 00 (9 times) F3.

here we have a fill length of 9, which can't be shown using the type 1 run.

so, we go for type 3 run.

fill length = 9-4 =5.

001 0 0001 0000 0101 1111 0011

0001 = > there is only one literal byte.

0000 0101 => this is the last Multi byte counter. (as MSB is 0).

and the length of fill is 5 + 4 =9

once we compress this bit, we can identify this run, if a number has a value >= 32 and < 64.

Type#4 run:

<header> <multi byte counter> <literal>

0001 <fill bit> <odd bit position>

The disadvantage of Type 2 run is that it can accommodate run which are less than or equal to 120. To overcome this we have Type 4 run, where in we can store runs which are even larger.

fill bit represent the type of run.

The length of fill bits is represented using the multibyte counter.

If MSB of multibyte counter is 1, it signifies that there is another multibyte counter following.

If MSB of multibyte counter is 0, it signifies that there is a no multibyte counter following.

no.of.fill bits = sum of lower 7 bits in the multi byte counter + 4

we add 4 because, initially we new that we have started from type 1 run, which has accounted for 3 runs. So, if we are working on type 3 run it means we will be having runs which is >=4.

Tail holds the literal bits. we can have only one literal byte.

e.g 00 (9 times) 02.

here we have a fill length of 9, which can't be shown using the type 1 run.

so, we go for type 3 run.

fill length = 9-4 =5.

0001 0 001 0000 0101

001 = > one is present in the 1st position.

0000 0101 => this is the last Multi byte counter. (as MSB is 0).

and the length of fill is 5 + 4 =9

once we compress this bit, we can identify this run, if a number has a value >= 16 and < 32.

2.3.2. Word Aligned Hybrid Code (WAH)

Similar to BBC, this technique is also a hybrid between the run length encoding and the literal scheme. WAH scheme is much simpler and efficient than BBC [Wu et al., 2004]. WAH stores compressed data in words rather than in bytes.

Two types of word in WAH

1) Literal Word
2) Fill Word

MSB of a word to distinguish between a literal word (0) and a fill word (1) without explicitly extracting the bit.

Lower bits of a literal word contain the bit values from the bitmap. Second MSB of a fill word is the fill bit and lower bits store the fill length.

Imposing word alignment requires all fill lengths to be integer multiples of no. of bits, which ensures that logical operation functions only need to access words not bytes or bits. In this section, we briefly review the main characteristics of the compression algorithm used; namely the word-aligned hybrid run-length code or WAH for short. We don't describe how the logical operations are performed without decompression. Interested readers can find details in a technical report [Wu et al., 2001]. As the name suggests, this scheme is a variation on the run-length code. The essence of the run-length code is to represent a list of consecutive identical bits by its length and its bit value. As is common in the literature, we refer to a sequence of identical bits as a *fill*. The bit value of a fill is called the *fill bit*. The number of bits in a fill is called the *fill length*. To ensure efficient operations, WAH encodes the fill length and the fill bit in one whole word. Compared to the uncompressed scheme, this only reduces the space requirement if the fill is longer than a word. For fills that are shorter, WAH stores them literally. Altogether, there are two types of code words in WAH, those that contain literal bit values (called *literal words*) and those that contain fills (called *fill words*).

In our current 32-bit implementation, we use the Leftmost Bit (LMB) of a word to distinguish between a literal word and a fill word, where 0 indicates a literal word and 1 indicates a fill word. The lower 31 bits of a literal word contains literal bit values. The second leftmost bit of a fill word is the fill bit and the 30 lower bits store the fill length. To achieve fast operation, it is crucial that we impose the word-alignment requirement on this scheme. The word-alignment requirement in WAH requires all fill lengths to be integer multiples of 31 bits (i.e., literal word size). Given this restriction, we represent fill lengths in multiples of literal word size. For example, if a fill contains 62 bits, the fill length will be recorded as two (2), as shown in figure 2.7.

124 bits	1,20*0,3*1,79*0,21*1		
31-bit groups	1,20*0,3*1,7*0	62*0	10*0,21*1
Groups in hex	40000380	00000000 00000000	001FFFFF
WAH(hex)	40000380	80000002	001FFFFF

Figure 2.7: A WAH bit vector.

Decompressed					
A	40000380	00000000	00000000	001FFFFF	0000000F
B	7FFFFFF	7FFFFFFF	7C0001E0	3FE00000	00000003
C	40000380	00000000	00000000	00000000	00000003
Compressed					
A	40000380	80000002		001FFFFF	0000000F
B	C0000002		7C0001E0	3FE00000	00000003
C	40000380	80000003		00000003	

Figure 2.8: A bitwise logical AND operation on WAH compressed bitmaps

Figure 2.8 shows a decompressed version of the three bitmaps involved in the operation for the purpose of illustration only. The logical operations can be directly performed on the compressed bitmaps and the time needed by one such operation on two operands is related to the sizes of the compressed bitmaps. Let the compression ratio be the ratio of size of a compressed bitmap and its uncompressed counterpart. When the average compression ratio of the two operands is less than 0.5, the logical operation time is expected to be proportional to the average compression ratio [Wu et al., 2001]

Even with small data size, in the majority of the test cases, the word-aligned scheme is still significantly faster. When the compression ratio is one, the logical operations on WAH bit vectors are about 80 times faster than the same operations on BBC bit vectors. Even when the time to read two bit vectors is included, the WAH scheme is still about 20 times faster than BBC. If we sum up all the total time values from all test cases (including different logical operations), the sum for BBC is about 12 times that of WAH. In other words, on the average WAH is about 12 times as fast as BBC. If the IO time is not included, the differences are even larger. Compared to the literal scheme, the BBC scheme is faster in less than half of the test cases; WAH is faster in about 60% of the test cases. On the average, WAH-compressed bit vectors use less than a third of the space required by the uncompressed scheme (LIT). Compared to BBC, WAH uses only about 50% more space.

In using the WAH compressed bitmap index to answer queries, bitwise logical operations are the most important operations. For this reason, we next examine the complexity of the bitwise logical operation procedures. Two different algorithms, one performs an arbitrary bitwise logical operation on two compressed bitmaps, and the other performs a bitwise OR between a decompressed bit vector and a compressed one. The first one is for general use and the second one is mainly used to sum together a large number of sparse bitmaps. Before we give detailed analyses, we first summarize the main points.

The time to perform an arbitrary logical operation between two compressed bitmaps is proportional to the total size of the two bitmaps. The exception is when the two operands are nearly decompressed; in which case the time needed is constant. The time to perform a logical OR operation between a decompressed bit vector and a compressed one is linear in the size of the compressed one. When performing OR operation on large number of sparse bitmaps using in-place OP, the total time is linear in the total size of all input bitmaps. In this case, using generic operation takes more time because it allocates memory for intermediate results and compresses them too. In contrast, using in-place Oravoids all of these operations. Operations on WAH compressed bitmaps are faster than the same operations on BBC compressed bitmaps for three main reasons.

1. The encoding scheme of WAH is much simpler than BBC. WAH has only two kinds of words, and one test is sufficient to determine the type of any given word. In contrast, our implementation of BBC has four different types of runs; other implementations have even more. It may take up to three tests in order to decide the run type of a header byte and many clock cycles may also be needed to fully decode a run.

2. During the logical operations, WAH always accesses whole words, while BBC accesses bytes. For this reason, BBC needs more time to transfer its data between the main memory and CPU registers than WAH.

3. BBC can encode short fills; say those with less than 60 bits, more compactly than WAH. However, this comes at a cost. Each time BBC encounters a short fill it starts a new run. WAH typically represents such a short fill in literal words. It takes much less time to operate on a literal word in WAH than on a run in BBC. This situation is common when bit density is greater than 0.01 in random bitmaps.

We believe it is worthwhile to trade this 50% more space for 12 fold increase in operation speed. The gzip scheme is based on an asymptotically optimal compress scheme. Even compared again this optimal scheme, WAH scheme uses no more than twice as much space. For this extra space, WAH is able to perform logical operations several orders of magnitudes faster than gzip. Overall, we believe WAH is the most appropriate scheme for compressing bitmap indexes. Wu and others [2004] propose a simple algorithm for compressing the bitmap indexes that improves the speed of logical operations by an order of magnitude at a cost of small increase in space. This algorithm not only supports faster logical operations but also enables the bitmap index to be applied to attributes with high cardinalities. Our tests show that by using WAH compression, we can achieve good performance on scientific datasets where most attributes have high cardinalities. From their performance studies, Johnson and colleagues came to the conclusion that one has to dynamically switch among different compression schemes in order to achieve the best performance [Amer-Yahia and Jhonson, 2000]. We found that since WAH is significantly faster than earlier compression schemes, there is no need to switch compression schemes in a bitmap indexing software. The new compression scheme not only improves the performance of the bitmap indexes but also simplifies the indexing software.

Goyal and others [1999] have considered the application of compression techniques to data warehouse indexes. They examined a recently proposed access structure for warehouses known as *DataIndexes* and discussed the application of several compression methods to this approach and discusses when each of them should be used. Wu and others [2001] talks about bitmap compression as bitmaps are easy to compress but compressing them reduce the query processing efficiency. To solve this problem they developed a new word-aligned compression scheme technique. They also evaluated several bitmap encoding schemes, like equality encoding, range encoding and interval encoding. Their results shows that the compressed bitmap indexes are not only much smaller in size than their uncompressed versions, but are also just as fast in query processing as their uncompressed counterparts. Systematic analysis of the effectiveness of the two most efficient bitmap compression techniques, the Byte-aligned Bitmap Code (BBC) and the Word-Aligned Hybrid (WAH) code have been discussed in [Wu et. al., 2004]. Their analysis shows that both compression schemes can be optimal. They also proposed a novel strategy to select the appropriate algorithms so that their optimality can be achieved in practice. Their results show that the sizes of the compressed bitmap indexes are relatively small with the typical B-tree indexes. This is even true for high cardinality attributes.

Wu and others [2006] discusses word-aligned Hybrid Compression technique for bitmap indexes making them efficient even for high-cardinality attributes. They proved its optimality for one-dimensional range queries. Their main result states that the time required to answer a one-dimensional range query is a linear function of the number of hits. This strongly supports the well-known observation that compressed bitmap indexes are efficient for multidimensional range queries because results of one-dimensional range queries computed with bitmap indexes can be easily combined to answer multidimensional range queries. They show that WAH not only reduces the bitmap index sizes but also improves the query response time. Amer-yahia and Johnson [2000] discuss about the compressed bitmap indexes to accelerate decision support queries. They showed that there are several fast algorithms for evaluating Boolean operators on compressed bitmaps. These algorithms have different execution times for different Boolean operations and for different bitmaps. They present a linear time dynamic programming search strategy based on a cost model to optimize query expression evaluation plans. Their results show that the optimizer requires a negligible amount to time to execute, and that optimized complex queries can execute up to three times faster than un-optimized queries on real data.

Johnson and others [2004] have presented a lossless compression strategy to store and access large matrices efficiently on disk. Their approach is based on viewing the columns of the matrix as points in a very high dimensional Hamming space, and then formulating an appropriate optimization problem that reduces to solving an instance of the Traveling Salesman Problem on this space. Richards [1986] talks about file compression when all the records of the file have the same field structure. Here 'differencing' compression scheme is analyzed. The idea is to arrange the records in some order and output the first record, which differs from the previous record. The problem is to sort the records so that they are in a Gray-code order. They presented an improvement to Ernwall's algorithm, extend it to the full mixed-radix case and present another algorithm. Stockinger and others [2006] described an integration effort that can significantly reduce the unnecessary reading all variables into memory by using an efficient compressed bitmap indexing into ROOT framework. By using this index, any arbitrary combinations of queries can be answered very efficiently. Their performance results show that for multi-dimensional queries, bitmap indexes outperform the traditional analysis method up to a factor of 10. Wu and others [2001] presented a comparison of two word based compression schemes with BBC. Their results show that these word-aligned schemes take only 50% more space than BBC but perform logical operations 12 times faster on both real application data and synthetic data. Wong and others [1985] introduces a file structure called bit transposed file which suits the special characteristics of large Scientific/Statistical Database applications. The data is stored by vertical bit partitions. The bit patterns of attributes are assigned using one of several data encoding methods. The bit partitions can be compressed using a version of the run length encoding scheme. Results from experiments shows that bit transposed may be a reasonable alternative file structure for large Scientific/Statistical Databases.

To reduce the query response time, [Wu et. al., and 2006] designed a CPU-friendly scheme named the word-aligned hybrid code. They proved that the sizes of WAH compressed bitmap indexes are about two words per row for large range of attributes. This size is smaller than typical sizes of commonly used indexes, such as a B-tree. Therefore, WAH compressed indexes are not only appropriate for low cardinality attributes but also for high cardinality attributes. In the worst case, the time to operate on compressed bitmaps is proportional to the total size of the bitmaps involved. The total size of the bitmaps required to answer a query on one attribute is proportional to the number of hits. These indicate that WAH compressed bitmap indexes are optimal. Tests on a STAR dataset show that it is 12 times faster than BBC while using only 60% more space. They have also shown through both analyses and tests that the query processing time grows linearly as the index size increases. In addition, they demonstrated that the query processing time is linear in the number of hits when using a WAH compressed bitmap index. This proves that WAH compressed bitmap indexes are optimal. Wu and others [2002] studied the effects of compression on bitmap indexes. To make compressed bitmaps operate more efficiently, they designed word-aligned hybrid code (WAH). They demonstrated from tests that improving the compression scheme actually improves the query answering speed, not only logical operations. Tests show that WAH compressed indexes are not only smaller than the uncompressed indexes, they also take less time to answer queries. Compared to the indexes compressed with BBC, the WAH compressed indexes are faster by a factor of four or five. They did not see a factor of 12 improvements because the times spent in query processing are dominated by logical operations on very sparse bitmaps. On very sparse bitmaps, WAH scheme is faster than BBC usually by a factor of about four or five. During query processing there is also some amount of time spent in parsing the query, obtaining the locks.

Pinar and others [2005] evaluates compressed bitmap indexes for scientific databases such as high energy physics. Study has been carried out to reorganize bitmap tables for improved compression rates. Their algorithms are used just as a preprocessing step, thus there is no need to revise the current indexing techniques and the query processing algorithms. We introduce the tuple reordering problem, which aims to reorganize database tuples for optimal compression rates. We propose Gray code ordering algorithm for this NP-Complete problem, which is an in-place algorithm, and runs in linear time in the order of the size of the database. We also discuss how the tuple reordering problem can be reduced to the traveling salesperson problem. Their experimental results on real data sets show that the compression ratio can be improved by a factor of 2 to 10.

2.4. Binning Techniques

The simplest form of bitmap indexes works well for low-cardinality attributes. However, for high-cardinality attributes simple bitmap indexes are impractical due to large storage and computational complexities. We have seen the simple bitmap indexes and encoding schemes like range encoding and interval encoding which are best suited for one sided range queries and two sided range queries. In addition we have also seen equality encoding which is the basic bitmap index scheme for simple equality queries. We have just discussed how different encoding methods could reduce the index size and improve query response time. Next, we describe a strategy called ***binning*** to reduce the number bitmaps.

Koudas [2000] introduced the idea of binning for bitmap indexes. Instead of representing one bitmap vector for each distinct value of the attribute domain, design bitmap for a group of attribute value by taking into consideration the query and data distribution. By doing so, we can control the size of bitmap index by reducing the number of bitmap vectors. Now the size of bitmap index is no longer directly proportional to the cardinality of the attribute domain. This makes bitmap indexing suitable for high cardinality attributes as well. Here, mathematical formation and its optimal solution is also discussed. The solution for choosing bitmap ranges efficiently is optimal and polynomial time. Compressed bitmaps were also considered and branch and bound algorithm for choosing bitmap ranges is proposed. Wu and Yu [1998] introduced the term Range-based bitmap indexing for binning, where the attribute values are partitioned into ranges and a bitmap vector is used to represent a range. Range-based indexing results in non-uniform search times for different queries as the number of records assigned to different ranges can be highly uneven. They proposed and evaluated a dynamic bucket expansion and contraction approach to construct range-based bitmap indexes for multiple high-cardinality attributes with skew. Their results shows better performance with highly skewed data for naïve partition approach and performs favorably with the optimal approach.

All these bitmap encoding techniques and schemes discussed so far are optimized for discrete attribute values and they cannot work with the real time scientific data and multi dimensional index data structures for such scientific data. Hence we go for an algorithm which works well with real attribute values. This generic algorithm is called the “Generic Range Encoding Algorithm”. The main problem with this algorithm is that we have to do candidate check for getting the exact hits as we are encoding the attribute ranges and not exact values of attributes. Hence candidate check problem determines the performance factor in this case Since the encoding methods described before only take certain integer values as input, we may also view binning as a way to produce these integer values (bin numbers) for the encoding strategies. The basic idea of binning is to build a bitmap for a *bin* rather than each *distinct attribute value*. This strategy disassociates the number of bitmaps from the attribute cardinality and allows one to build a bitmap index of a prescribed size, no matter how large the attribute cardinality is. A clear advantage of this approach is that it allows one to control the index size. However, it also introduces some uncertainty in the answers if one only uses the index. To generate precise answers, one may need to examine the original data records (candidates) to verify that the user specified conditions are satisfied. The process of reading the base data to verify the query conditions is called ***candidate check*** [Stockinger et al., 2004, Rotem et al., 2005b]. A small example of an equality-encoded bitmap index with binning is given in Figure 2.9. In this example we assume that an attribute *A* has values between 0 and 100. The values of the attribute *A* are given in the second leftmost column. The range of possible values of *A* is partitioned into five bins [0, 20), [20, 40).... A “1-bit” indicates that the attribute value *falls into a specific bin*. On the contrary, a “0-bit” indicates that the attribute value *does not* fall into the specific bin. Take the example of evaluating the query “Count the number of rows where 37 <= A < 63”. The correct result should be 2 (rows 5 and 7). We see that the range in the query overlaps with bins 1, 2 and 3. We know for sure that all rows that fall into bin 2 *definitely qualify* (i.e., they are *hits*). On the other hand, rows that falls into bins 1 and 3 *possibly qualify* and need further verification.

In this case, we call bins 1 and 3 *edge bins*. The rows (records) that fall into edge bins are *candidates* and need to be checked against the query constraint.

In the following example, there are four candidates, namely rows 1 and 3 from bin 1, and rows 5 and 6 from bin 3. The candidate check process needs to read these four rows from disk and examine their values to see whether or not they satisfy the user-specified conditions.

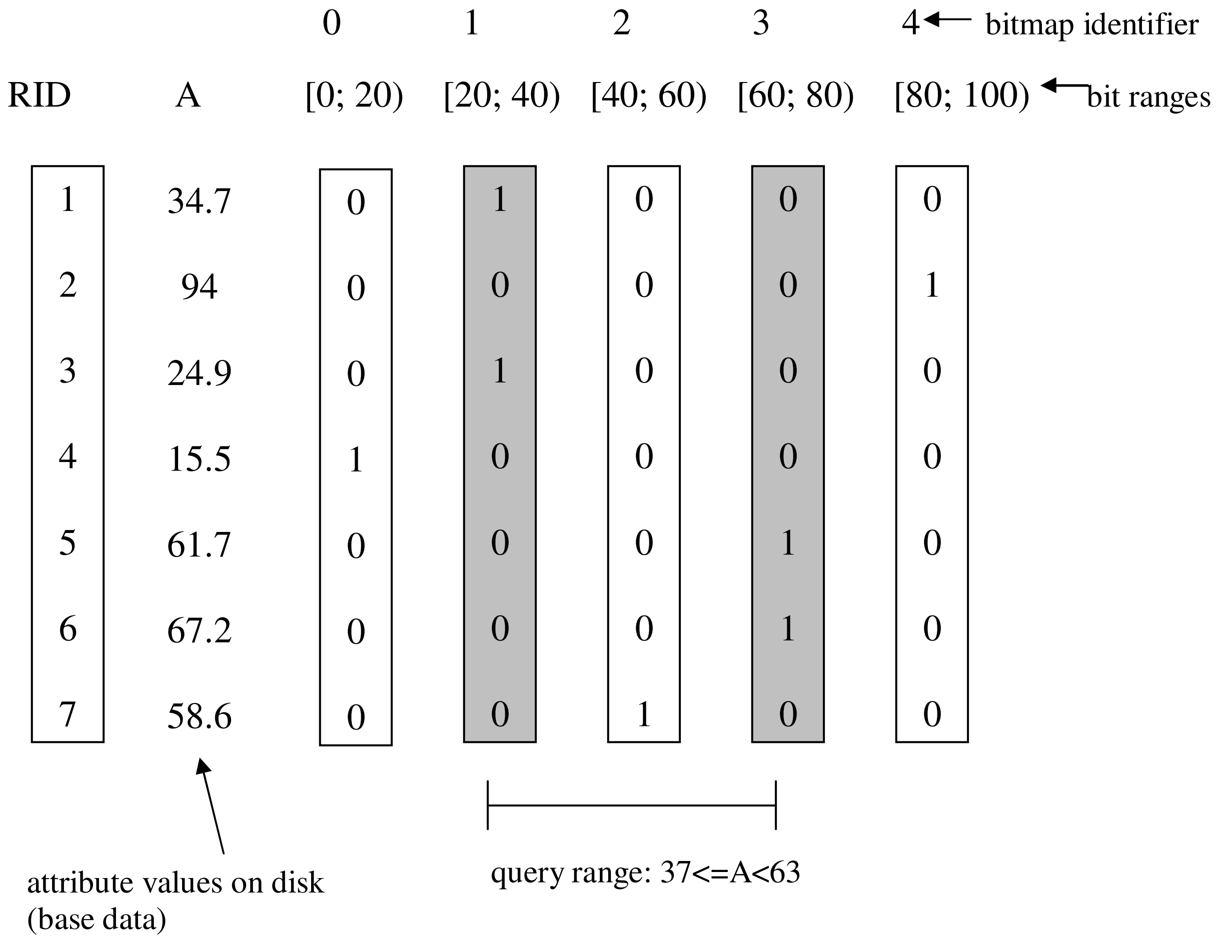


Figure 2.9: Range query "37 <= A < 63" on a bitmap index with binning.

On a large data set, a candidate check may need to read many pages and may dominate the overall query response time [Rotem et al., 2005b]. There are a number of strategies to minimize the time required for the candidate check [Stockinger et al., 2004, Rotem et al. 2005a, 2005b]. Koudas [2000] considered the problem of finding the optimal binning for a given set of equality queries. Rotem and others considered the problem of finding the optimal binning for range queries. Their approaches are based on dynamic programming. Since the time required by the dynamic programming grows quadratic with the problem size, these approaches are only efficient for attributes with relatively small attribute cardinalities or with relatively small sets of known queries. Stockinger and others [2004] considered the problem of optimizing the order of evaluating multi-dimensional range queries. The key idea is to use more operations on bitmaps to reduce the number of candidates checked. This approach usually reduces the total query response time. Further improvements to this approach are to consider the attribute distribution and other factors that influence the actual time required for the candidate check. To minimize number of disk page accesses during the candidate check, it is necessary to *cluste*r the attribute values. A commonly used clustering (*data layout*) technique is called the *vertical partition* or otherwise known as *projection index*. In general, the vertical data layout is more efficient for searching, while the horizontal organization (commonly used in DBMS) is more efficient for updating. To make the candidate check more efficient, we recommend the *vertical data organization*. We have seen that for high dimensional data the storage overhead for bitmaps indexes is very high. There are two possible solutions to deal with this problem namely, generic range evaluation algorithm where in attribute ranges are encoded rather than attribute values. The other one is to compress the bitmap that has been built and thus save the large storage overhead. Though the reduction in storage over head that is achievable with range encoding or binary encoding comes at the cost of degraded query response hence study of bitmap compression is of great importance.

2.4.1. Binning Strategies

Dynamic Bucket Expansion and Contraction (DBEC), the data are first scanned into the buffer to construct the bucket ranges by counting the data points falling into each bucket. If a bucket grows beyond a threshold, it is expanded into smaller-range buckets [Wu andYu, 1998]. After the scan, adjacent buckets are combined it into the final required number of buckets with approximately balanced count. Bitmap vectors are then built for the contiguous ranges represented by the final buckets. [Stockinger et. al., 2004] presented a new strategy to evaluate queries using bitmap indexes for very high cardinality attributes. They considered scientific data analysis applications where most of the attributes have very high cardinalities. They analyzed how binning affects the number of pages accessed during query processing and proposed an optimal way of using bitmap indexes to reduce the number of pages accessed. Their strategy reduces the query response time by up to a factor of two by minimizing the number of records scanned during the candidate checking, but requires more operations on bitmaps. They provided detailed analyses and experimental measurements to verify the efficiency of their new strategy. Three discussed strategies are as follows:

All queries to be conjunctive with a one-sided range condition on each attribute $x_i < v_i$.

Strategy 1: In the first phase the bitmap index is scanned for both attributes. In the next phase, the candidate check for attribute 1 is performed by reading the attribute values from disk and checking them against the range condition and similarly for attribute 2 is done.

Strategy 2: This strategy evaluates each dimension separately. The bitmap index for the first attribute is evaluated first and the candidates of this attribute are checked immediately afterward. Similarly, dimension 2 is evaluated.

Strategy 3: The third strategy is an optimal combination of the above two strategies.. In this the first part is similar to strategy one i.e. we find the combined total candidates for all the dimension asked in the query and instead of reading all those pages for candidate checking we now follow a different strategy.

We do the candidate checking for only those which are in the intersecting regions of the Ctotal (candidates of all dimensions) and candidates of dimension 1 i.e. C1. We then combine the candidates that passed the candidate checking to the Ctotal and use the new Ctotal for doing the same with rest of the dimensions. By proceeding in this way we are filtering out all the unnecessary candidate checks that would have been done if candidate checking is done for each attribute separately and then combining them and finally filtering out the rest which do not meet all the query requirements i.e. we are performing the masking of candidates that satisfy the query constraint and eliminating the unnecessary candidates which even might be successful in their individual candidate checking but do not form the part of final result. Thus the third strategy seems to be more advantageous over the other two as it has both the strategies intertwined to give a better performance.

Since many scientific applications operate on floating point number, Stockinger, [2001] presented a novel algorithm called GenericRangeEval for processing one-sided range queries over floating point values. They also presented a cost model for predicting the performance of bitmap indexes for high-dimensional search spaces. They studied analytically and experimentally the performance behavior of multi-dimensional queries. Their experimental evaluation showed that the cost model predicted the performance of the bitmap index fairly well and could thus easily be incorporated into a query optimizer. It can decide whether the query shall be answered by using the index and tell its estimated query response time.

Stockinger [2002] analyzed the behavior of GenericRangeEval bitmap index algorithm against various queries based on different data distributions. They also implemented an improved version of BBC. Results show that depending on the underlying data distribution and the query access patterns, proposed bitmap indexes can significantly improve the response time of high-dimensional queries when compared to conventional access methods. They demonstrated that the query response times of compressed bitmap indexes can be significantly lower than for uncompressed bitmap indexes. Stockinger and others [2000] discuss the design and implementation of bitmap indexes for High-Energy Physics analysis, where the potential search space consists of hundreds of independent dimensions. They evaluated both equality and range encoding techniques and found that the number of bitmap vectors per attribute is a central optimization parameter. Their results helps in choosing the optimal number of bitmap vectors for multi-dimensional indexes with attributes of different value distribution and query selectivity. They solved the candidate check bottleneck by increasing the number of bins and came to an optimal query performance. 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.

Stockinger and others [2005] propose a novel approach to scalable data analysis for large scientific databases by combining bitmap indexing and visualization infrastructure. They combined bitmap indexing with a visualization pipeline for generating images of abstract data results in a tool suitable for use by scientists in fields where data size and complexity poses a barrier to efficient analysis. They present an architectural overview of the system (Dexterous Data Explorer) along with an analysis showing substantial performance over traditional visualization pipelines. Bitmap Indexes are used to quickly locate features in data and grow them into connected regions. The results are then used as input to the visualization pipeline. Rotem and others [2004] studied the problem of finding optimal locations for the bin boundaries in order to minimize the access costs subject to storage constraints. They proposed a dynamic programming algorithm for optimal partitioning of attribute values into bins that takes into account query access patterns as well as data distribution statistics. Their optimized binning strategy improves the query response time and reduces candidate check costs. Re-arranging the bin boundaries reduces the total size of the bitmap indexes and also re-arranging attribute values between bins. This strategy can be used for periodically reorganizing the bitmap index based on observed query workloads.

Rotem and others [2005] studies strategies for minimizing the access costs for processing multi-dimensional queries using bitmap indexes with binning. Their approach includes optimally placing the bin boundaries and dynamically reordering the evaluation of the query terms. They derive several analytical results concerning optimal bin allocation for a probabilistic query model Based on data distribution and query distributions, their approach place bin boundaries such that the number of candidates that need to be checked against the query constraints is minimized. They also suggest reordering the evaluation of the attributes in multi-dimensional queries according to the estimated attribute selectivity. The results show that as the number of query dimensions increase, the efficiency of their algorithm increases.

Rotem and others [2006] studied strategies for minimizing the I/O costs for candidate checking for multi-dimensional queries. Determining the number of bins allocated for each dimension and then placing bin boundaries in optimal locations do this. Their algorithms also take data distribution and query distributions and finds bin allocation for each attribute. They performed analytical and experimental studies to evaluate the efficiency of their algorithm**.** Rotem and others [2005b] have introduced a novel algorithm for improving the query response time of bitmap indexes by computing optimal bin boundaries. They also aimed at working on candidate check problem by minimizing the total time required to answer queried by optimally placing the bin boundaries using their dynamic programming algorithm. They presented an analytical and experimental evaluation of the performance of synthetic and real query workloads for a large data set from the Sloan Digital Sky Survey. Their results show improvement by a factor of 2. Rotem and others [2005a] studied several strategies for optimizing the candidate check cost for multi-dimensional queries. They present an efficient candidate check algorithm based on attribute value distribution, query distribution as well as query selectivity with respect to each dimensional. They also showed that re-ordering the dimensions during query evaluation can be used to reduce I/O costs. Their results show significant improvements.

2.4.2. Binning Cost Model

So far we have discussed various optimization techniques, different strategies and different algorithms for achieving the faster query response time and lesser I/O costs. We have discussed different binning strategies where in we chose some number of bins for binning and the sole concentration was on selecting the bin boundaries and optimizing this selection process but now we pay attention to selection of the number of bins that we are going to use and the optimization of this number. Here we present a probabilistic model that optimizes the number of bins to be used based on frequency with which the attributes appear in the queries i.e. more bins for attributes that appear more frequently and fewer bins for attributes that appear less frequently. A brief introduction to what we have already discussed in detail includes choosing of optimal bin boundary placement. A dynamic programming algorithm that works well for the single attribute case but when it comes to multi attribute case a lot of parameters arise such as how many bins have to be chosen for each attribute, likelihood of an attribute to appear in a query and the selectivity factor. On a whole we can treat it as the total cost associated with binning of each attribute A_i which is $\sum$Cost (A_i). But this is an NP-hard problem as the function Cost(A_i) depends on ki(number of bins) for each attribute A_i and in general depends on the query and data distributions. Hence we apply a closed form solution to multi dimensional bin allocation problem where each Cost (A_i) is a differentiable function under the probabilistic query model [Rotem et al., 2005].

The solution is described in two pahses. In the first phase we determine the number of bins to be allocated to each attribute based on data and query statistics. In the second phase we proceed with this bin allocation and applying the dynamic programming algorithm we have used for the single attribute case for each attribute separately.

Probabilistic query model:

In this model that we are going to develop on the query set Q we assume attribute independence i.e. the probability p*i* that an attribute A_i will show up in a query is independent of the probability of appearance of other attributes in the query. Thus for a query q_i the probability $\left[\prod_{Ai\in q} p_i\right]\left[\prod_{Ai\notin q}(1-p_i)\right]$ where A_i *belongs to* q_i, implies that A_i is specified in the query q_i and vice versa. The selectivity of an attribute is denoted by S_i. The query evaluation is done in phases where in each phase the range condition is evaluated on one of the attributes and the candidates that pass through the current phase are sent to next phase to check for the range condition of next attribute. Thus the number of candidates checked in each phase is a multiple of total records in the data base and selectivities of attributes checked in the previous phases

A cost formula for candidate check is the derived as a recursive equation which is

$$C(t) = \sum_{i=1}^{t} p_i * f(k_i) \prod_{j=i+1}^{t}(1-p_j(1-s_j)) \qquad (2.7)$$

Where Cost of candidate check on attribute A*i* i.e. Cost (A*i*) is a function *f(k_i)* on the number of bins k_i. We denote it as expected cost of candidate check on *j* attributes in the order A_j, A_{j-1},…,A_t. with k_i bins allocated to each attribute and is simply denoted as C(*j*,N) where N is the total number of records or simply C(*j*).

Finally a formula is derived for finding the optimal binning allocation which is

$$\frac{f'(k_i)}{f'(k_i+1)} = \frac{p_i+1}{p_i(1-p_i+1(1-s_i+1))}$$ for 1< *i* <t, where t is the total number of attributes.

Choosing proper binning strategies is also important as it may affect the query response time considerably. Equal depth binning and equal width binning are two possible binning strategies which can be implemented based on knowledge of data distribution and query patterns.

2.4.3. Factor effecting Binning strategies

Following are the main factors which guides the binning strategies:

- Due to disk storage constraints, bitmap indexing systems that use binning must limit the number of bins that are allowed per attribute. Such constraints are still applicable even when bitmap compression is effectively used.
- Effective binning strategies attempt to compute bin location boundaries that minimize the I/O cost incurred by the candidate check step subject to total index storage constraints. It turns out that an optimal binning strategy must be sensitive to both query distribution as well as data distribution.
- Query distribution, in terms of location of query endpoints and popularity of queries, may affect bin boundary locations as the number of edge bins may be minimized by attempting to align bin boundaries with query endpoints. In addition, more bins can be allocated to data regions that are heavily hit by queries.
- Data distribution affects the binning strategy as one can allocate more bins to densely populated regions of the data to avoid costly candidate check operations on edge bins with many values.

2.5. Complexity Issues and Research Gaps

The simple bitmap indexing works well with attributes having low cardinalities. The size of bitmap index can be very large for a high cardinality attribute where there are thousands or even millions of distinct values. B-trees have been widely adopted in database systems for external indexing. Their strength is their dynamic nature, performance and stability under update – the properties that are not required in a Data Warehouse (DW). In the DW environment, building simple bitmap indexes usually costs less time and space than building B-trees. The restriction on simple bitmap indexes is that they are not suitable for high cardinality attributes. Therefore, there is a need for a new indexing technique, which does not have this restriction, but at the same time hold the advantage of bitmap indexes.

It is well accepted that I/O cost dominates the query response time when using out-of-core indexing methods. Thus, most indexing techniques focus on minimizing I/O cost. For bitmap indexes, most research efforts concentrate on reducing the sizes of indexes. However, tests show that the computation time can dominate the total time when using compressed bitmap indexes. In addition, as main memories become cheaper, we expect that "popular" bitmaps would remain in memory. This would further reduce the average I/O cost and make the time spent in CPU more prominent part of the total query processing time. For these reasons, we seek to improve the computational efficiency of operations on compressed bitmaps. The space requirement of a simple bitmap index is a linear function of the cardinality of the indexed attribute and of the indexed table, and the index processing time for a single value selection is a linear function of the length of bitmaps. The sparsity of the bit vectors increases with the cardinality resulting in poor space utilization and high processing cost. Many variations of bitmap indexing have been proposed to solve the sparsity problems. Two common objectives of the proposed methods are (1) reducing the space complexity of the index and (2) improving the performance of index processing. Many strategies have been devised to reduce the index sizes, such as, more compact encoding strategies, binning and compression.

Based on above literature survey, we aimed at following objectives for our research:

- Finding new encoding scheme and encoding algorithms to process queries.
- Finding new compression algorithms that retains are advantages of bitwise operators and solves are queries without decompressing the compressed bitmaps. Finding new strategies to improve compression ratios for existing compression techniques.
- Finding new binning strategies and algorithms to solve queries with better performance than existing techniques.

2.6. Summary

This chapter gives an overview of some approaches on bitmap indexing. As has been shown, bitmap indexes offer an efficient way of indexing data that remains consistent most of the time. The methods and approaches discussed in this chapter can help integrating bitmap indexes in existing databases. All in all, bitmap indexes are becoming more and more popular, as indicated by the increasing use in commercial databases like the ones by Oracle, RedBrick and Sybase.

In this chapter, we reviewed a number of recent developments in the area of bitmap indexing technology. We reviewed the existing literature and organized much of the research work under the three orthogonal categories of encoding, compression and binning.

Chapter 3: Compression Strategies and Encoding Techniques

3.1. Introduction

Data Warehouses are becoming more important for decision makers. Most of the queries against a large data warehouse are complex and iterative. The ability to answer these queries efficiently is a critical issue in the data warehouse environment. If the right index structures are built on the columns, performance of the queries especially ad-hoc queries will be greatly enhanced. This efficient query processing can be done using bitmap indexing techniques, which helps in improving the time and space complexities of the vast and complex queries that data warehouses deal with. The simplification of query processing in turn speeds up the analysis process which is vital to any large organization.

Bitmap Indexing is one such technique. It primarily functions by representing the huge amount of information stored in data warehouses as bitmap vectors which nothing but binary representations of the data stored and the processing of binary data is many folds faster than the processing of textual/numerical data. Though simple bitmap indexing simplifies query processing and improves time complexity thereby speeding up the processing of complex real time ad-hoc queries, it has a major bottleneck of dealing with space efficiently caused by huge cardinality datasets. In this chapter, an attempt has been made to address the problem.

3.2. Encoding of Bitmap Indexes

The simple bitmap index consists of a collection of bitmap vectors each of which is created to represent each distinct value of the indexed column. The i^{th} bit in a bitmap vector, representing value *x*, is set to 1 if the i^{th} record in the indexed table contains *x*. A bitmap for a value: an array of bits where the i^{th} bit is set to 1 if the i^{th} record has the value. A bitmap index consists of one bitmap for each value that an attribute can take [O'Neil, 1987].

3.2.1. Simple Bitmap Indexing

The basic idea of simple bitmap indexing is to use a string of bits (0 or 1) to indicate whether an attribute in a table is equal to a specific value or not. The position of a bit in the bit string denotes the position of the tuple in the table. The bit is set, if the content of an attribute is associated with a specific value. For a typical example, a simple bitmap index on the attribute gender, where the domain of gender is {*Male, Female*}, results in two bitmap vectors B_{Male} and B_{Female}. For B_{Male} the bit is set to 1, if the associated tuple has the attribute gender = *Male,* otherwise the bit is set to 0. For B_{Female} the bit is set to 1, if the associated tuple has the attribute gender = *Female,* otherwise the bit is set to 0. The simple bitmap index on the attribute Gender, B_{grnder}, is a collection of bitmap vectors { B_{Male}, B_{Female} } [Buchmann, 1998].

3.2.2. Encoded Bitmap Indexing

Suppose that a fact table attributes, sales, containing N tuples of sales data and a dimension table, products, containing information about 12000 different products. Traditionally, in order to build a simple bitmap index on products table, it will result in 12000 bitmap vectors of N bits in length. In encoded bitmap indexing instead of 12000 vectors, only $[\log_2 N] = 14$ bitmap vectors, plus a mapping table is used. For example, suppose that the domain of attribute A of a table T is {a, b, c}. Instead of 3 bitmap vectors, we use $[\log_2 3] = 2$ vectors to build the index on attribute A. 2 bits are used to encode the domain {a, b, c}, where 'a' is encoded as 00, 'b' is encoded as '01' and 'c' is encoded as '10' respectively. For those tuples with A = 'a', set the corresponding positions in both bitmap vectors B_1 and B_2 to 0 and so on. The retrieval Boolean functions for 'a','b','c' are $B'_1B'_2$, B'_1B_2 and $B_1B'_2$ respectively [Wu and Buchmann, 1998].

3.2.3. Maintenance of Encoded Bitmap Indexes

As data in a data warehouse is updated, the encoded bitmap indexes also need to be maintained to reflect the changes. There are two categories of updates:

1) Updates without domain expansion

2) Updates with domain expansion

Updates without domain expansion: In the example stated in the previous section, if a tuple with A = b is appended to table T, then we need to append only B_1 [j] = 0 and B_0 [j] = 1 at the end of bitmap vectors B_1 and B_0 where j is the position of the newly inserted tuple in table T.

Updates with domain expansion: If a tuple with A = d is appended to T, i.e. the domain is now expanded to {a, b, c, d}, then the following equation should be first tested.

$$\text{Upper Limit } (\log_2|A^{(m-1)}|) = \text{ Upper Limit } (\log_2|A^{(m)}|)$$

Where $|A^{(m-1)}|$ denotes the cardinality of A before insertion and $|A^{(m)}|$ denotes the cardinality of 'A' after insertion.If equation (1) is true, as is the case in our example, then add the mapping $M^A(d) = 11$ into the mapping table and set $B_i[j] = M^A(d)[i]$.

If another tuple with A = e is added to table T, the domain of A is now expanded to {a, b, c, d, e}, then Upper Limit $(\log_2|A^{(m-1)}|)$ < Upper Limit $(\log_2|A^{(m)}|)$. The following actions need to be taken to reflect the change to the encoded bitmap index which will be explained by the next algorithm.

a.) Expand the mapping table M^A = {a, b, c, d} to M^A = {a, b, c, d, e}

b.) Add the bitmap vector B_2 to B_A and set B_2 equal to 0

c) Set $B_i[j] = M^A$ (e) [i], where i = 0, 1, 2 and j = the position of the newly inserted tuple in the table T

d.) Revise the retrieval Boolean functions

T

…	A	…
	a	
	b	
	c	
	b	
	a	
	. . .	
	d	

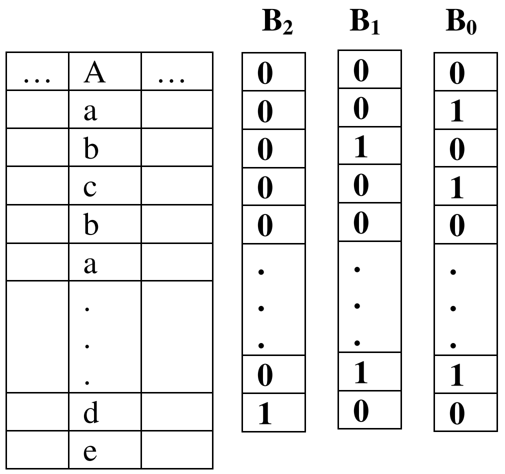
B_1
0
1
0
1
0
. . .
1

B_0
0
0
1
0
0
. . .
1

Mapping Table

a	**00**
b	**01**
c	**10**
d	**11**

Figure 3.1(a): Encoded Bitmap Index and mapping table without expansion



Mapping Table

a	**000**
b	**001**
c	**010**
d	**011**
e	**100**

Figure 3.1(b): Encoded Bitmap Index and mapping table with expansion

3.2.4. Solving Queries using Indexing Techniques

As data warehousing applications grow in size, existing data organizations and access strategies, such as relational tables and B-tree indexes, are becoming increasingly ineffective. The two primary reasons for this are that these datasets involve many attributes and the queries on the data usually involve conditions on small subsets of the attributes. Two strategies are known to address these difficulties well, namely vertical partitioning and bitmap indexes. Here, we primarily concentrate on the latter one. One important observation is that indexing is often more efficient than using B-tree based indexes to answer ad hoc range queries on static datasets. For range queries, compressed bitmap indexes are in most cases more efficient than other indexing techniques. We also evaluate the performance of two different encoding schemes for bitmap indexes. The program aims at comparing the efficiency and the performance of the indexing techniques discussed above for solving queries corresponding to a relational database.

3.2.4.1. Database Generation

The program requires large database with multiple records and multiple attributes for each record as input in order to simulate a real ad-hoc data warehouse. Data records are generated using simulation programs.

3.2.4.2. Construction of Mapping Tables

A mapping table is constructed for each field of the record. So, if a table has degree (arity) 5, then the number of mapping tables required is 5. In case of simple bitmap indexing, the value of the new tuple is verified with the already existing tuples in the table. If a match is found, corresponding to the attribute, a '1' is inserted in the position. If the match is not found, a '0' is inserted and a new record corresponding to the newly entered value is made and a '1' is inserted.

In the case of encoded bitmap indexing, there will be two types of tables. One is the mapping table and the other is the encoded bitmap table. First the mapping table is verified for the value of the newly entered tuple. If a match is found, then the value corresponding to the tuple in the mapping table is added to the encoded bitmap table. If a match is not found, then the problem of domain expansion comes into light, i.e. whether to domain expansion should take place or not. If the domain expansion takes place, then there will be a slight change in the representation (a bit added) and all the updates take place as mentioned in the section 3.2.3.

3.2.4.3. Solving Queries

The query that is to be solved is present in the input file. The program takes the query as input and constructs mapping tables using both the indexing techniques. In the case of simple bitmap indexing, logical 'and'ing or 'or'ing is performed on the bits of the corresponding record number, as given in the query. If the result is '1', then the record is included in the output, otherwise, eliminated. For encoded bitmap indexing, retrieval Boolean functions are defined. If the o/p corresponding to the Boolean function defined is '1', then the record is included.

3.2.4.4. Performance Evaluation

An analysis of performance of both types of indexing has been done. The amount of space consumed for executing the "SQL select queries" using both types of indexing, simple and encoded, has been recorded, varying the number of records in the database.

SAMPLE MEASUREMENTS (SPACE REQUIRED):

NO OF RECORDS	SIMPLE BITMAP INDEXING	ENCODED BITMAP INDEXING	DIFFERENCE
5	12	14	-2
10	40	31	9
20	137	72	65
40	475	160	315
80	1750	355	1395
160	6700	785	5915
320	26200	1725	24475
640	103600	3765	99835
1280	412000	8165	403835

Table 3.1: Table showing the space required using simple bitmap indexing and encoded bitmap indexing

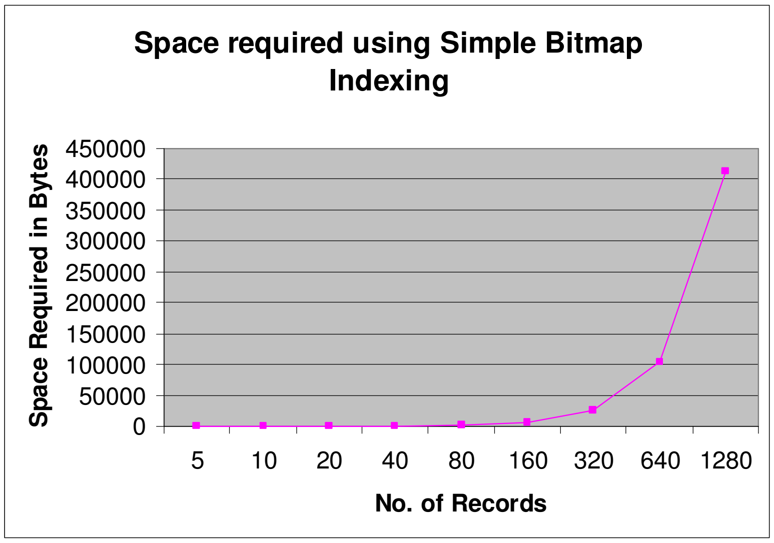


Figure 3.2: Space required for simple bitmap indexing

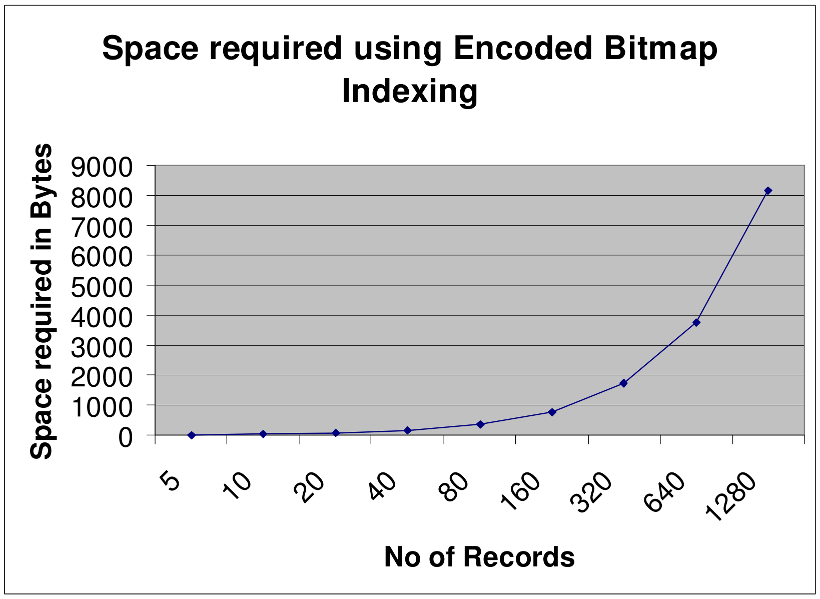


Figure 3.3: Space required for Encoded Bitmap Indexing

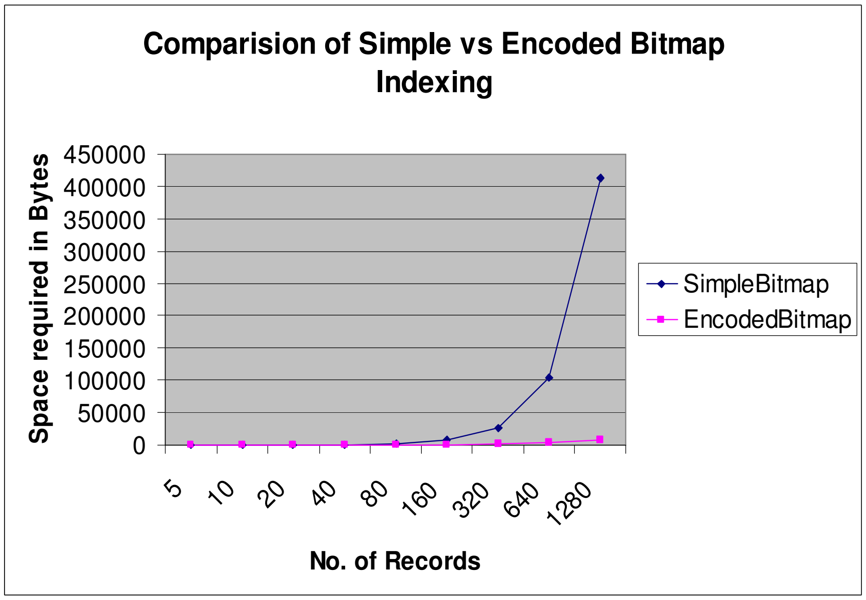


Figure 3.4: Space Comparison for simple and encoded bitmap indexing

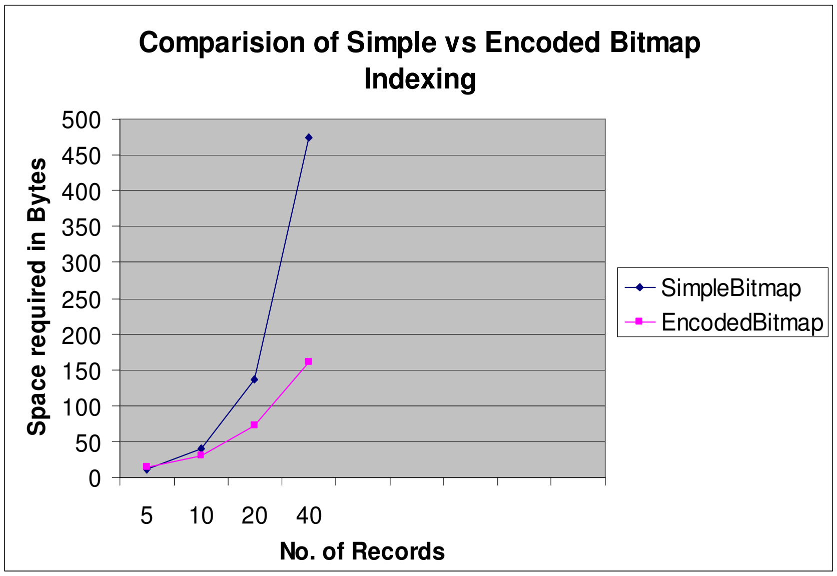


Figure 3.5: Simple and Encoded bitmap indexing analysis of break-even point

Figures 3.2 and 3.3 depict the space required for both types of indexing. Figures 3.4 and 3.5 compare both the indexing techniques in terms of the space required (in bytes). From figure 3.5, we can observe that the break-even point occurred when the record count is around 7. This suggests that, if the record count is less than 7 simple bitmap indexing is more efficient than encoded bitmap indexing as far as the space is concerned and if the record count is above 7, then encoded bitmap indexing is better compared to simple bitmap indexing.

SAMPLE MEASUREMENTS (TIME TAKEN):

Number of records = 1280

Total time required for Encoded Bitmap Indexing in clock ticks 984.00

Total time required for simple bitmap indexing in clock ticks 3344.00

Difference in execution times in clock ticks 2360.00

2360 / CLOCKS_PER_SEC gives the time in seconds

The experiments were conducted on a Pentium 4 machine with 1 Gb RAM running Linux.

Space requirements for comparing Huffman Encoding and Gray Code Encoding will always be same which is equal to [$\log_2$ N], N as the cardinality of an attribute.

3.2.5. New Encoding Scheme

The proposed new encoding scheme is explained through using the following student database:

NAME	AGE	DISCIPLINE	CGPA	Gender
Ajay	19	EEE	6.163266	M
Ram	22	Civil	6.024028	M
.	20	.	9.713227	F
.	22	.	6.031079	M
.	18	.	7.837317	F
.	19	.	4.468817	M
.	22	.	7.211664	M
.	18	.	5.597524	M
.	20	.	4.122381	F
.	21	.	8.157804	M
.	21	.	7.259367	M
.	22	.	8.829899	F
.	20	.	5.219405	M
.	18	.	8.355764	M
.	18	.	6.070122	F
.	19	.	7.835711	M
.	22	.	4.996277	F
.	21	.	5.456334	F
.	21	.	8.319742	M
.	19	.	8.375886	M
Yashveer	19	CS	4.202766	F

Figure 3.6: Sample Student Database

The dataset has five different values for the attribute age i.e.18,19,20,21,22. The bitmap of attribute age is given below:

18: 0000100100000011000000

19: 1000010000000000100011

20: 0010000100010000000000

21: 0000000011000000011000

22: 0101001000010000010000

We have created five bitmap vectors for each distinct value of age. Each bitmap consists of 22 bit values since our example comprises 22 students. For instance, the last bit of the bitmap of age 21 is set to 0 because the last student is not of age 21. Space overhead is clearly arising for high cardinality of data which slows down the query processing. This bottleneck can be brought under control only by bringing to a halt to the increase in number of records or by devising a way by which the encode attribute values to control the space requirements. We would discuss its solution details in the forthcoming section.

3.2.5.1. Implementation Issues and Performance Analysis

To begin with we constructed randomly generated data comprising of student information with "Age" and "Gender" as the indexed columns. After generating the random data we need to look for distinct elements of "Age" and "Gender" in order to process queries dealing with the sample database that have been generated. The bitmap on each of the columns can be generated by constructing bitmap vector for each of the distinct elements of each indexed column.

Query processing is done by asking the user for the student age and/or gender whose information he/she wants to use in his/her decision-making. Now, using AND and OR binary operators, query posed by user is resolved. Both equality and range queries are considered. The time complexity graph for this approach is shown in Figure 3.7.

Method1: Normal implementation by generation of simple bitmaps on corresponding indexed columns. From the above approach we can see how increase in cardinality drastically affects the query processing time. For processing a query over 10,000 records is taking around 25 milliseconds while to process a query over 100,000 records is taking around 250 milliseconds. That is about 10 times more and is the direct effect of increasing cardinality.

To solve this particular overhead we can go for the second approach defined below.

Method2: This method represents the improved implementation that saves time by storing only two positions (start and end) of each of the distinct indexed column entries.

This alternative to "Method1" can save a lot of space. In this improved method we need to first sort the random generated data based on the distinct elements and then store the beginning and ending indexes of each of the distinct elements of a particular record in a data structure and then we can go for query processing. This new method will save a lot of space as we are just storing two indexes for each of the distinct elements of each of the indexed columns and thus improve the query processing time to a large extent.

The output of this approach on a student database with 80,000 records will be as follows:

```
For age = 18 start index = 0 , end index = 15950
For age = 19 start index = 15951 , end index = 31934
For age = 20 start index = 31935 , end index = 48061
For age = 21 start index = 48062 , end index = 64095
For age = 22 start index = 64096 , end index = 79999

For gender = M start index = 0 , end index = 39708
For gender = F start index = 39709 , end index = 79999
```

The above method clearly illustrates the processing done by just storing the first and last positions of each distinct indexed column (age and gender) entries and thus removing the bottleneck. From the above approach we can see how the increase in cardinality is being dealt with efficiently. For processing a query over 10,000 records is taking around 20 milliseconds while to process a query over 100,000 records is taking around 120 milliseconds. That is about 6 times more and is almost half the amount of time taken to process queries with the previous approach (Method1).

The time complexity comparison for both approaches is shown below:

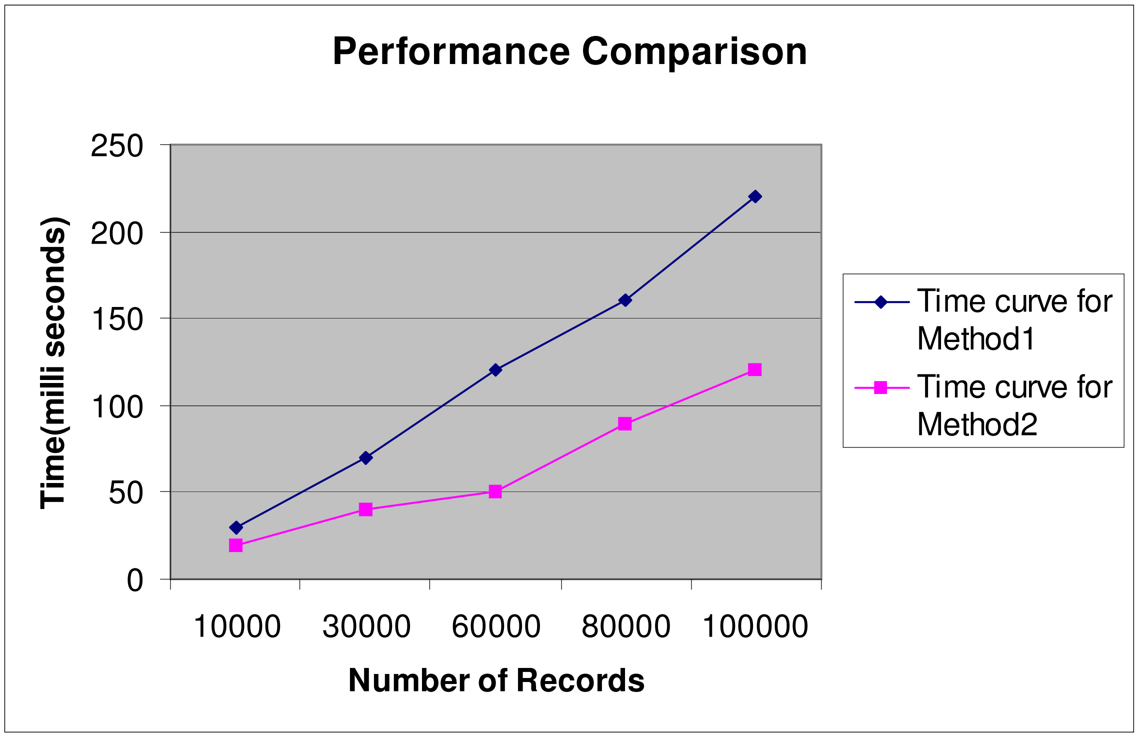


Figure 3.7: Query Processing Time Comparisons for Simple and Encoded Bitmap Indexes

3.3. Compression of Bitmap Indexes

The bitmap index is one of the most promising strategies for indexing high-dimensional data arising in such environments as data warehousing, decision support systems, and scientific databases. One of the first database systems to use such a scheme is a system called Model 204 [O'Neil, 1987]. Most of the major commercial database systems now support some form of a bitmap index. In the research community, the earliest forms of the bitmap indexes are known by such names as bit transposed files. Some research results on signature files are also directly useful to enhancing the effectiveness of bitmap indexes. Recently, a number of optimal bitmap schemes have also been proposed. For high dimensional data, various tests have shown that bitmap schemes are faster than commonly used tree based indexes.

The main advantage for using a compressed bitmap index is to reduce the space requirement. It also reduces the amount of I/O time that is required for long sequential data transfers. The classical bitmap index produces one bitmap for each distinct value of the attribute being indexed. The size of the indexes could be much larger than the size of the dataset. This is especially true for scientific databases where most of the attributes have high cardinality. However, the bitmaps from the bitmap indexes are often very sparse, i.e., they contain mostly zero bits. They are therefore prime candidates for compression. The common compression schemes, such as g-zip and bzip2, aren't designed for compressing bitmap indexes. If a bitmap index is compressed using such a scheme, the query processing usually takes much longer than using the uncompressed index. One solution to this problem is to use specially designed compression schemes. Recently, a number of studies were performed on compression schemes especially designed for bitmap indexes. It is also possible to perform database operations (e.g. aggregation) directly on compressed data, there by potentially reducing CPU time requirements. One of the most promising compressing schemes is the byte-aligned bitmap code (BBC) [Antoshenkov, 1994]. This scheme permits efficient operations without decompression, thereby reducing both the disk space requirement and the memory requirement for performing operations. The question we address in this chapter is whether a compressed bitmap index can outperform its uncompressed counterpart.

We briefly review three well known schemes for representing bitmaps. These three schemes are selected as representatives from a number of schemes studied previously a straightforward way of representing a bitmap is to use one bit of computer memory for each bit of the bitmap. We call this the *literal* (LIT) *bit vector*. This is the uncompressed scheme and logical operations on uncompressed bitmaps are extremely fast. The second type of scheme is the general purpose compression scheme such as gzip. They are highly effective in compressing data files. We use gzip as the representative because it is usually faster than others in decompressing the data files. As mentioned earlier, there are a number of compression schemes that offer good compression and also allow fast bitwise logical operations.

The BBC scheme performs bitwise logical operations efficiently and it compresses almost as well as gzip. BBC scheme is a version of the two-sided BBC. Many of the specialized bitmap compression schemes, including BBC, are based on the basic idea of run-length encoding that represents consecutive identical bits (also called a *fill* or a *gap*) by their bit value and their length. The bit value of a fill is called the fill bit. If the fill bit is zero, we call the fill a *0-fill*, otherwise it is a *1-fill*. Compression schemes generally try to store repeating bit patterns in compact forms. The run length encoding is among the simplest of these schemes. This simplicity allows logical operations to be performed efficiently on the compressed bitmaps. Different run-length encoding schemes commonly differ in their representations of the fill lengths and the short fills. A naive run-length code may use a word to represent all fill lengths. This is ineffective because it uses more space to represent short fills than in the literal scheme. One common improvement is to represent the short fills literally. The second improvement is to use as few bits as possible to represent the fill length.

Given a bit sequence, the BBC scheme first divides it into bytes and then groups the bytes into *runs*. Each BBC run consists of a fill followed by a *tail* of literal bytes. Since a BBC fill always contains a number of whole bytes, it represents the fill length as the number of bytes rather than the number of bits. In addition, it uses a multi-byte scheme to represent the fill lengths. This strategy often uses more bits to represent a fill length than others such as ExpGol. However it allows for faster operations. Another property that is crucial to the efficiency of the BBC scheme is the byte alignment. This property limits a fill length to be an integer multiple of bytes. More importantly, it ensures that during any bitwise logical operation a tail byte is never broken into individual bits. Because working on individual bits is much less efficient than working on whole bytes on most CPUs, byte-alignment is crucial to the operational efficiency of BBC. Removing the alignment may lead to better compression. For example, the ExpGol scheme can compress better than BBC partly because it does not obey the byte alignment. However, bitwise logical operations on ExpGol bit vectors are often much slower than on BBC bit vectors. Most of the known compression schemes are byte based, that is, they access computer memory one byte at a time. On most modern computers, accessing one byte takes as much time as accessing one word. A computer CPU with MMX technology offers the capability of performing a single operation on multiple bytes. This may automatically turn byte accesses into word accesses. However, because the bytes in a compressed bit vector typically have complex dependencies, logical operations implemented in high-level languages are unlikely to take advantage of the MMX technology. Instead of relying on the hardware and compilers, we developed a new scheme that accesses only whole words. It is named the *word-aligned hybrid code* (WAH).The word-aligned hybrid (WAH) code is similar to BBC in that it is a hybrid between the run-length encoding and the literal scheme [Wu et.al, 2001]. Unlike BBC, WAH is much simpler and it stores compressed data in words rather than in bytes. There are two types of words in WAH: *literal* words and *fill* words. In our implementation, we use the most significant bit of a word to distinguish between a literal word (0) and a fill word (1). This choice allows one to easily distinguish a literal word from a fill word without explicitly extracting the bit. The lower bits of a literal word contain the bit values from the bitmap.

The second most significant bit of a fill word is the fill bit and the lower bits store the fill length. WAH imposes the word-alignment requirement on the fills, it requires that all fill lengths be integer multiples of the number of bits in a literal word. The word alignment ensures that logical operation functions only need to access words not bytes or bits

3.4. New Strategy to improve performance

The major concern for solving queries is minimum space requirements for bitmap indexes and minimum response time. As we have seen various bitmap index compression schemes which satisfy these requirements. We propose a new area to be explored regarding the run length encoding of the bitmap indexes. In this newly proposed scheme we will encode only the runs of ‘one’ and taking rest of the bitmap as filled with ‘zeros’.

These run lengths will be represented as starting and ending location (integer format) of runs of ‘ones’ in the bitmap. So in our bitmap index representation we will represent only ‘ones’ as rest of the bits can be taken as default ‘zero’. In this way we can minimize the space requirements to store the bitmap indexes. This scheme if incorporated with the sorted data will drastically minimize the storage requirements. But one should have to be cautious towards the implementation of query response. As it can lead to large overheads in terms of query response and computations involved.

3.4.1. New Approach Adopted

To study and implement we will first go through the schemes which are already introduced or in use. Then we will take random database to simulate a data warehouse after that indexes can be built on the various attributes. These indexes can be compressed using schemes like WAH, BBC etc. and finally by our proposed scheme. Various types of queries can be run based on these compressed indexes and finally the space utilized to store the compressed indexes and query response time can be compared. As WAH and BBC schemes are proven to be best in space utilization and query response we can hope that our scheme can also stand side by side or can be proved better than these schemes.

The approach adopted here, is very straightforward. BBC and WAH provide better results in terms of compression when there are long runs of zeros and ones. In random data, it is very unlikely that there will be long runs. If the column on which the bitmap index is created has high cardinality then the bit vectors would be sparse. But there will not be long runs of ones. To guarantee long runs of both zeros and ones, the column on which the bitmap index is to be created, is sorted. The index created on the sorted column is called the *clustering bitmap index* as it similar to clustering index. Any other bitmap index on the same table will be called a *secondary bitmap index.* The only processing overhead that the proposed strategy has is that the table, on which the clustering bitmap index is to be created, must be sorted. Also we need to decide in advance about the field on which the clustering bitmap index is to be created, as the sorting will be done during the ETL process. This increases the amount of processing during the ETL phase, but the benefits of doing this in terms of saved space and processing time of queries is enormous. If the bitmap indexing strategy has to be changed, it can be done at the next refreshing of the data warehouse, whenever it is scheduled. To summarize, the new strategy has the following issues:

- Sorting of the table on the column on which the *clustering bitmap index* is to be created
- There can be only one clustering bitmap index on a table
- Need to decide in advance (i.e. before loading the data on to the data warehouse) about the bitmap indexing strategy

- Indexing strategy can be changed only during the next refresh of the data warehouse

But fortunately these limitations have no effect on the query performance. In this newly proposed scheme we will encode only the runs of 'one' and taking rest of the bitmap as filled with 'zeros'. These run lengths will be represented as starting and ending location (integer format) of runs of 'ones' in the bitmap. So in our bitmap index representation we will represent only 'ones' as rest of the bits can be taken as default 'zero'. In this way we can minimize the space requirements to store the bitmap indexes. This scheme if incorporated with the sorted data will drastically minimize the storage requirements. But one should have to be cautious towards the implementation of query response. As it can lead to large overheads in terms of query response and computations involved.

3.5. Experimental Work and Results

This section presents experimental results comparing the space-time performance of the proposed bitmap indexing strategy against BBC and WAH. The space-efficiency of an index is in terms of the disk space for storing the index. The time-efficiency of an index is in terms of the processing time taken to answer a query. The processing time includes disk I/O time, CPU time for bitmap operations, and the decompression time for compressed bitmaps.

The experiments were run on a 2.4 GHz Pentium-IV processor with 256 MB RAM running Linux Red Hat 9.1.

All the results presented in the this section are obtained using randomly generated data with table cardinality varying from 5000 to 2.5 million and column cardinality C taking values 5 and 10.

3.5.1. Space Efficiency

The index file size comparisons for increasing table cardinality is done in Figure 3.8 and 3.9 for C = 5 and 10 respectively. First thing to note is that the space requirements increase with C for all compressed indexes. It is clear from the figures that the space requirements for BBC and WAH increase exponentially for increasing table cardinality. It is true for both C = 5 and 10. The most interesting result is that the space required by WAH-sorted (WAH index created on a sorted column) is constant for increasing number of records and it requires only 125 bytes of space for C = 5 and 250 bytes for C = 10. The space required by WAH-sorted increases linearly with C. This means that even for very large tables the bitmaps can be stored using just a few bytes using WAH-sorted.

BBC-sorted (BBC index created on a sorted column) takes only a very small fraction of the space required by BBC but it increases with the increase in number of records. It is therefore recommended that the field on which bitmap index is to be created, should be sorted and WAH should be used to get maximum compression.

3.5.2. Performance Efficiency

Equality and range queries are considered for testing the performance of the proposed strategy. The results are presented graphically comparing the effect of sorting on the performance of BBC and WAH for C = 5 and 10. The results for equality queries BBC and WAH are presented in Figures 3 and 4 respectively. Figure 3.8 and 3.9, it is clear that the BBC-sorted performs better than BBC for both C = 5 and C = 10. The performance benefit of using BBC-sorted is more pronounced for C = 10 and it increases with the increase in number of records. BBC-sorted, on an average, takes up to less than 20% time for C = 5 than BBC. This figure goes up to 60% for C = 10.

In case of WAH-sorted also the performance gains are more for C = 10 and increases with the increase in number of records as can be seen from Figure 3.9. WAH-sorted, on an average, takes up to less than 50% time for C = 5 than WAH. This figure, as expected, goes up to 64% for C = 10.

Results for range queries for WAH are presented in Figure 3.12. The type of range queries considered has the general form:

*select * from table where X < 1*

If X takes values 0, 1, 2, 3, and 4, then the OR operation is carried out between the bit vectors for X = 0 and X = 1 to answer the query. Figure 2.12 shows the advantages of using WAH-sorted are clear. It is faster than WAH for both values of column cardinalities considered. For C = 5, WAH-sorted takes, on an average, 27% less time than WAH for number of records more than 10000, whereas for C = 10, it is 34% faster.

From the results presented in this section, it is clear that BBC-sorted and WAH-sorted offer better compression and performance than BBC and WAH respectively. This is true for all combinations of table and column cardinalities and for both equality and range queries.

Figure 3.8: Index File Size for C = 5

Figure 3.9: Index File Size for C = 10

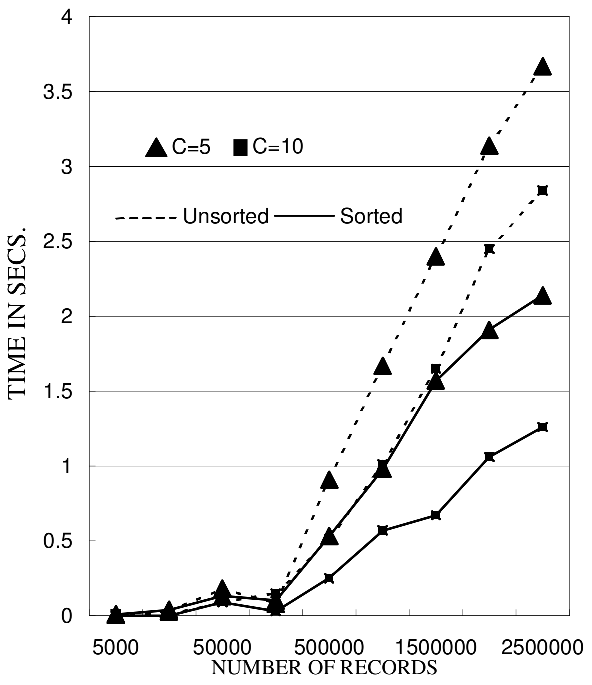


Figure 3.10: Performance of BBC-sorted for Equality Queries

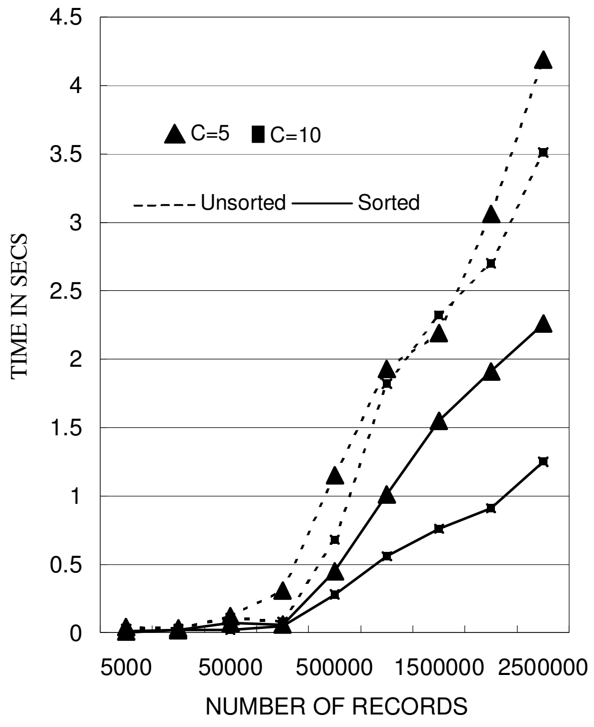


Figure 3.11: Performance of WAH-sorted for Equality Queries

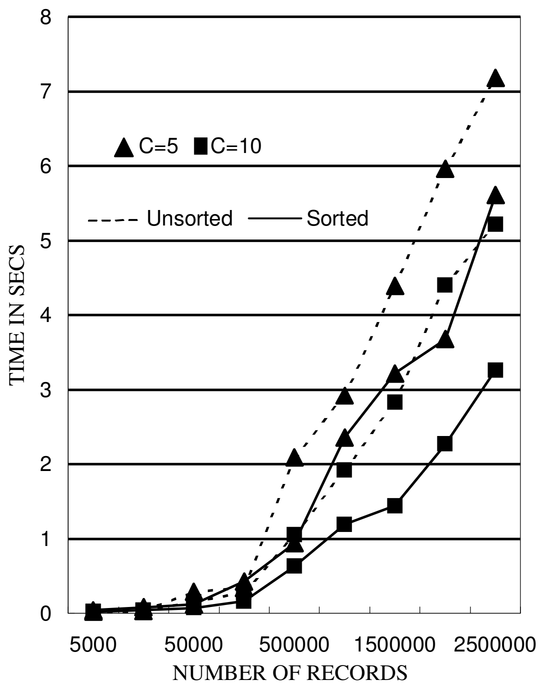


Figure 3.12: Performance of WAH-sorted for Range Queries

3.6. Contributions and Summary

In this chapter, we have discussed about simple and encoded bitmap indexes, how to solve queries using these techniques, performance analysis of both these techniques and compressing simple bitmap indexes on sorted fields. We have given a performance analysis of both simple and encoded bitmap indexes using graphical approach. The result is satisfactory and shows that as the cardinality and the range of selections increase, encoded bitmap indexes perform and more stable than simple bitmap indexes. However, still some implementation-oriented problems to be solved. Also, an efficient algorithm for logical reduction of the boolean retrieval functions is indispensable, if we want to achieve a better performance of the encoded bitmap indexes. As part of future work, one can look into the compression aspects of encoded bitmap indexes.

A new bitmap indexing strategy for speeding up queries in a data warehouse has been proposed. The strategy is to sort the column on which a bitmap index is to be created and then using BBC and WAH compression schemes. Sorting produces long runs of zeros and ones and this gives higher degree of compression for both BBC and WAH. Moreover, the response time of queries is found to decrease considerably for both equality and range queries. It is found that in some cases BBC-sorted is up to 60% faster than BBC and WAH-sorted is up to 64% faster than WAH. With such considerable gains in terms of space saved and performance, the new strategy offers a simple yet effective solution to query performance challenges in a data warehousing environment.

To sum up, we can say that simple bitmap indexing works well with low cardinalities attributes. To deal with high cardinalities, our new approach makes simple bitmap indexes suitable. Compressed bitmap indexing is shown as a promising technique to overcome space problem. Also, there are several fast algorithms for evaluating boolean operators on compressed bitmaps are available, which can be examined. Another issue is that we also need some other efficient encoding techniques to lower the number of logical operations.

Chapter 4: Multi-Component Encoding and Data Reorganization

4.1. Introduction

Designing efficient bitmap schemes for storage and retrieval of massive scientific data is a challenging problem. Large volumes of data have been generated in scientific experiments including biology, high-energy physics, astrophysics, and climate modeling. Querying such huge amounts of data is becoming increasingly difficult. Being an effective way to store the synopsis of the original data, bitmap index is a particularly promising strategy for accessing these types of data efficiently. However, the size of bitmap index is still large. Run length encoding and its lossless compression variants have been applied to further compress the bitmap indexes. Along with Word-Aligned Hybrid code (WAH), Gray code ordering of the data tuples has already been shown to greatly boost the compression ratio. For both conventional and scientific databases, the number of tuples in the database will far exceed the number of attributes for the tuple. Every tuple in the database is represented by one row of the bitmap index whereas each column of the bitmap is generated by classifying every tuple attribute into a few categories. Thus, the bitmap index generally has much higher order of rows than the number of columns.

Bitmap index used in database indexing is a special kind of bit matrix. Each binary row vector in the bitmap represents one tuple in the database. It is usually generated by quantizing the attributes of the tuples. The quantization process proceeds in two steps. First, many categories are produced by limiting the possible values of each attribute. Next, the tuple data are encoded according to the attribute category to which it belongs. Bitmap Compression may not be enough for the enormous data generated in some applications such as high-energy physics. We are concerned with rearranging the order in which data is stored in a database so as to maximize the amount of compression. To improve the compression rates, reorganization of bitmap tables is studied where tuple reordering problem has been introduced and Gray code ordering algorithm has been proposed [Pinar et al., 2005]. In this chapter, we study how to reduce the number of columns for binned bitmap tables to improve better ordering and improve the space complexity for tuple reordering problem. In tuple reordering problem, we found that for lesser number of columns, Gray code ordering gives better compression rates. We applied Multi-Component indexing technique to reduce the number of columns, which is a new idea to apply on binned bitmap tables.

In this chapter, we discuss bitmap compression indexes using multi-component indexing for the efficient storage and fast retrieval of large data. The bitmap compression indexes embedded multi-component shows superiority over bitmap compressed indexes. Tuple reordering problem is studied to reorganize database tuples for optimal compression rates. Gray Code ordering algorithm is also used which runs in linear time in the order of the size of the database. Reduction in the number of columns is observed in our study when multi-component indexing on the binned data is applied. An improvement in the space requirement for bitmap index by 25% is observed when one time component indexing is applied. Satisfactory improvement factor is observed when gray code ordering and WAH compression technique is used. Due to processing overhead, two component index is used. Our experimental results on real data sets show that the compression ratio can be improved by a factor of 2 to 8

4.2. Tuple Reordering Problem

Our objective in reordering is to increase the performance of run-length encoding by having longer uniform segments and thus fewer number of blocks. Recall that run-length encoding, when used on bitmaps, packs each segment of "1"s into a block and stores a pointer to each block together with the length of the block. Thus the storage size is determined by the number of such blocks. Consider two consecutive tuples in the bitmap table. If the tuples are on the same bin for an attribute, they will be packed to the same block. If not, a new block should start. Efficiency can be enhanced by reordering tuples so that they fall into the same bins as much as possible. An example is illustrated in Figure 1. In this example, the original table has 12 blocks, whereas the reordered table requires only 7 blocks. Let diff(t_i, t_j) be the number of attributes that tuple t_i and tuple t_j fall in different bins. Notice that diff(π_i, π_{i+1}) gives how many new blocks start at the ith tuple after reordering when run-length encoding is used, where π_i denotes the i^{th} tuple in ordering π [Pinar et al.,2005].

Definition 1 (Tuple reordering problem) *Let π be an ordering of m tuples so that π_i denotes the ith tuple in the ordering. Tuple reordering problem is finding an ordering π that minimizes*

$$\sum_{i=1}^{m-1} diff(\pi_i, \pi_{i+1}) \tag{4.1}$$

In equation (4.1), we sum diff values over all consecutive tuples to attain the number of new runs that start for the whole table. The first tuple requires starting a run for each attribute. Thus the number of blocks can be computed as A + $\sum_{i=1}^{m-1} diff(\pi_i, \pi_{i+1})$, where A is the number of attributes. Thus an ordering that minimizes equation (1) also minimizes the number of blocks in the reordered table. For instance, equation (1) returns 2 + 2 + 2 + 1 + 2 = 9 for the initial ordering in Figure 4.1, which means with the addition of the number of attributes, there will be 9+3 = 12 blocks in the compressed table. Whereas for the reordered table in the same figure, Equation 1 returns 0+1+1+1+1= 4, which means only 7 blocks in the compressed file.

1	0	1	0	0	0	t_1
0	1	0	1	0	1	t_2
1	0	0	1	1	0	t_3
1	0	1	0	0	1	t_4
1	0	1	0	1	0	t_5
0	1	0	1	1	0	t_6

(a) Original Table

t_1	1	0	1	0	0	1
t_4	1	0	1	0	0	1
t_5	1	0	1	0	1	0
t_3	1	0	0	1	1	1
t_6	0	1	0	1	1	0
t_2	0	1	0	1	0	1

(b) Reordered Table

Figure 4.1: Tuple Reordering

4.3. Gray Code Encoding

A Gray code is an encoding of numbers so that adjacent numbers have only a single digit differing by 1. For binary numbers two adjacent numbers differ only by one digit.

For instance (000, 001, 011, 010, 110, 111, 101, 100) is a binary 3-bit Gray code. An n-bit Gray code corresponds to a Hamiltonian cycle on an n-dimensional Hamming space. From another perspective, it is a kind of Space-Filling Curve (SFC) in Hamming space, where a space-filling curve is a mapping from a one dimensional set to a multi-dimensional set. Gray code is analogous to a binary version of Hilbert SFC, because both are optimal in minimizing the changes between adjacent points [Liao et al., 2001, Mokbel and Aref, 2001]. It is worth noting that Gray code is not unique. After a certain transformation, such as the cyclic shift of the entries or permutation of the bits, the code is still Gray code. Binary Gray code is often referred to as the "reflected" code, because it can be generated by the reflection technique described below.

1. Let S = (s_1, s_2, . . . , s_n) be a Gray code.
2. First write S forwards and then append the same code S by writing it backwards, so that we have (s_1, s_2, . . . , s_n, s_n, . . . ,s_2, s_1).
3. Append 0 at the beginning of the first n numbers, and 1 at the beginning of the last n numbers.

As an example, take the Gray code (0, 1). Write it forwards, and then add the same sequence backwards, and we get (0, 1, 1, 0). Then we add 0's and 1's to get (00, 01, 11, 10). We can use this new sequence as an input to our algorithm. After the reflection step we get (00, 01, 11, 10, 10, 11, 01, 10). We add the first digits to attain: (000, 001, 011, 010, 110, 111, 101, 100). It is worth noting that Gray codes are not unique, and different orders on the same group of numbers might satisfy the Gray code property. We use the term fundamental Gray code to refer to a Gray code generated by the reflection technique described above with using (0, 1) as the initial sequence. We will refer to ordering a set of numbers with respect to the fundamental Gray codes as Gray code ordering, which we describe next.

Definition 2 (Gray code rank) *The Gray code rank g(s) of an n-bit binary number s is the rank of this number in an n-bit fundamental Gray code.* For instance, g(0000) = 1, since it is the first number in the 4-bit fundamental Gray code. And g(0001) = 2, since it follows 0000 in the fundamental Gray code.

Definition 3 (Gray code sorting) A sequence S = (s_1, s_2, . . . , s_m) is Gray code sorted iff $g(s_i) \leq g(s_{i+1})$ for i = 1, 2, …..m - 1, where g(s_i) refers to the Gray code rank of s_i.

The sequence (0001, 0010, 0101, 1100, 1110, 1011) is Gray code sorted because g(0001) = 2 < g(0010) = 4 < g(0101) = 7 < g(1100) = 9 < g(1110) = 12 < g(1011) = 14.

This brings the question of how to efficiently order a set of numbers to be Gray code sorted. We can reverse the fundamental Gray code generation process, to sort numbers with respect to the fundamental Gray code. As the first step, we can divide numbers as those that start with 0 and those that start with 1. Clearly those that start with 0 will precede others in the ordering. Then we can recursively order those that start with 0. The same can be applied to the second group of numbers that start with 1, but we need to reverse their ordering due to the reflective property of the Gray code. Gray Code sorting algorithm is explained in reference [Pinar et al., 2005].

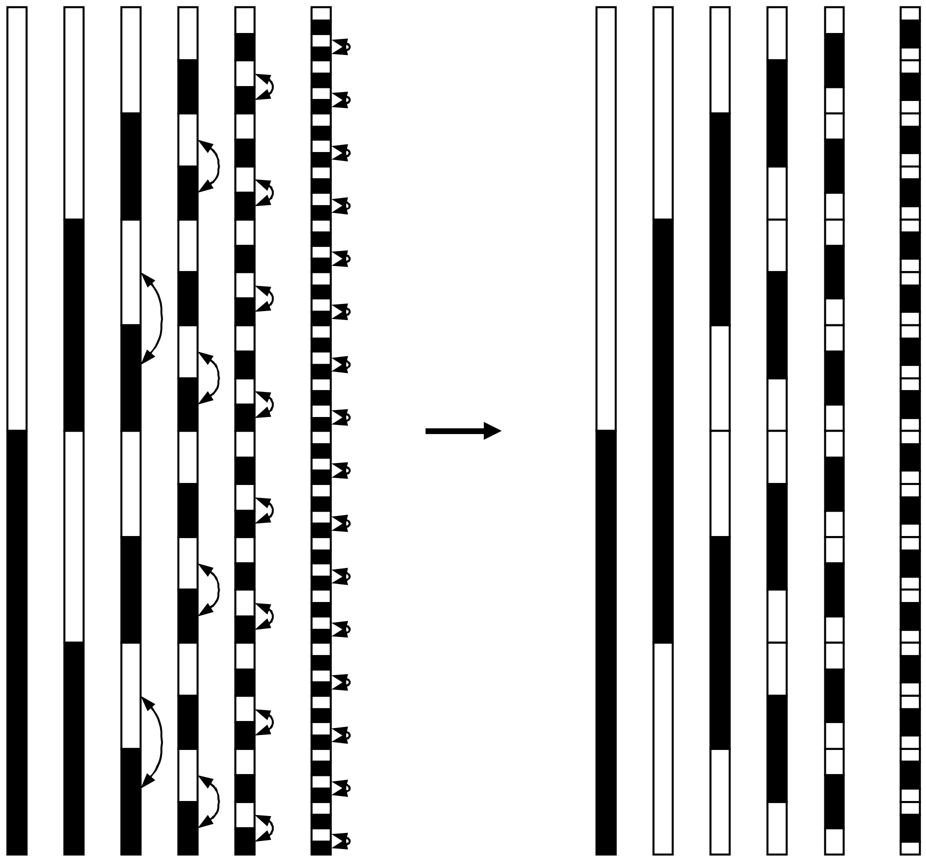


Figure 4.2: Gray code ordering algorithm.

In Figure 4.2, each bar represents one column in the bitmap. Shown are the first 6 columns of the bitmap. The white portion of the bar represents the continued sequence of 0s whereas the black portion represents the continued sequence of 1s. As mentioned earlier, Gray code ordering is essentially a numerical ordering with reflecting or reversing. Depicted graphically, the reflecting or reversing operation is to flip a certain segment with two portions: one white and one black. The part (a) in the figure demonstrates the ordering before flipping or simply the ordinary numerical sorting whereas the part (b) in the figure depicts the result of flipping or the outcome of Gray code ordering. It is easy to see that many runs concatenate to form longer runs after flipping. This explains why Gray code ordering is more effective than the numerical/lexicographic ordering.

4.4. Multi-Component Encoding

Simple bitmap indexes take huge amount of space for high cardinality data since we need to make bitmap vector for each distinct value. By using Multi-component Indexes, number of bitmap vectors can be reduced. The general idea behind multi-component index is to perform the Attribute Value Decomposition [Chan and Ioannidis, 1999]. One value can be decomposed into several components with same base or different bases. Instead of representing bitmap with single table, the same values can be represented with several smaller bitmaps working together. Let C be the attribute cardinality, which means the number of actual values that an attribute can have. Then you can create a bitmap Index in the following way:

Consider an attribute value *v* and a sequence of (*n*-1) numbers $\langle b_{n-1}, b_{n-2}, ..., b_1 \rangle$.

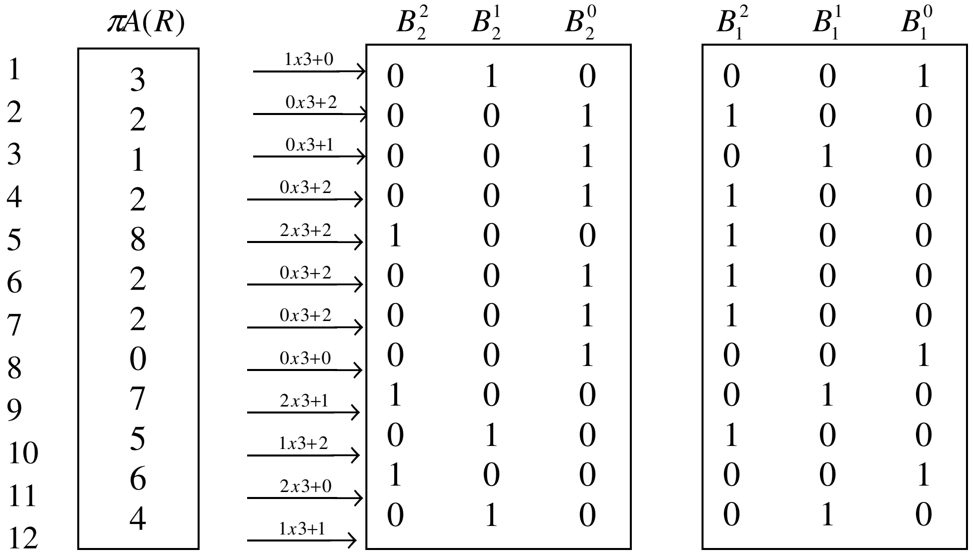
Let us define $b_n = \left\lceil \frac{C}{\prod_{i=1}^{n-1} b_i} \right\rceil$.

Then *v* can be decomposed into a sequence of *n* digits $\langle v_n, v_{n-1}, \cdots, v_1 \rangle$ as follows:

$$
\begin{aligned}
v &= V_1 \\
&= V_2 b_1 + v_1 \\
&= V_3(b_2 b_1) + v_2 b_1 + v_1 \\
&= V_4\,(b_4 b_2 b_1) + v_3\,(b_2 b_1) + v_2 b_1 + v_1 \\
&= \ldots\ldots\ldots\ldots
\end{aligned}
$$

$$= v_n\left(\prod_{j=1}^{n-1} b_j\right) + \ldots + v_i\left(\prod_{j=1}^{i-1} b_j\right) + \ldots + v_2 b_1 + v_1$$

where $v_i = V_i \bmod b_i$, $V_i = \left\lfloor \frac{V_{i-1}}{b_{i-1}} \right\rfloor$, 1 <i <n range 0 = $v_i < b_i$. Based on the above, each choice of *n* and sequence $<b_{n-1}, b_{n-2}, \ldots, b_1>$.gives a different representation of attribute values, and therefore a different index. An index is well-defined if $b_i \geq 2, 1 \leq i \leq n$. The sequence $<b_{n-1}, b_{n-2}, \ldots, b_1>$ is the base of the index, which is in turn called a base- $<b_{n-1}, b_{n-2}, \ldots, b_1>$ index. If $b_n = b_{n-1} = \ldots = b_1 \equiv b$, then the base is called uniform and the index is called base-b for short. The index consists of *n* components, i.e., one component per digit. Each component individually is now a collection of bitmaps, constituting essentially a base b_j index. As you can see v_i has to be smaller than b_i, which means they have to be consecutive integer values with a range from 0 to C - 1. If this is not the case, then the index can either be built on the entire domain of present values making it generally much larger, or the values can be mapped on C consecutive values via a lookup table. The sequence $<b_{n-1}, b_{n-2}, \ldots, b_1>$ is the base of the index, which in turn is called a base $<b_{n-1}, b_{n-2}, \ldots, b_1>$ index. The index consists of *n* components, which means that each choice of *n* and a sequence $<b_{n-1}, b_{n-2}, \ldots, b_1>$ gives a different representation of the index because each component individually is now a collection of bitmaps.



	$\pi A(R)$		B_2^2	B_2^1	B_2^0	B_1^2	B_1^1	B_1^0
1	3	$\xrightarrow{1x3+0}$	0	1	0	0	0	1
2	2	$\xrightarrow{0x3+2}$	0	0	1	1	0	0
3	1	$\xrightarrow{0x3+1}$	0	0	1	0	1	0
4	2	$\xrightarrow{0x3+2}$	0	0	1	1	0	0
5	8	$\xrightarrow{2x3+2}$	1	0	0	1	0	0
6	2	$\xrightarrow{0x3+2}$	0	0	1	1	0	0
7	2	$\xrightarrow{0x3+2}$	0	0	1	1	0	0
8	0	$\xrightarrow{0x3+0}$	0	0	1	0	0	1
9	7	$\xrightarrow{2x3+1}$	1	0	0	0	1	0
10	5	$\xrightarrow{1x3+2}$	0	1	0	1	0	0
11	6	$\xrightarrow{2x3+0}$	1	0	0	0	0	1
12	4	$\xrightarrow{1x3+1}$	0	1	0	0	1	0

Figure 4.3: Example of a 2-Component index with base < 3, 3 >

Let n_i denote the number of bitmaps in the i^{th} component of an index and $\{B_i^{n_{i-1}}, B_i^{n_{i-2}}, \ldots, B_i^0\}$ denote the collection of n_i bitmaps that form the i^{th} component. Let's give an example on this to make things clear. We take the same 12-record relation used in Figure 1 and transform it in a base-< 3; 3 > Value- List index (Figure 2). By doing this, the number of bitmaps has been reduced from 9 to 6. If we now want to calculate the actual value of a record, we use the appropriate line from the calculation shown above. Since the example uses a 2-component index, the formula to use is $V_2b_1 + v_1$; V_2 is then replaced by the number in the first component which contains the "1" denoting the actual value, in this case the one from B_1^2, which is the bitmap in the middle of the component. This value is then multiplied with the base b_1, resulting in a "3", and added to v_1 determined in the same way, in this case a "0". So the final outcome of the calculation and the value stored in the first record is a "3".

High cardinality data is divided into number of bins to reduce the number the bitmap vectors for each attribute. The basic idea of binning is to build a bitmap for a *bin* rather than each *distinct attribute value*. This strategy disassociates the number of bitmaps from the attribute cardinality and allows one to build a bitmap index of a prescribed size, no matter how large the attribute cardinality is. A clear advantage of this approach is that it allows one to control the index size. If a value falls into a bin, this bin is marked "1" otherwise "0". Since a value can only fall into a single bin, only a single "1" can exist for each row of each attribute. After binning, the whole database is converted into a huge 0-1 bitmap, where rows correspond to tuples and columns correspond to bin. Table 1 shows a binning example with three attributes, each partitioned into two bins. The first tuple t_1 falls into the first bins in the attributes 1 and 2, and the second bin in attribute 3. Note that after binning we can treat each tuple as a binary number. For instance t_1 = 101001 and t_2 = 010101.

Tuple	**Attribute 1**		**Attribute 2**		**Attribute 3**	
	bin1	**bin2**	**bin1**	**bin2**	**bin1**	**bin2**
t1	1	0	1	0	0	1
t2	0	1	0	1	0	1
t3	1	0	0	1	1	0
t4	1	0	1	0	0	1
t5	1	0	1	0	1	0
t6	0	1	0	1	1	0

Table 4.1: Binning Example

WAH Compression scheme is a variation on the run-length code. The essence of the run-length code is to represent a list of consecutive identical bits by its length and its bit value. In 32-bit implementation, the Leftmost Bit (LMB) of a word is used to distinguish between a literal word and a fill word, where 0 indicates a literal word and 1 indicates a fill word. The lower 31 bits of a literal word contains literal bit values. The second leftmost bit of a fill word is the fill bit and the 30 lower bits store the fill length. To achieve fast operation, it is crucial to impose the word-alignment requirement on this scheme. The word-alignment requirement in WAH requires all fill lengths to be integer multiples of 31 bits (i.e., literal word size). Given this restriction, we represent fill lengths in multiples of literal word size. For example, if a fill contains 62 bits, the fill length will be recorded as two (2) [Wu, et al, 2001, Jhonson, 1999], see Figure 4.2.

a) Input bitmap with 124 bits
100000000………………..11111
124 bits 1, 20*0, 3*1, 79*0, 21*1

b) Group bits into 4 31-bit groups
1,20*0,3*1,7*0 31*0 31*0 10*0,21*1

c) Merge neighboring groups with identical bits
1,20*0,3*1,7*0 62*0 10*0,21*1

d) Encode each group using one word
0100 … 0011 100…. 10000…10 0000…1111..
31 literal bits run length is 2
Fill bit 0
Bit 0 indicates “tail” word Bit 1 indicates “fill” word

Input bit vector	100000000………………..11111 (124 bits)		
124 bits	1, 20*0, 3*1, 79*0, 21*1		
31-bit groups	1,20*0,3*1,7*0	62*0	10*0,21*1
Groups in hex	40000380	00000000 00000000	001FFFFF
WAH(hex)	40000380	80000002	001FFFFF

Table 4.2: WAH bit vector

4.5. Experimental Work and Results

Basic format of input file

Run length encoding and it various form segments of a sequence, thus their performances depend directly on the longer data segments. Thus the data is stored in a compressed manner in a file which ultimately depicts a 0-1 matrix in the bitmap table.

The following code will generate the bitmap table given the file where the info is stored in compressed manner.

Input :

File name "input" which has the following information

6 6 18

0 3 6 9 12 15 18

0 2 5 1 3 5 0 3 4 0 2 5 0 2 4 1 3 4

The first row:

6 6 18 represents the number of rows, number of columns and number of non-zero elements, respectively.

The second row

0 3 6 9 12 15 18 represents the starting positions of each column in the bitmap (third row). Basically, this line means from [0,3) is for first row of the matrix, [3,6) second row, [6,9) third row, [9,12) forth row and so on.

The third row

0 2 5 1 3 5 0 3 4 0 2 5 0 2 4 1 3 4 represent the column number of those non-zero elements.

Output :

A 0-1 bitmap table based on the information present in the input file.

```
1  0  1  0  0  1
0  1  0  1  0  1
1  0  0  1  1  0
1  0  1  0  0  1
1  0  1  0  1  0
0  1  0  1  1  0
```

In this section we present the results of our experiments on six datasets. The experiments were conducted on a Pentium 4 machine with 1 Gb RAM running Linux. We compare here three preprocessing schemes on bitmap data, before actually applying WAH Compression algorithm. Since, the warehoused data is read mostly, the time involved in preprocessing is not a major concern here. We took the uncompressed data with bin size 8, in the first scheme, applied multi-component indexing, thereby reducing the bin size from 8 to 6. In the second scheme, we applied multi-component indexing twice, in a single step, reducing the bin size from 8 to 3. And in third scheme, we applied multi-component indexing (as in scheme 1), but saved the two components in two different files, in order to see the effect on Compression algorithm. We have collected the results by looking at the improvement in storage of datasets, which is how much compression is achieved. The Improvement Factor (IF) is defined as

$$Improvement\ Factor = \frac{Original\ Size}{Compressed\ size}$$

That is the ratio of original file size to compressed file size. The Improvement Factor achieved by applying multi component indexing is a straight forward 1.33 for multi component indexing, as initially, the bin size is initially 8 while multi component indexing reduces it to 6. The case for double multi component indexing is 2.66, as it reduces bin size to 3. This is clearly visible in the graph in figure 4.4.

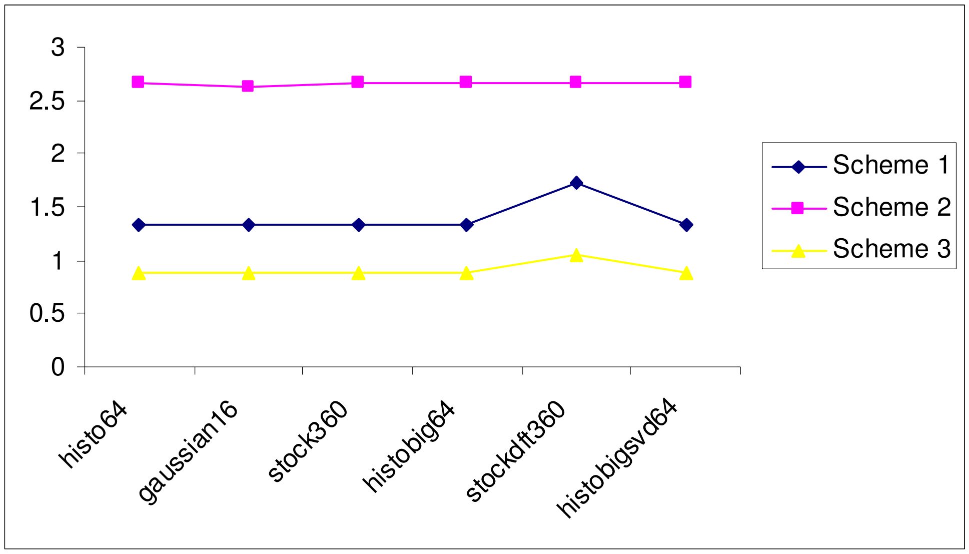


Figure 4.4: Double Multi-Component on all three schemes

The second set of experiments deals with the performance achieved by WAH Compression algorithm when either of schemes one, two or three is used as a preprocessing steps before applying WAH.

It is to be noted here that these improvement factors have been calculated using the file size obtained by applying WAH on dataset and by applying one of the schemes and then applying WAH to it. This allows for comparison of the effect of each of the schemes on the WAH compression. The second scheme out performs the other two, which is obvious. The more interesting point in these results is that storing in two different files doesn't affect the compression of WAH.

Name	Uncompressed size (bytes) Original	Uncompressed size (bytes) MultiComponent	Compressed size (bytes) Original	Compressed size (bytes) Reordered	Improvement Factor
histo64	6208839	4659655	267399	135558	1.97258
gaussian16	12900000	9700000	2786733	1965789	1.417616
stock360	18726500	14046500	1448424	196461	7.372578
histobig64	57641193	43258985	1935619	497493	3.890746
stockdft360	18726500	10858304	1172268	612837	1.912854
histobigsvd64	57641193	43258985	4011647	1996345	2.009496

Table 4.3: Improvement in compression of real data sets

Table 4.3 reveals the effectiveness of our approach on six data sets from various applications. In this table, the first two columns give sizes of the uncompressed bitmap tables for the original and multi-componented data and next two columns give sizes of the compressed bitmap tables for the original and multi-componented data. The last column presents the improvement factor. The data set, histogram, comes from an image database with 112,361 images. Images are collected from a commercial CD-ROM and 64-dimensional color histograms are computed as feature vectors. The data set, stock, is a time- series data which contains 360 days stock price movements of 6500 companies, i.e., 6500 data points with dimensionality 360. The data set histogram is partially correlated, whereas the stock data set is highly correlated.

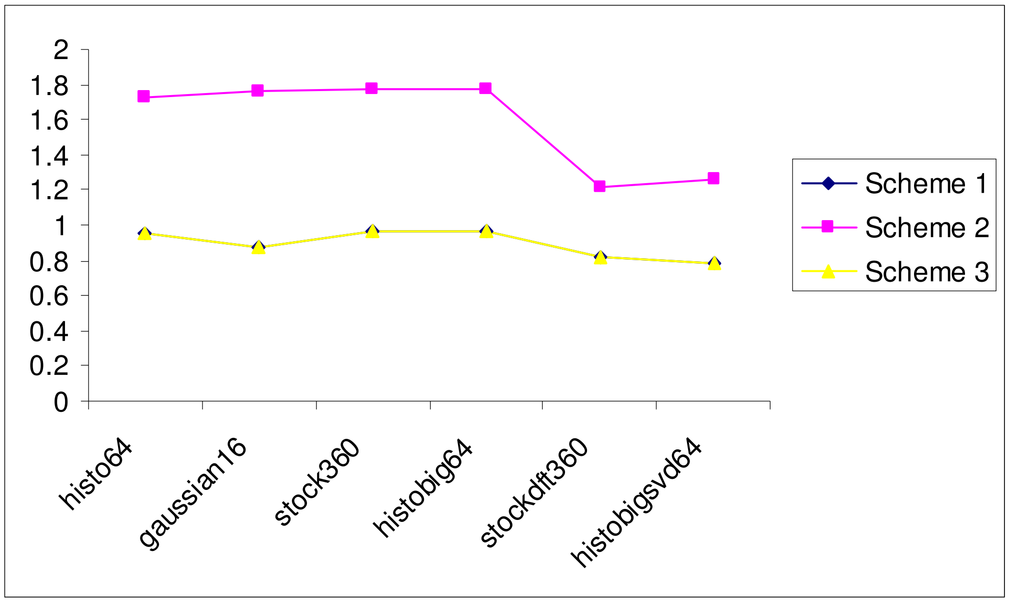


Figure 4.5: Effect of Preprocessing Schemes without WAH compression

The third set of experiments deals with the effect of the three preprocessing schemes on re-ordering tuples before applying the compression algorithm. As with results above, the improvement factors here are calculated with respect to running the re-ordering and compression algorithms directly on datasets. Ignoring the obvious good results of Scheme 2, in the figure 4.5 clearly shows Scheme 3 out performs Scheme 1. This is because the reordering algorithm is better able to align the tuples for better compression.

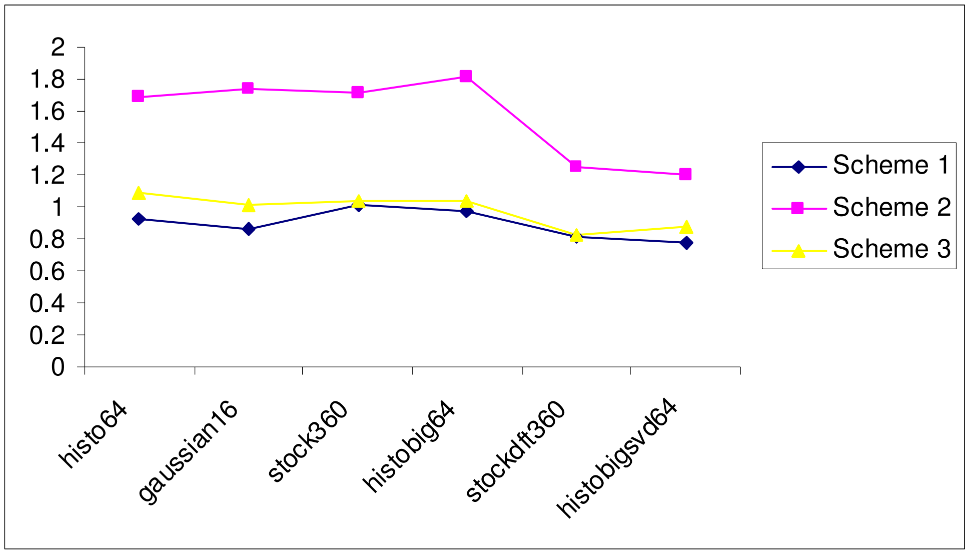


Figure 4.6: Effect of Preprocessing Schemes with WAH compression

From the experiments, it is clear that multi component indexing improves the performance of compression algorithms providing for better storage of warehoused bitmap data. Choosing the base for multi component indexing is critical, as shown by results of Scheme 2. And also, it was established that storing components in different files improves storage, if tuple re-ordering is done before the actual compression.

4.6. Contributions and Summary

We studied tuple reordering problem to improve bitmap compression rate for large datasets. WAH is indeed a very efficient compression method for bitmap indexes.We applied multi-component indexing to get maximum benefits of Gray Code sorting algorithm, which is an in-place algorithm and runs in linear time in the order of the size of the database. It has been found that efficiency of gray code can further be improved by using hybrid indexing technique, consisting of both gray code and component indexing Multi component indexing improves the performance of compression algorithms providing for better storage of warehoused bitmap data. An improvement in the space requirement for bitmap index by 25% is observed when one time component indexing is applied. Satisfactory improvement factor is observed when gray code ordering and WAH compression technique is used. And also, it was established that storing components in different files improves storage, if tuple re-ordering is done before the actual compression. Choosing the base for multi component indexing is critical and thus finding a good base that maximizes the performance of WAH will be another interesting research project.

Chapter 5: Binning Strategies and Algorithms

5.1. Introduction

Bitmap indexes are known to be both space and time efficient for low cardinality data, but for high cardinality and high dimensionality data, the associated space overheads tend to negate the performance benefits. In this chapter, we propose methods that make bitmap indexes suitable for high cardinality data. A common approach for reducing the space requirements of bitmap indexes for high-cardinality attributes is *binning* [Yu and Wu, 1998]. This technique partitions the attribute values into a number of ranges, called *bins*, and uses bitmap vectors to represent bins (attribute ranges) rather than the distinct attribute values. Binning definitely reduces the space requirements, but have an adverse affect on query performance. In case of binning, answering queries may require an additional step called *candidate check* [Stockinger, 2000] which requires reading data from the disk thus increasing the disk I/Os. It is found that candidate checks usually dominate the total query processing time. In order to have acceptable query response times, it is critical to minimize the number of candidate checks for a given set of queries.

In this chapter, we propose a binning strategy for bitmap indexes for high cardinality attributes. The idea is to do binning in such a way so as to minimize the number of candidate checks. Attempts have been made towards finding a solution to the problem of placing the bin boundaries optimally to reduce the overall number of disk reads thereby improving the response time with minimal increase in storage overheads. The main feature of the proposed approach is that it takes into consideration the query access patterns and also the data distribution. A new concept of overlapping and exact bins is introduced to compensate for the marginally increased space requirements in terms of improved query response times. In the following section we describe this new approach to binning. We also describe in detail the algorithms developed to answer queries.

The concept of overlapping bins, in a very primitive way, was first introduced by [Stockinger et al., 2000] and it was meant for answering one-sided range queries. We have introduced range binning for two sided range queries as an extension of one sided range queries and the algorithm for the same is presented. The concept of exact match binning has been introduced to improve the query processing time and minimize the candidate check problem. Another feature of the approach is that it allows bins to overlap. Although number of bins increases but it reduce CPU processing time sharply as less number of candidate checks need to be performed.

5.2. Candidate Check Problem

The basic bitmap index uses every distinct value of the indexed attribute as a key, and generates one bitmap containing as many bits as the number of records in the dataset for each key. The sizes of these basic bitmap indexes are relatively small for low-cardinality attributes, such as “gender”, “types of houses sold in the San Francisco Bay Area”, or “car models produced by Ferrari.” However, for high-cardinality attributes such as “temperature values in a supernova explosion”, the index sizes may be too large to be of any practical use. In this case, bitmap indexes are often designed with bins. This bitmap index strategy partitions the attribute values into a number of ranges, called bins, and uses bitmap vectors to represent bins (attribute ranges) rather than distinct values.
This strategy disassociates the number of bitmaps from the attribute cardinality and allows one to build a bitmap index of a prescribed size, no matter how large the attribute cardinality is. A clear advantage of this approach is that it allows one to control the index size. However, it also introduces some uncertainty in the answers if one only uses the index. To generate precise answers, one may need to examine the original data records (candidates) to verify that the user specified conditions are satisfied. The process of reading the base data to verify the query conditions is called ***candidate check*** [Stockinger et al., 2004, Rotem et al., 2005b].

▓ Edge bin ░ Internal bin

Record ID	Original Values	0-10	11-20	21-30	31-40	41-50
1	5	1	0	0	0	0
2	34	0	0	0	1	0
3	23	0	0	1	0	0
4	9	1	0	0	0	0
5	12	0	1	0	0	0
6	6	1	0	0	0	0
7	34	0	0	0	1	0
8	42	0	0	0	0	1
9	11	0	1	0	0	0
10	22	0	0	1	0	0
11	44	0	0	0	0	1
12	23	0	0	1	0	0
13	18	0	0	0	0	1
14	41	0	1	0	0	0
15	39	0	0	0	1	0

$\longleftrightarrow$

8 < A < 37

Figure 5.1: Two- sided range query 8 < A < 37 on a bitmap index with binning

Here, we are focusing on aggregation queries that are common in data warehousing and scientific applications. These types of queries do not return result records but rather statistical information on the result set, e.g. compute the size of the result set. Figure 5.1 shows a small example of evaluating such queries with binned bitmap indexes [Rotem et al., 2005b]. In this example we assume that an attribute A has values between 0 and 50. The values of the attribute A are given in the second leftmost column. The range of possible values of A is partitioned into five sub-ranges (bins), namely [0, 10], [11, 20] etc. with a bin allocated to each sub-range. The values of the sub-ranges are called bin boundaries. In this example, the width of each bin is of the same size. A "1-bit" indicates the attribute value falls into the range and "0-bit" otherwise. Assume that we want to evaluate the query "Count the number of rows where 8 < A < 37". The correct result should be 9. We know that all records that fall into internal bins (highlighted in light gray) are sure hits (qualifying records). These records are indicated by a “1-bit” and are calculated by performing a Boolean OR operation on all internal bins. On the other hand, records that fall into so-called edge bins (highlighted in dark gray) contain both qualifying and non-qualifying values. In order to prune the false positives, the original data values need to be checked against the query constraint. In particular, all records of the edge bins with a bit set to “1”, need to be checked. Such a check may involve additional accesses to disk pages depending on how the attribute values are stored. Given the query 8 < A < 37, let us look at the candidate check for the left edge bin. The candidate records are the records with RIDs 1, 4 and 6. The values of these records are 5, 9 and 5, respectively. The only qualifying record is record 4 that represents the value 9. The other two records do not fulfill the query constraint and do not qualify. This is evident from the above example that the cost of performing a candidate check on an edge bin is related to the number of “1-bits” in that bin. The larger the number of candidates that need to be checked, the higher the total query processing cost.

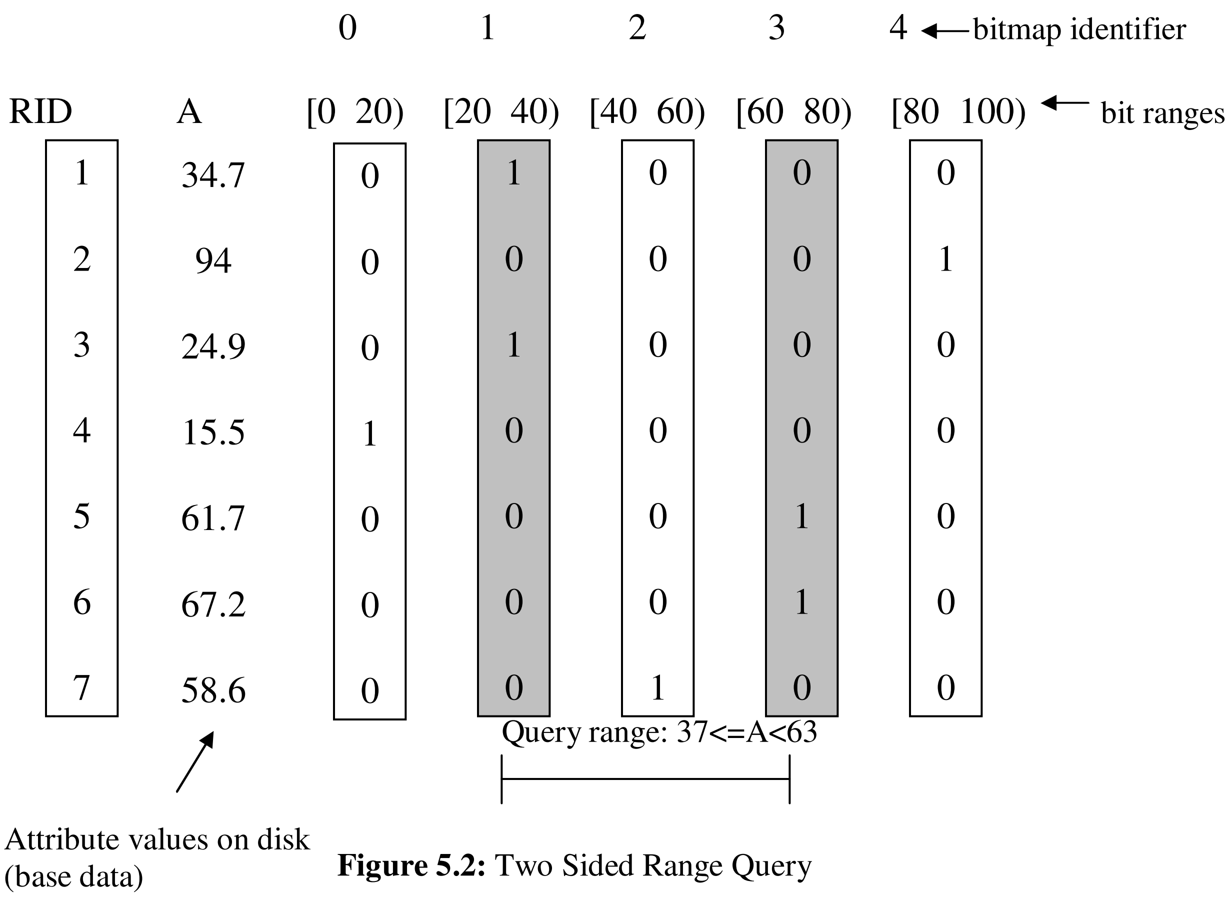


Figure 5.2: Two Sided Range Query

In this example we assume that an attribute *A* has values between 0 and 100. The values of the attribute *A* are given in the second leftmost column. The range of possible values of *A* is partitioned into five bins [0, 20), [20, 40).... A "1-bit" indicates that the attribute value *falls into a specific bin*. On the contrary, a "0-bit" indicates that the attribute value *does not* fall into the specific bin. Take the example of evaluating the query "Count the number of rows where 37 $\leq$ A < 63". The correct result should be 2 (rows 5 and 7). We see that the range in the query overlaps with bins 1, 2 and 3. We know for sure that all rows that fall into bin 2 *definitely qualify* (i.e., they are *hits*). On the other hand, rows that falls into bins 1 and 3 *possibly qualify* and need further verification. In this case, we call bins 1 and 3 *edge bins*. The rows (records) that fall into edge bins are *candidates* and need to be checked against the query constraint.

In the above example, there are four candidates, namely rows 1 and 3 from bin 1, and rows 5 and 6 from bin 3. The candidate check process needs to read these four rows from disk and examine their values to see whether or not they satisfy the user-specified conditions. On a large data set, a candidate check may need to read many pages and may dominate the overall query response time [Rotem et al., 2005b]. [Koudas, 2000] considered the problem of finding the optimal binning for a given set of equality queries. [Rotem et al. 2005a, 2005b] considered the problem of finding the optimal binning for range queries. Their approaches are based on dynamic programming. Since the time required by the dynamic programming grows quadratic with the problem size, these approaches are only efficient for attributes with relatively small attribute cardinalities [Koudas, 2000] or with relatively small sets of known queries [Stockinger et al. 2004]. They also considered the problem of optimizing the order of evaluating multi-dimensional range queries. The key idea is to use more operations on bitmaps to reduce the number of candidates checked. This approach usually reduces the total query response time. Further improvements to this approach are to consider the attribute distribution and other factors that influence the actual time required for the candidate check.

To minimize number of disk page accesses during the candidate check, it is necessary to *cluste*r the attribute values [Rotem et al., 2006]. A commonly used clustering (*data layout*) technique is called *vertical partitioning*. In general, the vertical data layout is more efficient for searching, while the horizontal organization (commonly used in DBMS) is more efficient for updating.

The behavior of the bitmap index in the multi-dimensional space is depicted in Figure 5.4. Note that for each attribute the candidate check is done separately, e.g after “XOR"ing the “candhits" slice with the “previous" slice. However, for all remaining attributes, the bit slice which is yielded after “XOR"ing, is “AND"ed together with the “global" hit slice. 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. Throughout the thesis we will refer to this process of sieving out the hits from the candidates as the candidate check. The details of the approach described above, is given in [Stockinger et al. 2004].

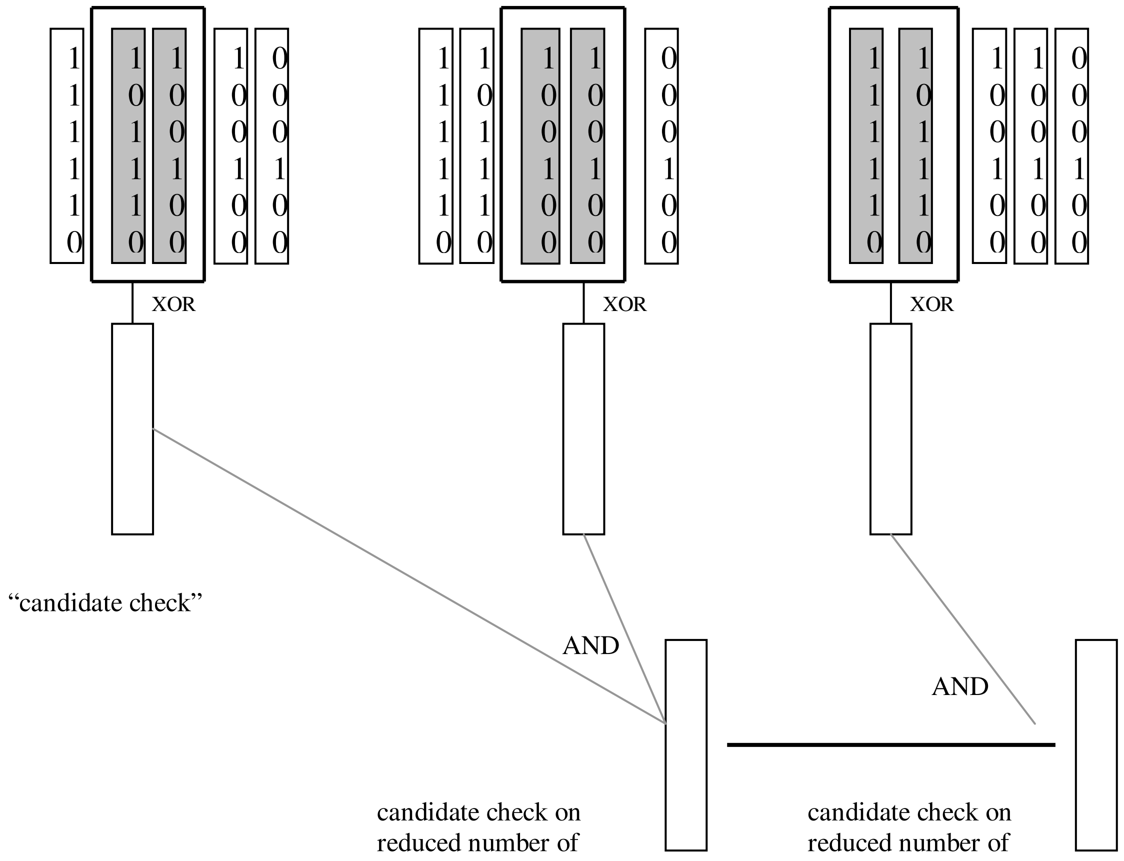

XOR
XOR
XOR
“candidate check”
AND
AND
candidate check on reduced number of
candidate check on reduced number of

Figure 5.3: Candidate check in multi-dimensional space.

5.3. Strategies for Efficient Binning

We extend the concept of equi-width bins, that is, each bin has the same width and thus the distribution of data values is reflected in the number of entries for each bin. Along with equi-width binning, there is further possibility of equi-depth binning which guarantees that each bin has the same number of entries. Both binning strategies have their own advantages and disadvantages. 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 choose 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. Binning could be made even more effective if the query access patterns can be incorporated in deciding the bin boundaries. 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.

Consider an attribute, A, with values ranging from 200 to 240, having *equi-width binning*: 200 - 210, 210 - 220, 220 - 230 and 230 - 240. To implement the equi-width binning, width of bin is taken as input. The query end points are searched in all the bin boundaries to locate which bins contains the lower and higher query end points. If the selected bins are b_1 and b_2. All the bins that fall between these bins so obtained are ORed, as they all contain only hits. The records corresponding to the result of the OR operation are directly retrieved. The bins b_1 and b_2 contain some records which are not hits and some of them are hits. Thus, for these two bins each record is taken and verified to check if it falls within the query boundaries and printed only if it is a hit.

Now, consider *range binning*: 200 - 210, 200 - 220, 200 - 230 and 200 - 240. The bins b_1 and b_2 are obtained in a similar manner to solve the query. We construct three intermediate equi-width bitmaps b_f, b_{f_1} and $b_f^{'}$. The pseudo-code for answering two sided range query using range binning is as follows:

```
If ( $b_1$ != 0)
{
$b_f = b_{f_1}$ XOR ($b_{f_1}$ - 1)    // $b_{f_1}$ - 1 indicates the bin before $b_1$
$b_f^{'}$ = NOT ($b_{f_1}$)
}
If($b_{f_1}$ == 0)
{
$b_f = b_{f_1}$
$b_f^{'} = 0xffffffff$
}
```

To search all records corresponding to "1" bit in b_f, and whenever a *hit* record is found, clear the bit corresponding to that record in b_f.

$$b_f = b_f \text{ XOR } b_f^{'}$$
$$b_{f_1} = b_2 \text{ XOR } (b_2 - 1)$$

To search all records corresponding to "1" bit in b_f, and whenever a *miss* record is found clear the bit corresponding to that record in b_f.

$$b_{f_1} = b_{f_1} \text{ OR } (b_2 - 1)$$
$$b_f = b_{f_1} \text{ AND } b_f$$

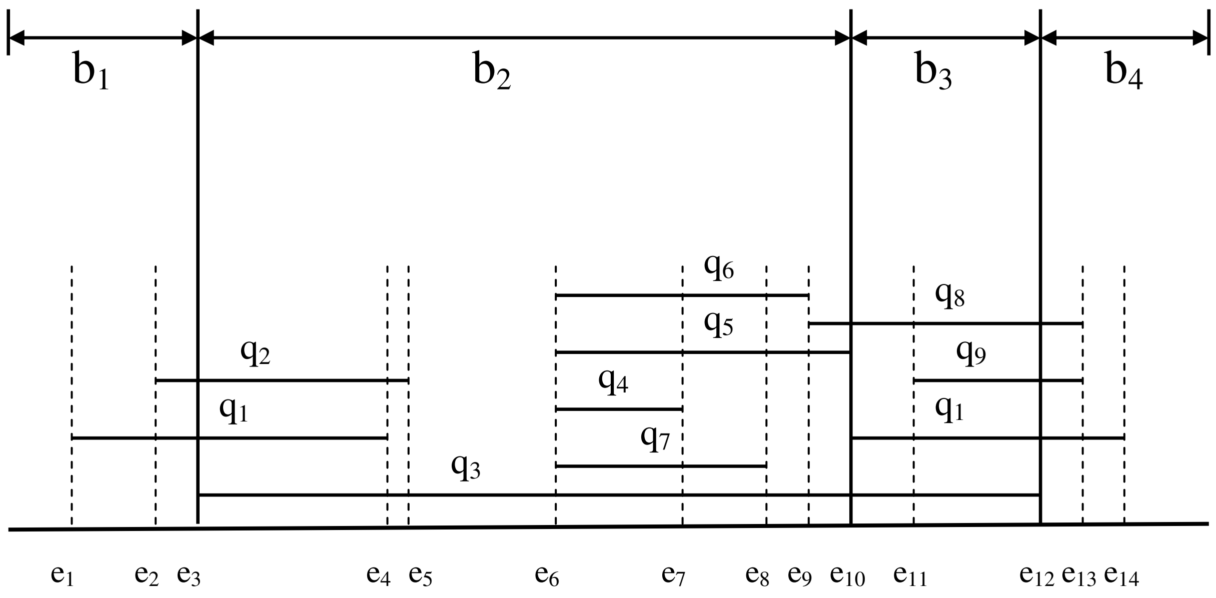
Finally, b_f contains only those records which lie within the query range and consequently printed without any record checking.

We introduce the concept of exact match binning, where we choose the bin boundaries exactly on the basis of the end points of most frequent queries on our system. It may happen that the entire range of a given attribute may not get covered by exact match bins. We create additional bins to cover the entire range. It may be noted that exact match binning produces non-uniform overlapping bins. When we pose any of our most frequent queries (satisfying a minimum threshold frequency) we don't need to perform the candidate check, we search the bin matching the query end points and get all hits from that bin. As all the high frequency queries match with the bin boundaries, this type of binning is the fastest possible binning. For answering low frequency queries, we may use the algorithms that were used with equi-width binning. This approach gives best results when we have more number of queries with frequencies greater than the threshold frequency. The concept of exact match binning is explained through an example.

Example: Suppose we consider two frequent queries $223 < A < 233$ and $228 < A < 235$. The bins are created as 200 – 223, 223 – 233, 228 – 235 and 235 – 240. When answering queries, for which exact bins are not there, and the query can be answered using multiple bins, we choose the bin which has less number of candidates so that the query response time may be reduced.

With exact match binning, the response time of queries is reduced considerably, at the expense of some additional space.

Figure 5.4 demonstrate a set of 10 range queries and a binning into 4 bins. The query q_3 has no edge bins since both of endpoints fall on bin boundaries. Each of the queries q_4, q_5, q_6, q_7, q_{10} has 1 edge bin and queries q_1, q_2, q_8, q_9 has 2 edge bins. Horizontal lines represent query ranges and dotted vertical lines mark query endpoints.



b_1
b_2
b_3
b_4
q_6
q_8
q_5
q_2
q_9
q_4
q_1
q_1
q_7
q_3
e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9 e_{10} e_{11} e_{12} e_{13} e_{14}

Figure 5.4: Query endpoints and bin boundaries.

Figure 5.5 shows same set of 10 range queries with overlapping bins. The query q_3 has no edge bins since both of endpoints fall on bin boundaries. Each of the queries q_4, q_5, q_6, q_7, q_{10} has 1 edge bin and queries q_1, q_2, q_8, q_9 has 2 edge bins.

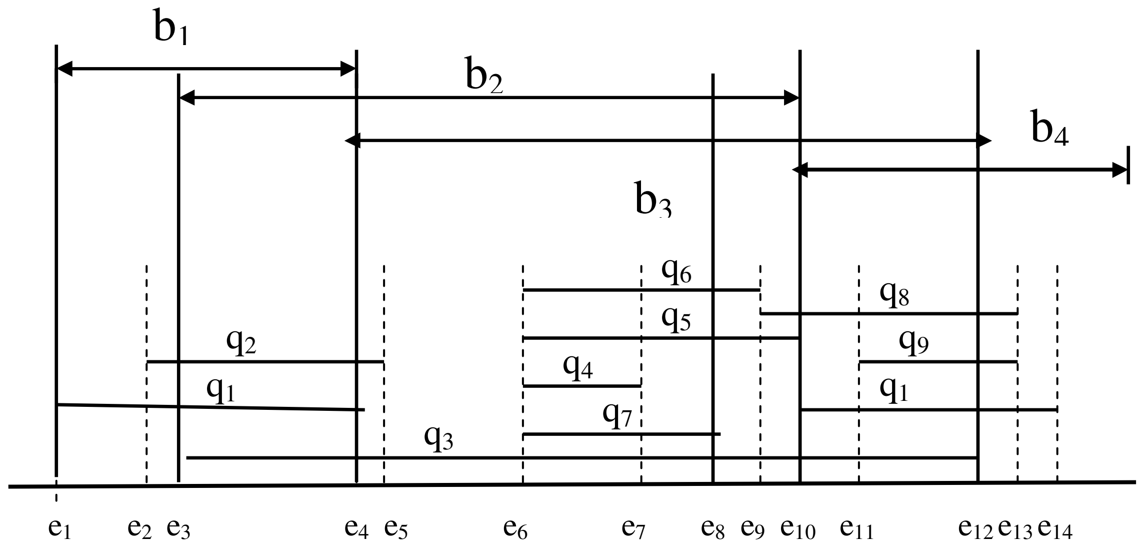


Figure 5.5: Exact match binning.

5.4. Algorithms for Query Processing

We now present algorithm for query processing for the new binning strategy described in the above section.

5.4.1. Algorithm for Equi-width Binning

Assuming attribute range 0 to 1000 and the number of bins is 10. Then equi-width binning is constructed as 0-100, 100-200, …... , 800-900, 900-1000.

POINT QUERY:

```
    Read $x$
    For (i = 0 to $N_b$)    //Find the bin that contains the entered $x$
    If ( $x_l$[i] ≤ $x$ < $x_u$[i] ) $b$ = i
    //scan through the entire bin and print records that match.
    // $b$ has only candidates and not hits. (check=1)
    Print record($b$,1)
```

LEFT SIDED QUERY:

```
    Read $x$
    For (i = 0 to $N_b$)      //Find the bin that contains the entered $x$
    If($x_l$[i] $\leq$ $x$ $<x_u$ [i]) $b$ = i
    If $b = 0$ GOTO LABEL
    //allocate memory equal to the size of one bin to some pointer named $b_f$
    For (i = 0 to $b$ - 1)
    //Now bitmap will have only hits as all bins to the left of $b$ have hits
    $b_f = b_f$ OR $b$[i]
    Print records ($b_f$, 0)
    LABEL :
    Print records ($b$, 1)
```

RIGHT SIDED QUERY:

```
    Read $x$
    For i=0 to $N_b$          //find the bin that contains the entered '$x$'
    If($x_l$[i]<= $x$ $<x_u$ [i]) $b$ = i
    If $b = N_b$ GOTO LABEL
    //allocate memory equal to the size of one bin to some pointer named '$b_f$'
    For(i=$b$+1 to $N_b$)//bitmap have only hits as all bins to the right of '$b$'have
    hits
    $b_f = b_f$ OR $b$[i]
    //print the records in bitmap without checking (check=0)
    Print records($b_f$,0)
    LABEL :
    //print the records in '$b$'with checking as it has only candidates (check=1)
    print_record($b$,1)
```

TWO SIDED QUERY:

```
    Read $l_q$ and $u_q$
    For i = 0 to $N_b$        //find the bin that contains the entered '$l_q$'
    If($x_l$[i]<= $l_q < x_u$ [i])
        $b_l$ = i
    For i=0 to $N_b$                    //find the bin that contains the entered '$u_q$'
        If($x_l$[i]<= $u_q$ $<x_u$ [i])
            $b_h$ = i
    //allocate memory equal to the size of one bin to some pointer named
'$b_f$'bitmap
    //if $b_l$ and $b_h$ are adjacent goto label
    If ($b_h$ - $b_l$) < 2
        GOTO LABEL
    For(i=$b_l$+1 to $b_h$-1)
        $b_f = b_f$ OR $b$ [i]

    //print the records in 'bitmap' without checking as it has only hits (check=0)
    Print records($b_f$,0)
    //print the records in $b_l$ and $b_h$ with checking as they have candidates
(check=1)
    LABEL :
    Print records($b_h$, $b_l$,1)
```

5.4.2. Algorithm for Exact match Binning

```
POINT QUERY:
    Read $x$
    Read $R_m$        //take a large number and call it '$R_m$'
    For(i=0 to $N_b$)
        If(($x_l$[i]<= $x < x_u$ [i]) && $R$[i]<$R_m$)
        {$b$ = i
        $R_m = R$[i]}
    //above step selects all bins containing the entered '$x$' and then
    //selects the bin with minimum number of candidates among them
    //print the records in '$b$' with checking as it has only candidates (check=1)
    Print records ($b$,1)

LEFT SIDED QUERY:
    Read $x$
    Read $R_m$        //take a large number and call it '$R_m$'
    For(i=0 to $N_b$)
        If(($x_l$[i]<= $x < x_u$ [i]) && $R$[i]<$R_m$)
        {$b$ = i
        $R_m = R$[i]}
    //above step selects all bins containing the entered id and then
    //selects the bin with minimum number of candidates among them
```

```
        //allocate memory equal to the size of one bin to some pointer named '$b_f$'
        //allocate memory equal to the size of one bin to some pointer named '$b_{f1}$'
        For(i=0 to $N_b$)
        If($x_u$ ($b$ [i])> $x_l$ ($b$))
                {$b_{f1}$ = $b$ OR $b$ [i] break }
        If($x_u$ ($b$ [i])<= $x_l$ ($b$))
                $b_f$ = $b_f$ OR $b$ [i]
        print the records in '$b_{f1}$' with checking as it has only candidates (check=1)
        //print the records in '$b_f$' without checking as it has only hits (check=0)
        Print records($b_f$,0)

RIGHT SIDED QUERY:
        Read $x$
        Read $R_m$        //take a large number and call it '$R_m$'
        For(i=0 to $N_b$)
        $x_l$[i]<= $x < x_u$ [i]) && $R$[i]<$R_m$)
                {$b$ =i  $R_m = R$[i]}

        //above step selects all bins containing the entered id and then
        // selects the bin with minimum number of candidates among them
        //allocate memory equal to the size of one bin to some pointer named '$b_f$'
        //allocate memory equal to the size of one bin to some pointer named '$b_{f1}$'
        For(i=0 to $N_b$)
        If($x_l$ ($b$ [i])< $x_u$ ($b$))
                {$b_{f1}$ = $b$ OR $b$ [i] break }
        If($x_l$ ($b$ [i])>= $x_u$ ($b$))
                $b_f$ = $b_f$ OR $b$ [i]
        //print the records in '$b_{f1}$' with checking as it has only candidates (check=1)
        Print records($b_{f1}$,1)
        //print the records in '$b_f$' without checking as it has only hits (check=0)
        Print records($b_f$,0)

TWO SIDED QUERY:
        Read $l_q$ and $u_q$
        If($f$>=5)
        For i=0 to $N_b$          //find the bin that exactly matches the query
                If( $x_l$[i]= $l_q$  AND $x_u$ [i]= $u_q$)
        .               $b$ =i
        //As this is a bin created specially for this particular query, it has all hits
        //print the records in '$b$' without checking as it has only hits (check=0)
        Print records(b,0)
        If( $f$ <5)
        For(i=0 to $N_b$)
                If(( $x_l$[i]<= $l_q < x_u$ [i]) && $R$[i]< $R_m$)
                { $b_l$ =i
                $R_m = R$[i]}
        For(i=0 to $N_b$)
                If( $x_l$[i]<= $u_q$ $< x_u$ [i]&& $R$[i]< $R_m$)
                { $b_h$ =i
                $R_m = R$[i]
                }
        //allocate memory equal to the size of one bin to some pointer '$b_f$'
        //allocate memory equal to the size of one bin to some pointer '$b_{f1}$'
        //allocate memory equal to the size of one bin to some pointer '$b_{f2}$'
        If( $b_h$ = $b_l$)
        //print the records in $b_l$' with checking as it has only candidates (check=1)
        Print records( $b_l$,1)
        EXIT
        For(i= $b_l$+1 to $b_h$-1)
        If( $x_u$ ($b$ [i])> $x_l$ [$b_h$])
                { $b_{f1}$ = $b_h$ OR $b$ [i] break }
        If( $x_l$ ($b$ [i])>= $x_u$ [$b_l$]&& $x_u$ [i]<= $x_l$ [$b_h$])
                $b_f$ = $b_f$ OR $b$ [i]
        For(i= $b_h$ to $b_l$+1)
        If( $x_l$ ($b$ [i])< $x_u$ ($b_l$))
                { $b_{f2}$ = $b_l$OR $b$ [i] break }
        //print the records in '$b_f$'without checking as it has only hits (check=0)
        Print records( $b_f$,0)
        //print the records in '$b_{f1}$'with checking as it has only candidates (check=1)
        Print records( $b_{f1}$,1)
        //print the records in '$b_{f2}$'with checking as it has only candidates (check=1)
```

```
    Print records($b_{f2}$,1)
```

5.4.3. Algorithm for Range Binning

POINT QUERY:

```
    Read $x$
    For(i=0 to $N_b$)        //find the bin that contains the entered 'id'
    If($x_l$[i]<= $x$ <$x_u$ [i]) $b$ = i
    If($b$=0) { $b_f$=$b$  GOTO LABEL }
    //allocate memory equal to the size of one bin to some pointer named '$b_f$'
    $b_f$ = $b$-1 AND $b$   //now bitmap will have less number of candidates, as
    we //eliminated many records not in the required range and which fall in
    previous bin
    //print the records in '$b_f$' with checking as it has only candidates (check=1)
    Print records($b_f$,1)
```

LEFT SIDED:

```
    Read $x$
    For i=0 to $N_b$          //find the bin that contains the entered '$x$'
    If($x_l$[i]<= $x$ <$x_u$ [i]) $b$ = i
    If($b$=0){
    Print records($b_l$,0)
    EXIT
     }
    //allocate memory equal to the size of one bin to some pointer named '$b_f$'
    $b_f$=$b$-1 AND$b$
    //print the records in '$b_f$' with checking as it has only candidates (check=1)
    Print records($b_f$,1)
    //print the records in '$b$-1' without checking as it has only hits (check=0)
    Print records($b$-1,0)
```

RIGHT SIDED:

```
    Read $x$
    For i=0 to $N_b$          //find the bin that contains the entered '$x$'
    If($x_l$[i]<= $x$ <$x_u$ [i]) $b$ = i
    //allocate memory equal to the size of one bin to some pointer named '$b_f$'
    //allocate memory equal to the size of one bin to some pointer named '$b_f'$'
    If($b$=0)
```

```
{ $b_f^{'} = \sim b$
$b_f = b$ }
If($b$ !=0)
{ $b_f = (b$ -1) AND $b$
$b_f^{'} = \sim(b$ -1)
$b_f^{'} = b_f^{'}$ AND $b_f$ }
//print the records in '$b_f$' with checking as it has only candidates (check=1)
Print records($b_f$,1)
//print the records in '$b_f^{'}$'without checking as it has only hits (check=0)
Print records($b_f^{'}$,0)
```

TWO SIDED :

```
Read $l_q$ and $u_q$
For i=0 to $N_b$          //find the bin that contains the entered '$l_q$'
If( $x_l$[i]<= $l_q < x_u$ [i])
        $b_l$ = i
For i=0 to $N_b$                    //find the bin that contains the entered '$u_q$'
If( $x_l$[i]<= $u_q < x_u$ [i])
        $b_h$ = i
//allocate memory equal to the size of one bin to some pointer named ' $b_f$'
//allocate memory equal to the size of one bin to some pointer named '$b_{f1}$'

//allocate memory equal to the size of one bin to some pointer named '$b_f^{'}$'
part1:
If($b_l$!=0)
$b_f$ <- ($b_l$ -1) AND ($b_l$)
$b_f^{'} = \sim(b_l)$
If($b_l$=0)
$b_f = b_l$
$b_f^{'}$ =0xffffffff
/*
candidate check on bitmap: miss -> do not alter the bit          //for this
purpose hitORmiss() function is used  hit -> clear the bit
*/
$b_f = b_f$ AND $b_f^{'}$
```

```
//after this step bitmap has only records greater than the lower limit
part2:
$b_{f1}$= $b_h$-1  AND $b_h$
/*
candidate check on $b_{f1}$: miss -> set the bit to zero        //for this purpose
hitORmiss() function is used hit -> do not alter the bit
*/
$b_{f1}$ = $b_{f1}$ OR $b_h$-1
//after this step bitmap has only records lesser than the lower limit
$b_f$ = $b_f$ & $b_{f1}$//right and left sided query results are combined
//Now bitmap is combination of Left and Right queries and contains exactly
the //results that are needed.
//print the records in '$b_f$'without checking as it has only hits (check=0)
Print records($b_f$,0)
```

5.5. Experimental work and Results

To compare the proposed binning strategy with the existing binning approaches, we have created a sample database and generated a set of 20 two-sided range queries with varying frequencies. Total number of queries in the set is 92. The sample queries are given in Table 5.1. The number of queries for which exact bins were created at different threshold frequencies is given in Table 5.2. The binning strategies considered are equi-width binning, range binning, and the proposed exact binning. We have developed a new query processing algorithm for exact binning which selects the best bin in case there is more than one bin from which a query can be answered. Testing of proposed algorithm for query processing was done for increasing number of records in the database starting from 1000 and going up to 50000. The CPU clock time for different query processing algorithms for different binning strategies is presented in Tables 5.3, 5.4, 5.5 and 5.6. The average time taken for answering the queries for frequency ν=5 and for increasing number of records is shown in Figure 5.6. The affect of increasing threshold frequency on the number of candidate checks is depicted in Figure 5.7. The improvement factor (%) over equi-width binning is given in Figure 5.8. The affect of increasing threshold frequency on average time needed to answer queries against 50000 records is presented in Figure 5.9. The variation in space required for exact binning with increasing threshold frequency is shown in Figure 5.10. The analysis of all the results presented is given below.

Following sample query set is considered:

Query No.	Query	Frequency
q1	100 ≤ IDNO ≤ 203	6
q2	053 ≤ IDNO ≤ 800	2
q3	417 ≤ IDNO ≤ 501	2
q4	121 ≤ IDNO ≤ 225	3
q5	600 ≤ IDNO ≤ 700	9
q6	817 ≤ IDNO ≤ 842	9
q7	052 ≤ IDNO ≤ 207	4
q8	333 ≤ IDNO ≤ 409	9
q9	701 ≤ IDNO ≤ 779	7
q10	321 ≤ IDNO ≤ 407	3
q11	505 ≤ IDNO ≤ 612	8
q12	170 ≤ IDNO ≤ 225	2
q13	213 ≤ IDNO ≤ 219	2
q14	714 ≤ IDNO ≤ 805	4
q15	117 ≤ IDNO ≤ 162	5
q16	070 ≤ IDNO ≤ 099	5
q17	400 ≤ IDNO ≤ 427	1
q18	221 ≤ IDNO ≤ 233	3
q19	072 ≤ IDNO ≤ 144	6
q20	513 ≤ IDNO ≤ 517	2

Table 5.1: Sample Queries

The number of exact and non exact queries at different frequency for the sample query set is as follows:

Frequency ≥	Number of Exact Queries	Non Exact
3	81	11
4	72	20
5	64	28
6	54	38
7	42	50

Table 5.2: Frequency of Exact and Non Exact Queries

From the tables 5.3 – 5.6 it is clear that the average time taken to answer the queries is minimum for exact binning and for the proposed query processing algorithm. It is true for all values of table cardinality considered. The summary of these results is presented in Figure 5.6. It can be seen from the figure that exact binning strategy performs far better than equi-width and range binning. The improvement achieved is more pronounced for higher number of records.

In Figure 5.7, the effect of increasing threshold frequency (ν) on the number of candidate checks is depicted. For equi-width binning, the number of candidate checks needed is independent of ν. For exact binning, the number of candidate checks increases with increasing ν. The proposed algorithm for exact binning reduces the number of candidate checks which in turn leads to faster response times for queries. In Figure 5.10, the space requirement for exact binning is plotted against threshold frequency. It takes more space than equi-width binning, as expected, but is found to decrease with frequency. From Figures 5.7- 5.10, it is evident that space-time performance optimality can be achieved by suitably choosing ν.

Query No.	**Equi Width**	**Range**	**Exact (Equi)**	**Exact(Overlap)**
q1	42760000	63490000	21520000	21560000
q2	190530000	195790000	168840000	209560000
q3	41990000	58260000	39000000	27100000
q4	42780000	64290000	48560000	81130000
q5	42880000	63170000	21080000	21180000
q6	41070000	45290000	5010000	5020000
q7	191300000	73570000	188420000	79510000
q8	42220000	57260000	15680000	15710000
q9	42530000	57800000	16160000	16460000
q10	42320000	59300000	42710000	78500000
q11	43030000	64250000	22150000	22200000
q12	42740000	53880000	48560000	79530000
q13	42550000	43170000	54130000	20780000
q14	41710000	59560000	23730000	77660000
q15	43030000	51610000	9440000	10320000
q16	41960000	46990000	5800000	5810000
q17	41640000	46420000	35150000	26660000
q18	42520000	44340000	54120000	20780000
q19	42450000	56550000	14810000	14850000
q20	42250000	42410000	44310000	21170000
	Average			
	57213000	62370000	43959000	42774500

Table 5.3: Query Processing Time for number of records = 50000

Query No.	Equi Width	Range	Exact (Equi)	Exact(Overlap)
q1	1720000	2620000	880000	960000
q2	7830000	8300000	7240000	7150000
q3	1750000	2540000	1730000	830000
q4	1700000	2650000	2030000	1010000
q5	1800000	2760000	940000	1000000
q6	1710000	1930000	230000	230000
q7	7820000	3070000	8050000	1460000
q8	1780000	2460000	670000	710000
q9	1750000	2470000	690000	740000
q10	1770000	2540000	1820000	3310000
q11	1770000	2730000	940000	1010000
q12	1710000	2250000	2030000	530000
q13	1720000	1810000	2320000	50000
q14	1730000	2540000	1000000	2770000
q15	1700000	2140000	400000	420000
q16	1690000	1980000	250000	270000
q17	1810000	2060000	1530000	950000
q18	1710000	1840000	2310000	930000
q19	1700000	2360000	630000	680000
q20	1700000	1750000	1880000	930000
	Average			
	2343500	2640000	1878500	1297000

Table 5.4: Query Processing Time for number of records = 10000

Query No.	Equi Width	Range	Exact (Equi)	Exact(Overlap)
q1	450000	700000	240000	240000
q2	2090000	2200000	1890000	1750000
q3	480000	700000	470000	220000
q4	440000	700000	540000	250000
q5	460000	690000	230000	240000
q6	470000	540000	70000	60000
q7	2060000	810000	2110000	370000
q8	500000	720000	210000	200000
q9	450000	630000	170000	170000
q10	510000	730000	490000	890000
q11	450000	700000	240000	240000
q12	460000	610000	540000	140000
q13	440000	460000	590000	10000
q14	470000	670000	250000	690000
q15	460000	560000	90000	90000
q16	430000	500000	60000	70000
q17	510000	580000	430000	250000
q18	450000	480000	590000	220000
q19	440000	610000	160000	140000
q20	440000	460000	480000	260000
	Average			
	623000	702500	492500	325000

Table 5.5: Query Processing Time for number of records = 5000

Query No.	Equi Width	Range	Exact (Equi)	Exact(Overlap)
q1	30000	40000	10000	10000
q2	130000	150000	120000	110000
q3	30000	40000	30000	10000
q4	30000	40000	30000	10000
q5	20000	40000	20000	20000
q6	20000	30000	10000	10000
q7	130000	50000	110000	50000
q8	30000	40000	10000	10000
q9	40000	40000	20000	10000
q10	20000	50000	20000	60000
q11	20000	40000	30000	10000
q12	40000	30000	20000	10000
q13	20000	30000	40000	10000
q14	20000	40000	20000	50000
q15	40000	40000	10000	10000
q16	30000	30000	10000	10000
q17	30000	40000	30000	10000
q18	20000	30000	30000	20000
q19	30000	30000	20000	10000
q20	30000	40000	30000	20000
	Average			
	38000	43500	31000	20000

Table 5.6: Query Processing Time for number of records = 1000

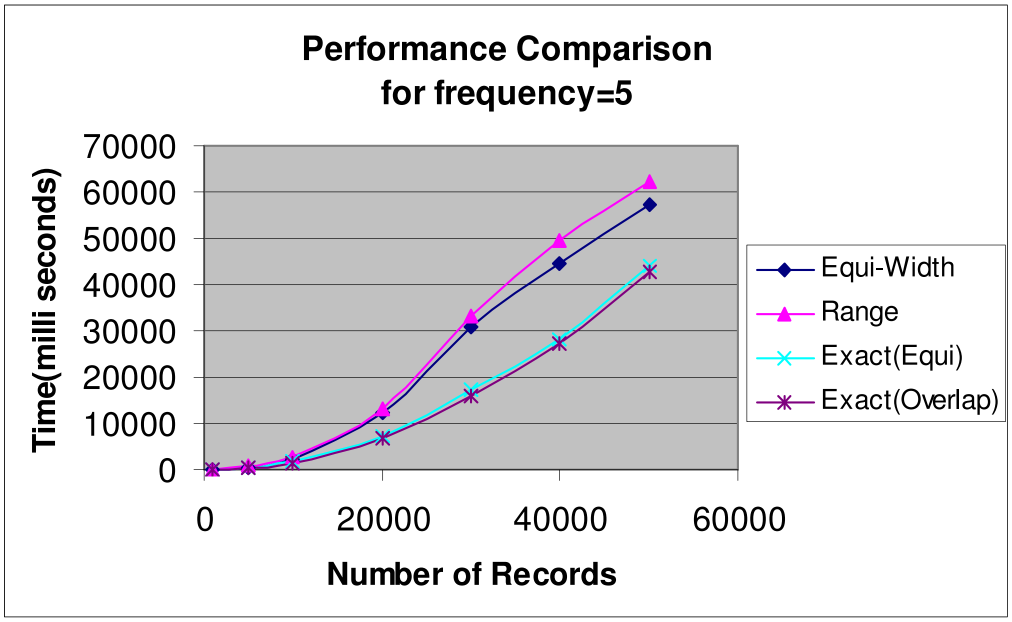


Figure 5.6: Average Time for Different Binning Algorithms

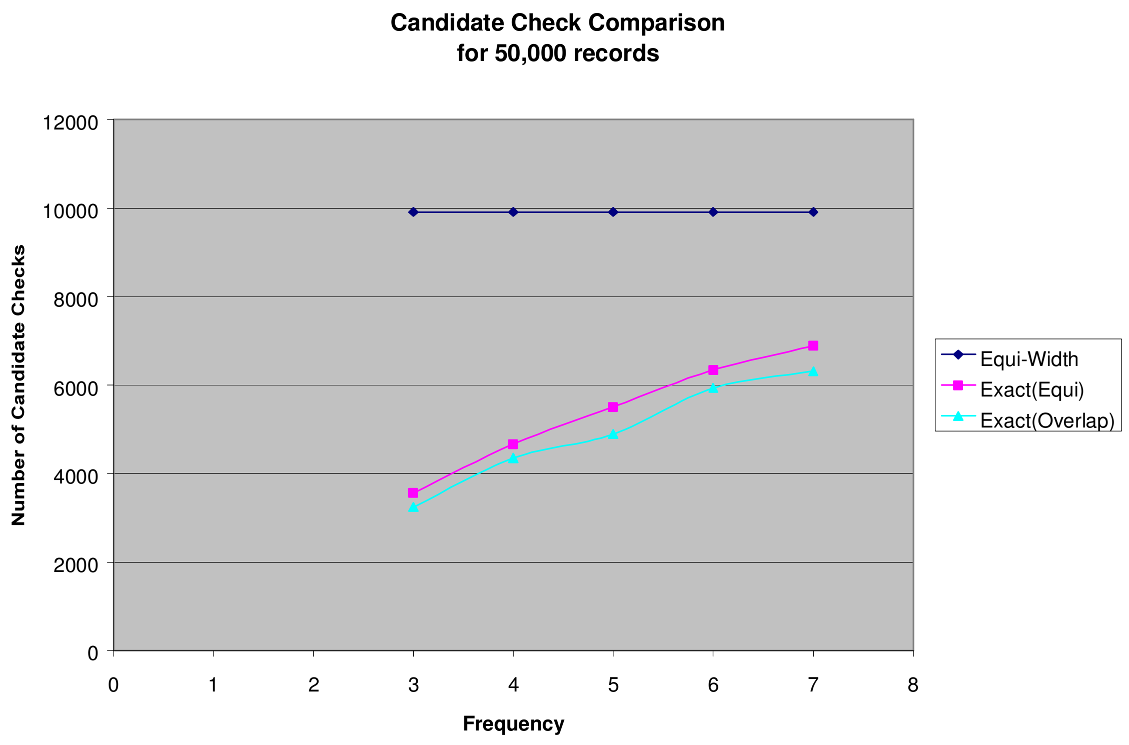


Figure 5.7: Number of candidates at different frequencies for binning algorithms

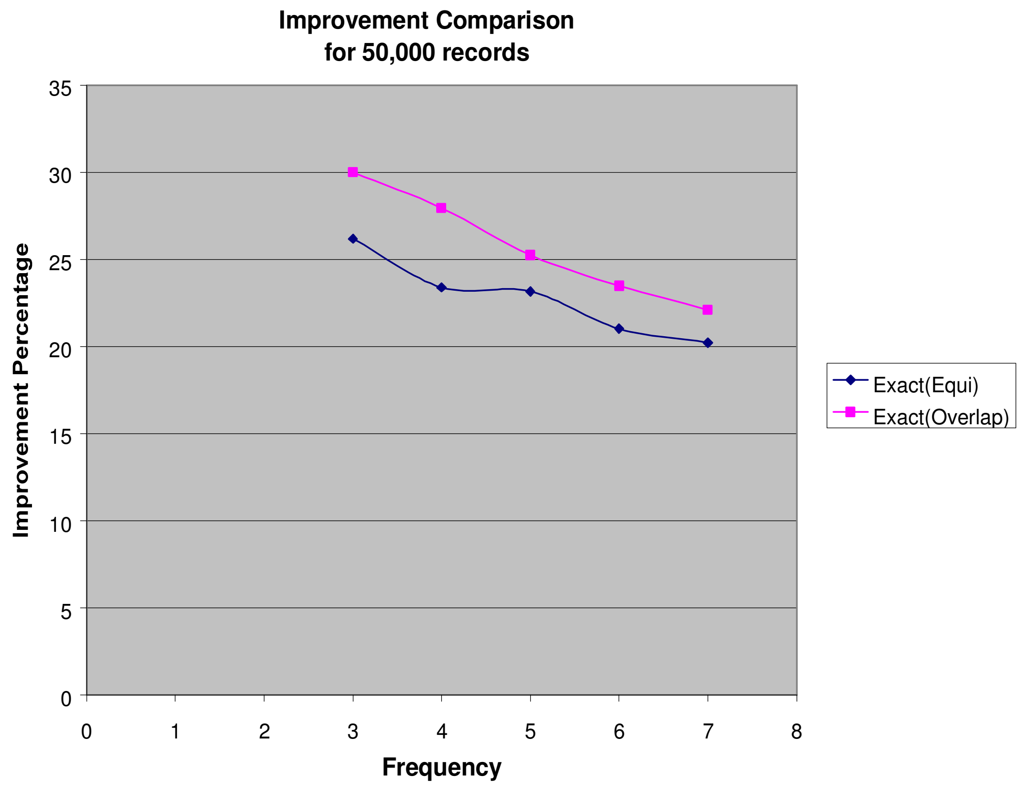


Figure 5.8: Improvement percentage at different frequencies for binning algorithms

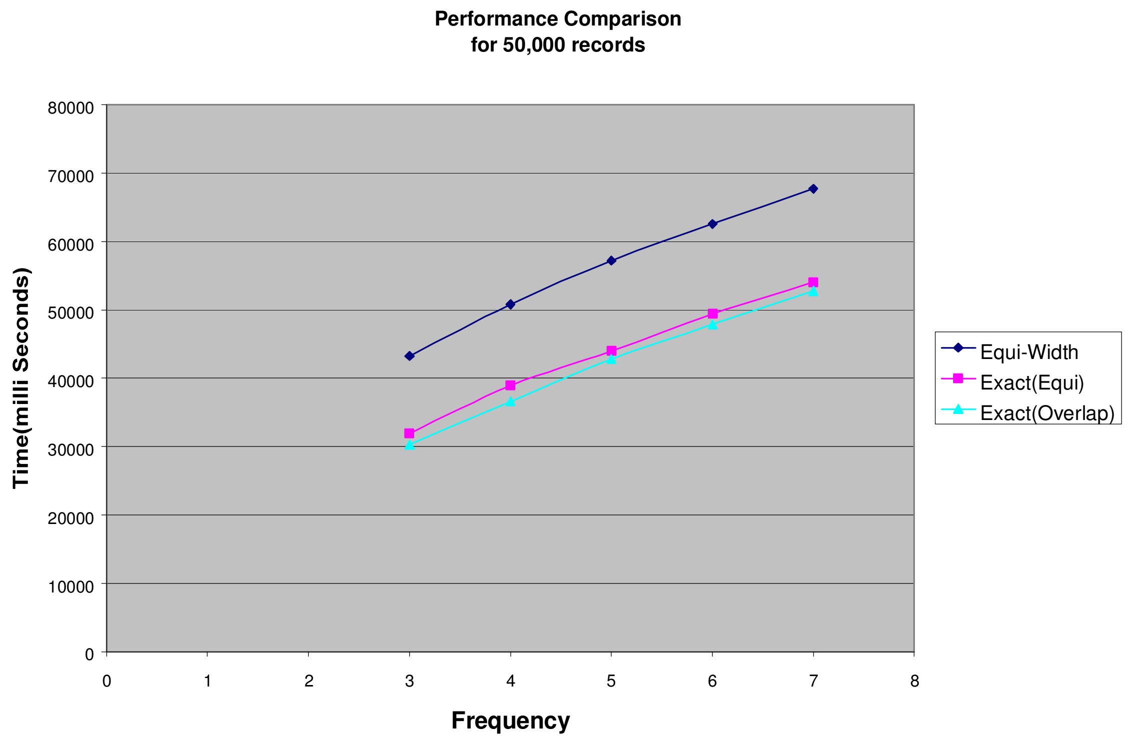


Figure 5.9: Query processing time at different frequencies for binning algorithms

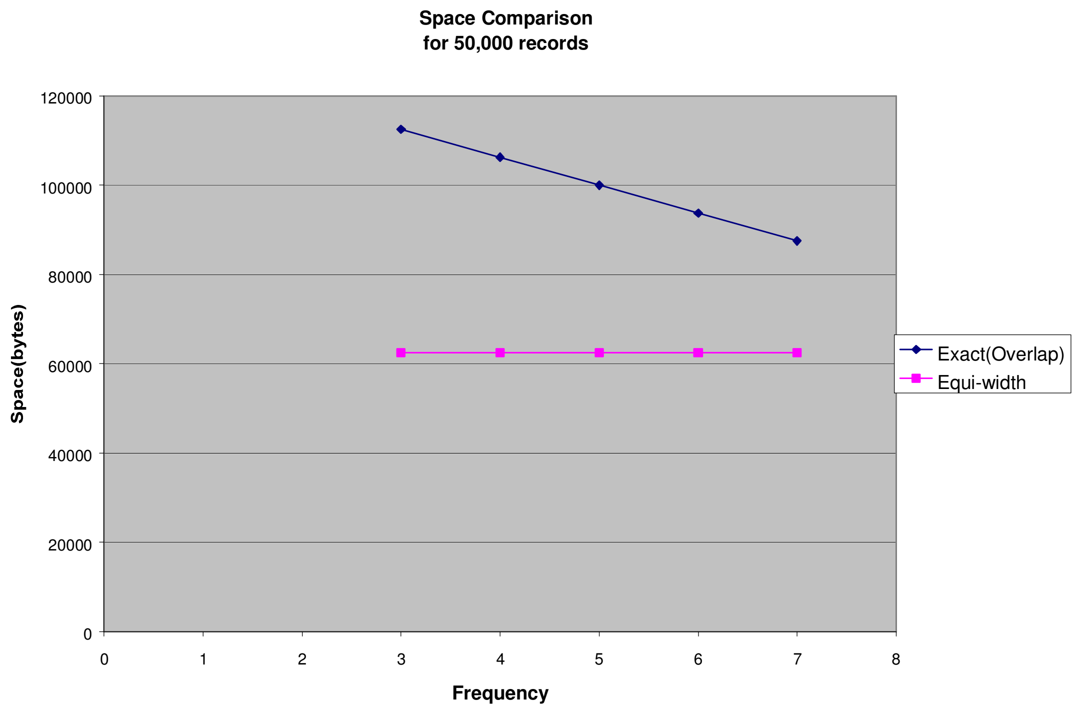


Figure 5.10: Space Comparison of exact binning with equi-width binning

5.6. Contributions and Summary

In this chapter, we proposed a new binning strategy called exact binning which takes query distribution into account. The bins are allowed to overlap and a given query could be answered from more than one bin. This necessitated the need for new query processing algorithms to be developed in order to minimize the number of candidate checks. We developed an algorithm for the same and compared its performance with the existing algorithms given for equi-width binning. It was found that the new binning strategy performs much better than any of the existing binning techniques at the expense of space. The threshold frequency ν can be optimally chosen to get maximum performance benefits with minimal space overhead.

Chapter 6: Conclusions and Recommendations

6.1. Conclusions

The ability to extract data to answer complex, iterative, and ad hoc queries quickly is a critical issue for data warehouse applications and scientific databases. A proper indexing technique is crucial to avoid I/O intensive table scans against large tables. The challenge is to find an appropriate index type that would improve the queries performance. Bitmap Indexes play a key role in answering data warehouse's queries because they have an ability to perform operations on index level before retrieving base data. This speeds up query processing tremendously. Bitmap indexes are the preferred multi-dimensional indexing structures especially suited for data warehouses and scientific databases which contain huge volumes of multidimensional data. In the thesis, an attempt has been made to improve the performance of bitmap indexes through better encoding, compression, and binning techniques.

One of the major issues with bitmap indexes is the space requirement. BBC and WAH are two very effective and popular compression schemes for bitmap indexes. Sorting the column on which a bitmap index has been created can drastically improve the compression achieved by BBC and WAH as both these compression schemes are variants of the run-length encoding (RLE) scheme. Using this simple technique, we observed that the space requirement increases only linearly with the increase in the number of tuples in the relation. It is very effective in read-mostly data warehouses and can be done during the ETL phase. Also, the response time of both equality and range queries is found to decrease. It was found that there is up to 60% improvement in response time of queries. With such promising gains in terms of both space and performance, the proposed strategy offers a simple yet effective solution to query performance challenges in large datasets. The other columns in the relation can continue to have BBC or WAH compressed bitmap indexes.

Data reorganization, mainly tuple reordering, plays an important role in improving the compression ratios achieved by BBC and WAH. In read-mostly environments, data reorganization is found to be very effective strategy to achieve good compression ratios. Multi-component encoding has been used as a preprocessing technique to improve the compression ratio achieved by Gray code ordering algorithm used for tuple reordering. Spacing saving of 25% was achieved by applying multi-component encoding just once. Choosing the base for multi component indexing is critical and thus finding a good base that maximizes the performance of WAH will be another interesting research project.

Binning plays a very important role in reducing the size of bitmap indexes on high cardinality columns. We have introduced the concept of exact overlapping bins to minimize the number of candidate checks needed to answer a set of queries. The overlapping and exact bins are created based on the frequency of queries. New algorithms for performing candidate checks have also been developed. Results are presented for different combination of parameters like cardinality and query frequency. The proposed new binning strategy and the associated algorithms greatly improve the response time of range queries as compared to the conventional binning techniques.

6.2. Recommendations for the future work

The techniques proposed in the thesis can be applied to bitmapped join indexes to reduce the space requirements of such indexes. It would be interesting to see the affect of this on the join operation.

It was observed for binning techniques that the number of candidate checks to be performed for a set of queries Q, is sensitive to the choice of the qualifying frequency. How to find an optimal frequency so that the numbers of candidate checks required for answering queries in a given set Q are minimized? This could be an interesting research problem. Also, dynamic binning strategies could be explored. If the binning boundaries could be changed based on the changing query profiles, query response times could be reduced drastically.

We are continuing our work on bitmap indexes to be applied in data mining as group bitmap index and improving its performance for doing better analysis. Bitmap indexing can be extensively used in data mining algorithms to extract data in faster.

Developing new indexing strategies based on bitmap indexes is still an interesting research area.

Despite the success of bitmap indexes, there are a number of important issues that remain to be addressed.

How to automatically select the best combination of encoding, compression and binning techniques?

How to use bitmap indexes to answer more general join queries?

Research work on bitmap indexes so far has concentrated on answering queries efficiently in a read-mostly environment, but has often neglected the issue of *updating* the indexes. Clearly, there is a need to update the indexes efficiently as and when the data file changes. Efficient solutions to this issue could lead to a wider adaptation of bitmap indexes in commercial systems.

List of Publications

1. Navneet Goyal, Yashvardhan Sharma**,** Susheel Kumar Zaveri, '**Improved Bitmap Indexing Strategy for Data Warehouses**', Proceedings of 9th International Conference on Information Technology (ICIT), Bhubaneshwar, ***IEEE Computer Society Press***, 18th - 21st December, pp 213-216, 2006

2. Yashvardhan Sharma, Navneet Goyal, Som Ranjan Satpathy, '**New Approach to overcome the complexity issues raised by Simple Bitmap Indexing**', Proceedings of International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering (CIS^{2}E 06), 4th -14th December, Bridgeport, USA, ***Springer***, pp.501-503, 2006.

3. Yashvardhan Sharma, '**Effective Usage of Metadata in Data Warehousing**', Proceedings of 2nd National Conference on Information & Emerging Technologies(NCIET), Ropar, Punjab, 7th – 8th September, pp. 97-105, 2007.

4. Yashvardhan Sharma and Navneet Goyal, '**Data Mining Algorithms with Bitmap Indexes**', Proceedings of National Conference On Emerging Trends In Information Technology(NCETIT), Indore**,** 18th – 20th December , pp. 316-320, 2007.

5. Yashvardhan Sharma and Navneet Goyal, '**An Efficient Multi-Component Indexing Embedded Bitmap Compression for Data Reorganization**', Information Technology Journal, Asian Network for Scientific Information Publications, Vol 7, No. 1, ISSN 1812-5638, pp.160-164, 2008.

6. Navneet Goyal and Yashvardhan Sharma**, 'Bitmap based Binning Algorithms for Two Sided Range Queries'**, ACM SIGMOD (Communicated).

REFERENCES

Aho, A. V. and Ullman, J. D., ‘Optimal Partial-Match Retrieval When Fields Are Independently Specified’, ACM Transactions Database Systems, Vol. 4, No. 2, pp.168–179, 1979.

Albrecht, J., Gunzel, H.and Lehner, W., ‘An Architecture for Distributed OLAP’, In Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), Las Vegas, USA ,July 13-16, 1998.

Amer-Yahia, S. and Johnson, T., ‘Optimizing Queries on Compressed Bitmaps’, In Proceedings of VLDB 2000, Morgan Kaufmann, September 10-14, pp. 329-338, 2000.

Antoshenkov, G. and Ziauddin, M., ‘Query Processing and Optimization in ORACLE RDB’, VLDB Journal, Vol. 5, pp. 229-237, 1996.

Antoshenkov, G., ‘Byte-aligned Bitmap Compression’, Technical Report, Oracle Corporation, 1994. U.S. Patent number 5,363,098.

Bayer, R., ‘UB-trees and UB-cache – A new processing paradigm for database systems’, Technical Report TUM-I9722, TU Mnchen, 1997.

Bayer, R., ‘The Universal B-Tree for Multidimensional Indexing: general Concepts’, In Proceedings of the International Conference on Worldwide Computing and Its Applications, March 10-11, pp.198-209, 1997 .

Bentley, J. L. , ‘Multidimensional binary search trees used for associative searching’, Communications of the ACM, Vol. 18, No. 9, pp. 509-517, 1975.

Berchtold, S. , Bohm, C., Keim, D. and Kriegel, H., ‘A cost model for nearest neighbor search in high-dimensional data Space’, In Proceedings ACM Symposium on Principles of Database Systems, Tuscon, Arizona, pp. 78–86, June 1997.

Berchtold, S., Keim, D. and Kriegel, H., ‘The X-tree: An index structure for high-dimensional data’, In Proceedings of the International Conference on Very Large Data Bases, Bombay, India, pp. 28–39, 1996.

Berchtold, S., Boehm, C., and Kriegl, H.P., ‘The Pyramid-Technique: Towards Breaking the Curse of Dimensionality’ SIGMOD Record, Vol. 27, No. 2, pp. 142-153, 1998.

Bernardo, Luis M., Shoshani, A., Sim, A. and Nordberg, H., ‘Access coordination of tertiary storage for high energy physics applications’, In IEEE Symposium on Mass Storage Systems, pp. 105-118, 2000.

Bohm, C., Berchtold, S. and Keim, D. A., ‘Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases’, ACM Computing Surveys, Vol.33, No. 3, pp.322–373, 2001.

Bookstein, A. and Klein, S.T., ‘Compression of Correlated Bit-Vectors’, Information. Systems, Vol.16, No.4, pp. 387-400, 1991.

Canahuate, G., FerhatosmaNoglu, H. and Pinar, A., ‘Improving Bitmap Index Compression by Data Reorganization’, IEEE Transactions on Knowledge and Data Engineering (Accepted 2006).

Chakrabarti, K. and Mehrotra, S., ‘The hybrid tree: An index structure for high dimensional feature spaces’, In Proceedings of the International Conference Data Engineering, Sydney, Australia, pp. 440–447, 1999.

Chan, C.Y. and Ioannidis, Y.E. 'Bitmap Index Design and Evaluation', SIGMOD, USA, ACM Press, pp. 355-366, 1998.

Chan, C.Y. and Ioannidis, Y.E., 'An Efficient Bitmap Encoding Scheme for Selection Queries', SIGMOD Conference, USA, ACM Press, Vol. 28, No. 2, pp. 215-226, 1999.

Chaudhuri, S. Dayal, U. and Ganti, V., 'Database Technology for Decision Support Systems', Computer, Vol. 34, No. 12, IEEE Computer Society Press , pp. 48-55, 2001.

Chaudhuri, S., and Dayal, U., 'An Overview of Data Warehousing and OLAP Technology', ACM SIGMOD Record, Vol. 26, No.1, pp. 65-74, 1997.

Chu, J-H. and Knott, G., 'An Analysis of B-Trees and Their Variants', Information Systems, Vol. 14, No. 5, pp. 359-370, 1989.

Codd, E. D., ' A relational model for large shared data banks', Communications of the ACM, Vol 13, No. 6, pp. 377-387,1970.

Colliat, G., 'OLAP, Relational and Multidimensional Database System', SIGMOD Record, Vol. 25, No. 3, pp.64-69, 1996.

Comer, D., 'The ubiquitous B-Tree', Computing Surveys, Vol. 11, No. 2, pp. 121-137, 1979.

Datta, A. , Ramamritham, K. and Thomas, H., 'Curio: A Novel Solution for Efficient Storage and Indexing of Data', In Proceedings of the 25th VLDB Conference, pp.730-733, 1999.

Datta, A., Moon, B. and Thomas, H., 'A Case for Parallelism in Data Warehousing and OLAP', In Proceedings of the 9th International Conference on Database and Expert Systems Applications (DEXA'98), pp. 226-231, 1998.

DeWitt, D.J. and Gray, J., 'Parallel Database Systems: The future of high performance database systems', Communications of the ACM, Vol. 35, No. 6, pp.85-98 , 1992.

Edelstein, H., 'Faster Data Warehouse', Information Week, pp. 77-88, 1995.

Evangelidis, G., Lomet, D. and Salzberg, B., 'The hB^{II} -tree: A modified hB-tree supporting concurrency, recovery and node consolidation', In Proceedings of Very Large Data Bases, pp. 551-561, 1995.

Ezeife, C.I., 'A Uniform Approach For Selecting Views and Indexes in a Data Warehouse', In Proceedings of the 1997 International Database Engineering and Application Symposium, Canada, IEEE publication, pp. 151 – 160, 1997.

FastBit - An Efficient Compressed Bitmap Index Technology. http://sdm.lbl.gov/fastbit/.

FastBit (2005), Retrieved January 11, from http://sdm.lbl.gov/fastbit, 2006.

French. C. D., '"One Size Fits All" Database Architectures Do Not Work for DSS', In Proceedings of the 1995 ACM SIGMOD Conference, pp. 449-450,1995.

Furuse, K. , Asada, K. and Iizawa, A., 'Implementation and performance evaluation of compressed bit-sliced signature files', In Proceedings of the 6th International Conference, CISMOD'95, Bombay, India, Volume 1006 of Lecture Notes in Computer Science, Springer, pp.64-177, 1995.

Gaede, V. and Guenther, O., ‘Multidimensional Access Methods’, ACM Computing Surveys, Vol. 30, No. 2, pp. 170—231, 1998.

Gosink, L. J., Shalf, J., Stockinger, K., Wu, K. and Bethel, W., ‘HDF5-FastQuery: Accelerating Complex Queries on HDF Datasets using Fast Bitmap Indices’, In Proceedings of the 18th International Conference on Scientific and Statistical Database Management, SSDBM 2006, 3-5 July, Vienna, Austria, IEEE Computer Society , pp.149-158, 2006.

Goyal, K. B., Ramamritham, K., Datta, A. and Thomas, H. M., ‘Indexing and Compression in Data Warehouses’, In Proceedings of the Int'l. Workshop Design and Management of Data Warehouses, Heidelberg, Germany, 14-15 June, pp. 11-17, 1999.

Gray J., Bosworth A., Layman A., and Pirahesh H., ‘Data Cube: A Relational Operator Generalizing Group-By, Cross-Tab, and Sub-Totals’, In Proceedings of the 12th International Conference on Data Engineering, pp. 152-159, 1996.

Gray, J. , Liu, D. T. , Nieto-Santisteban, M., Szalay, A., DeWitt, D. and Heber, G., ‘Scientific Data Management in the Coming Decade’, In SIGMOD Record, Vol. 34, No. 4, December 2005.

Guha, S. , Koudas, N. and Srivastava, D., ‘Fast Algorithms For Hierarchical Range Histogram Construction’, In Proceedings of the ACM Symposium on Principles of Database Systems (PODS), Madison, Wisconsin, USA, ACM Press, pp. 180-187, 2002.

Gupta, A., Davis, K.C., and Jennifer, G-L., ‘Performance comparison of property map and bitmap indexing’, International Workshop on Data Warehousing and OLAP, pp. 65-71, 2002.

Guttman, A., ‘R-trees: A dynamic index structure for spatial searching’, In Proceedings of 1984 ACM SIGMOD, pp. 47-57, 1984.

Hallmark, G. , ‘The oracle warehouse’, In Proceedings of the Very Large Data Bases, pp. 707-709, 1995.

Harinarayan,V., Rajaraman, A. and Ullman, J. D., ‘Implementing Data Cubes Efficiently’, In Proceedings of the 1996 ACM SIGMOD Conference, pp. 205-216, 1996.

Hellerstein, J. M. , Koutsoupias, E. and Papadimitriou, C. H., ‘On the Analysis of Indexing Schemes’, In Proceedings of the Symposium on Principles of Database Systems (PODS), Tucson, Arizona, USA, May, pp. 249-256, 1997.

HP, ‘HP Intelligent Warehouse’, Hewlett Packard white paper, *http://www.hp.com*, 1997.

Informatica, ‘EnterpriseScalable Data Marts: A New Strategy for Building and Deploying Fast, Scalable Data Warehousing Systems’, Informatica white paper, http://www.informatica.com, 1997.

Ishikawa, Y., Kitagawa, H. and Ohbo, N., ‘Evalution of signature files as set access facilities in OODBs’, In Proceedings ACM SIGMOD International Conference on Managerment of Data, Washington, D.C., ACM Press pp. 247–256, 1993.

Jeong, J. and Nang, J., ‘An Efficient Bitmap Indexing Method for Similarity Search in High Dimensional Multimedia Databases’, In Proceedings of ICME 2004, pp. 815-818, 2004.

Johnson, D. S., Krishnan, S., Chhugani, J., Kumar, S. and Venkatasubramanian, S., ‘Compressing large boolean matrices using reordering techniques’, In Proceedings of the 30th International Conference on Very Large Data Bases, pp. 13–23, 2004.

Johnson, T. and Shasha, D., ‘Some Approaches to index design for cube forests’, Bulletin of the Technical Committee on Data Eng., Vol. 20, No. 1, pp. 27-35, 1997.

Johnson, T., ‘Performance measurements of compressed bitmap indices’, In Proceedings of the 25th International Conference on Very Large Data Bases, Morgan Kaufmann, pp. 278-289, 1999.

Jurgens, M. and Lenz, H.-J. ,’Tree based indexes vs. bitmap indexes - a performance study’, International Journal of Cooperative Information Systems, Vol. 10, No. 3, pp.355-376, 2001.

Keim, D. and Hinneburg, A., ‘Optimal Grid-Clustering: Towards Breaking the Curse of Dimensionality in High-Dimensional Clustering’, In Proceedings of the International Conference on Very Large Data Bases (VLDB), San Francisco. Morgan Kaufmann, pp. 506-517, 1999.

Kiyoki, Y., Tanaka, K., Aiso, H. and Kamibayashi, N., ‘Design and Evaluation of a Relational Data Base Machine Employing Advanced Data Structures and Algorithms’, Symposium on Computer Architecture, Los Alamitos, CA, USA., IEEE Computer Society Press, pp. 407-423, 1981.

Knuth, D. E., ‘The Art of Computer Programming’, Vol. 3, Addison Wesley, 1998.

Koudas, N. , ‘Space Efficient Bitmap Indexing’, In Proceedings of the Ninth International Conference on Information Knowledge Management CIKM, McLean, VA , pp. 194-201, 2000.

Koudas, N. , Muthukrishnan, S. and Srivastava, D., ‘Optimal Histograms for Hierarchical Range Queries’, In Proceedings of the ACM Symposium on Principles of Database Systems (PODS), Dallas, Texas, USA, ACM Press, pp. 196 -204, 2000.

Lee, D. L. , Kim, Y. M. and Patel. G., ‘Efficient signature file methods for text retrieval’, IEEE Transactions on Knowledge and Data Engineering, Vol. 7, No. 3, pp. 423-435, 1995.

Leslie, H., Jain, R., Birdshell, D., and Yagmai, H., ‘Efficient Search of Multidimensional B-Trees’ , In Proceedings of the International Conference on Very Large Data Bases (VLDB), Zurich, Switzerland, pp. 710-719, 1995.

Liao, S., Lopez, M. A and Leutenegger, S. T. , ‘High dimensional similarity search with space filling curves’, In Proceedings of the 17th International Conference on Data Engineering, IEEE Computer Society, pp. 615-622, 2001.

Lim, S.J. and. Ng, Y. K., ‘A Formal Approach for Horizontal Fragmentation in Distributed Deductive Database Design’, In Proceedings of the International Conference on Database and Expert Systems Applications (DEXA’96), Zurich, Switzerland, pp234-243, 1996.

Lin, K. , Jagadish, H. V. and Faloutsos, C., ‘The TV-tree: An index structure for high-dimensional data’, VLDB Journal,Vol.3,pp.517–542, 1995.

Lomet, D. B. and Salzberg, B., ‘The hb-tree: A multi-attribute indexing method with good guaranteed performance’, ACM Transactions on Database Systems, Vol. 15, No.4, pp.625–658, 1990.

Lu, H., Ooi, B. C. and Tan, K. L., ‘Query Processing in Parallel Relational Database Systems’, IEEE Computer Society, May 1994.

Markl, V., and Bayer, R., ‘Processing relational OLAP queries with UB-trees and multidimensional hierarchical clustering’, In Proceedings of the Second International

Workshop on Design and Management of Data Warehouses, DMDW 2000, Stockholm, Sweden, June 5-6, pp. 1-10, 2000.

Markl, V., Zirkel, M. and Bayer, R., 'Processing operations with restrictions in RDBMS without external sorting: The tetris algorithm', In Proceedings of the 15th International Conference on Data Engineering, March, Sydney, Austrialia, IEEE Computer Society, pp. 562-571, 1999.

Meredith, M.A, and Khader, A., 'Divide and Aggregate – Designing Large warehouses', Database Programming and Design, Vol. 9, No. 6, pp. 24-30, 1996.

Mofat, A. and Zobel, J., 'Parameterised compression for sparse bitmaps', In Proceedings ACM-SIGIR International Conference on Research and Development in Information Retrieval, Copenhagen, June 1992, ACM Press, pp. 274-285, 1992.

Mohr, A. E., 'Bit Allocation in Sub-linear Time and the Multiple-Choice Knapsack Problem', In Proceedings of the Data Compression Conference, Snao Bird, Utah, USA, IEEE Computer Society Press, pp. 352-361, 2002

Mokbel, M. F. and Aref, W. G. , 'Irregularity in multi-dimensional space-filling curves with applications in multimedia databases', In Proceedings of the tenth international conference on Information and knowledge management, ACM Press, pp. 512-519, 2001.

Nam, B. and Sussman, A., 'Improving Access to Multidimensional Self-describing Scientific Dataset', In International Symposium on Cluster Computing and the Grid (CCGrid), May 2003, Tokyo, Japan. IEEE Computer Society Press, pp. 172-179, 2003.

O'Neil, E. , O'Neil, P., and Wu, K., 'Bitmap Index Design Choices and Their Performance Implications', In Proceedings of the 11th International Database

Engineering and Applications Symposium (IDEAS 2007) , IEEE Computer Society , pp. 72-84, 2007.

O'Neil, P. and Graefe, G., 'Multi-Table Joins Through Bitmapped Join Indices', ACM SIGMOD Record, pp. 8-11, 1995.

O'Neil, P. and Quass D., 'Improved Query Performance with Variant Indexes', In Proceedings ACM SIGMOD International Conference on Management of Data, ACM Press Tucson, Arizona, pp. 38-49, 1997.

O'Neil, P., 'Model 204 Architecture and Performance', In Second International Workshop in High Performance Transaction Systems, Volume 359 of Springer-Verlag Lecture Notes in Computer Science, Asilmore, California, pp. 40-59, 1987

OLAP Council, 'APB-1 OLAP Benchmark Release II', November 1998. http://www.olapcouncil.org.

O'Neil, P. 'Informix and Indexing Support for Data Warehouses', Database Programming and Design, Vol. 10, No. 2, pp. 38-43, 1997.

Otoo, E. J. , Shoshani, A. and Hwang, S., 'Clustering high dimensional massive scientific dataset', In SSDBM, Fairfax, Virginia, pp. 147–157, 2001.

Ozbutun, C., 'Bitmap Indexes, Oracle 7.3 and 8.0.', ORACLE Technical Report, ACTA Journal, June 1997.

Park, J. and Nang, J., 'A hierarchical bitmap indexing method for content based multimedia retrieval', In Proceedings of the 24th IASTED international conference on Internet and multimedia systems and applications, Austria, ACTA Press, pp. 223-228, 2006.

Pinar, A. and Heath, M., ‘Improving performance of sparse matrix-vector multiplication’, In Proceedings of Supercomputing , 1999.

Pinar, A., Tao, T. and Ferhatosmanoglu, H., ‘Compressing bitmap indices by data reorganization’, In International Conference on Data Engineering, pp. 310–321, 2005.

Poess, M. and Floyd, C., ‘New TPC Benchmarks for Decision Support and Web Commerce’ ACM SIGMOD Record, Vol. 29, No. 4, pp. 64-71, 2000.

Ramakrishna, M.V., ‘In Indexing Goes a New Direction’, Vol. 2, pp. 70-77, 1999.

Richards, D., ‘Data compression and gray-code sorting’, Information Processing Letters, Vol. 22, No. 4, pp.201–205, 1986.

Rinfret, D., O'Neil, P. E. and O'Neil, E. J., ‘Bit-Sliced Index Arithmetic’, In Proceedings of the ACM Conference on Management of Data (SIGMOD), Santa Barbara, CA, USA, ACM Press, pp. 47–57, 2001.

Robinson, J., ‘The K-D-B-tree: A search structure for large multidimensional dynamic indexes’, In Proceedings of 1981 ACM SIGMOD, pp. 10-18, 1981.

Rotem, D. , Stockinger, K. and Wu, K., ‘Efficient Binning for Bitmap Indices on High-Cardinality Attributes’, Technical Report LBNL-56936, Berkeley Lab, Berkeley, California, USA, Nov. 2004.

Rotem, D., Stockinger, K. and Wu, K. ‘Optimizing I/O Costs of Multi-Dimensional Queries using Bitmap Indices’, In Proceedings of the International Conference on Database and Expert Systems Applications (DEXA), Copenhagen, Denmark, Springer Verlag, pp. 220-229, 2005a.

Rotem, D., Stockinger, K. and Wu, K., ‘Minimizing I/O Costs of Multi-Dimensional

Queries with Bitmap Indices', In Proceedings of the 18th International Conference on Scientific and Statistical Database Management, Vienna, Austria, IEEE Computer Society, pp. 33-44, 2006.

Rotem, D., Stockinger, K. and Wu, K., 'Optimizing Candidate Check Costs for Bitmap Indices', In Proceedings of the Conference on Information and Knowledge Management (CIKM), Bremen, Germany, November, ACM Press, pp. 648-655, 2005b.

Rotem, D., Stockinger, K. and Wu, K., 'Towards Optimal Multi-Dimensional Query Processing with Bitmap Indices', Technical Report LBNL- 58755, Berkeley Lab, Berkeley, California, USA, 2005.

Sarawagi, S. and Stonebraker, M., 'Efficient organization of Large Multidimensional Arrays', In Proceedings of the tenth international Conference on Data Engineering ICDE, Houston, pp. 328-336, 1994.

Sarawagi, S., 'Indexing OLAP Data', Bulletin of the Technical Committee on Data Engineering, IEEE, Vol. 20, No. 1, pp. 36-43, 1997

Savage, C. , 'A survey of combinatorial Gray codes', SIAM Review, Vol. 39, No. 4, pp. 605-629, 1997.

Schuegraf, E.J., 'Compression of Large Inverted Files with Hyperbolic Term Distribution', Information Processing and Management, pp. 377-384, 1976.

SciDAC. Scientific data management center. http://sdm.lbl.gov/sdmcenter/, 2002.

Sellis, T., Roussopoulos, N. and Faloutsos, C., 'The R^+-tree: A dynamic index for multi-dimensional objects', In Proceedings of 13th VLDB Conference, pp. 507-518, 1987.

Shoshani, A., “Statistical Databases : Characteristics, problems and some solutions’, In Proceedings of the 8th International Conference on Very Large Data Bases (VLDB), Mexico City, pp. 208-213, 1982.

Shoshani, A., Bernardo, L. M., Nordberg, H., Rotem, D., Sim, A., ‘Multidimensional indexing and query coordination for tertiary storage management’, In 11th International Conference on Scientijic and Statistical Database Management (SSDBM), IEEE Computer Society, pp. 214-225, 1999.

Sinha, R. R. and Winslett, M., ‘Multi-Resolution Bitmap Indexes for Scientific Data’, ACM Transactions on Database Systems, Vol. 32, No. 3, Article 16, August 2007.

Sinha, R. R., Mitra, S., and Winslett, M., ‘Bitmap indexes for large scientific data sets: A case study’, In Proceedings of the IEEE International Parallel & Distributed Proceessing Symposium, IEEE Computer Society Press, Los Alamitos, CA, 2006.

Sloan Digital Sky Survey. http://www.sdss.org/dr1/.

SNAP. SuperNova acceleration probe. http://snap.lbl.gov/, 2004.

Staman, J.P., ‘Structuring databases for analyses’, IEEE Spectrum, pp. 55-58, 1993.

Stockinger, K. , Duellmann, D., Hoschek, W. and Schikuta, E., ‘Improving the performance of high-energy physics analysis through bitmap indices’, In 11th International Conference on Database and Expert Systems Applications DEXA 2000, London, Greenwich, UK, pp. 835-845, September 2000.

Stockinger, K. , Shalf, J., Bethel, W. , and Wu, K. ‘Query-Driven Visualization of Large Data Sets’, In IEEE Visualization, Minneapolis, MN, October 23-25, IEEE Computer Society Press, 2005.

Stockinger, K. , Wu, K., Brun, R. and Canal, P., 'Bitmap Indices for Fast End-User Physics Analysis in ROOT', Nuclear Instruments and Methods in Physics Research Section A, Vol. 559, No. 1, pp. 99-102, 2006.

Stockinger, K., 'Design and Implementation of Bitmap Indices for Scientific Data', In International Database Engineering & Applications Symposium, Grenoble, France, July 2001, IEEE Computer Society Press, pp. 47-57, 2001.

Stockinger, K., Rotem, D., Shoshani, A. and Wu, K., 'Analyzing Enron Data: Bitmap Indexing Outperforms MySQL Queries by Several Orders of Magnitude'**,** Technical Report LBNL- 61768, Berkeley Lab, Berkeley, California, USA, 2006.

Stockinger, K., Shalf, J., Bethel, W. and Wu, K., 'DEX: Increasing the Capability of Scientific Data Analysis Pipelines by Using Efficient Bitmap Indices to Accelerate Scientific Visualization', In Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM), Santa Barbara, California, USA, IEEE Computer Society Press, 2005.

Stockinger, K., Wu, K. , Campbell, S., Lau, S. , Fisk, M. , Gavrilov, E., Kent, A., Davis, C.E. , Olinger, R., Young, R. , Prewett, J.E. , Weber, P. , Caudell, T.P. , Bethel, E.W. and Smith, S. 'Network Traffic Analysis With Query Driven Visualization SC 2005 HPC Analytics Results' In Supercomputing 2005, HPC Analytics Challenge, November 2005

Stockinger, K., Wu, K. and Shoshani, A., 'Evaluation Strategies for Bitmap Indices with Binning', In Proceedings of the International Conference on Database and Expert Systems Applications (DEXA), Zaragoza, Spain, Springer-Verlag, pp. 120-129, 2004.

Stockinger, K., Wu, K. and Shoshani, A., 'Strategies for Processing ad hoc Queries on Large Data Warehouses', In Proceedings of the fifth ACM international workshop on Data Warehousing and OLAP, ACM Press, McLean, Virginia, USA, pp. 72-79, 2002.

Stockinger,K., 'Bitmap Indices for Speeding Up High-Dimensional Data Analysis', In Proceedings of the 13th International Conference on Database and Expert Systems Applications, pp.881-890, 2002

Szalay, A. , Kunszt, P., Thakar, A., Gray, J. and Slutz, D., 'Designing and Mining Multi-Terabyte Astronomy Archives: The Sloan Digital Sky Survey', In SIGMOD, Dallas, Texas, USA, ACM Press, 2000.

TeraScaleCombustion, 'TeraScale High-Fidelity Simulation of Turbulent Combustion with Detailed Chemistry', Retrieved January 11, 2006 from http://scidac.psc.edu.

Transaction Processing Performance Council (TPC), "TPC Benchmark D, Decision Support", Standard Specification Revision 2.0.1, December 5, 1998, http://www.tpc.org.

Weber, R. , Schek, H.J. and Blott, S., 'A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces', In Proceedings of the 24th International Conference on Very Large Data Bases, August 1998, New York City, New York, USA, pp. 194-205,1998.

Winter, R., 'Indexing Goes a New Direction', Intelligent Enterprise, 1999, Vol.2, No.2, pp. 70-73, 1999.

Wong, H. K. T., Liu H. F., Olken F., Rotem D. and Wong L. 'Bit Transposed files', In Proceedings of International Conference on Very Large Databases, pp. 448-457, 1985.

Wu, C.-L., Koh, J.-L., and An, P.Y., ‘Improved sequential pattern mining using an extended bitmap representation’, In Proceedings of the International Conference on Database and Expert System Applications, pp. 776—785, 2005.

Wu, K. , Otoo, E. and Shoshani, A., ‘A performance comparison of bitmap indexes’, In Proceedings of the 2001 ACM CIKM International Conference on Information and Knowledge Management, Atlanta, Georgia, USA, November 5-10, ACM, pp. 559-561, 2001.

Wu, K., Otoo, E. and Shoshani, A., ‘An Efficient Compression Scheme for Bitmap Indices’, Technical Report LBNL-49626, ACM Transactions on Database Systems (TODS), Vol. 31, pp. 1-38, 2006.

Wu, K., Zhang, W.M., Perevoztchikov, V., Lauret, J.and Shoshani, A., ‘The Grid Collector: Using an Event Catalog to Speedup User Analysis in Distributed Environment’, In Computing in High Energy and Nuclear Physics (CHEP) 2004, Interlaken, Switzerland, 2004.

Wu, K., Koegler, W., Chen J., and Shoshani ,A., ‘Using bitmap index for interactive exploration of large datasets’, In Proceedings of the 15th international conference on Scientific and statistical database management (SSDBM),IEEE Computer Society, pp. 65-74, 2003.

Wu, K., Otoo, E., and Shoshani, A., ‘Compressed bitmap indices for efficient query processing’, Technical Report LBNL-56936, Berkeley Lab, Berkeley, California, USA, 2001.

Wu, K., Otoo, E., and Shoshani, A., ‘Optimizing bitmap indices with efficient compression’, ACM Transactions on Database Systems, Vol. 31, No.1, pp. 1-38, 2006.

Wu, K., Otoo, E., Shoshani, A., and Nordberg, H., ‘Notes on design and implementation of compressed bit vectors’, Technical Report LBNL/PUB-3161, Lawrence Berkeley National Laboratory, Berkeley, CA, 2001

Wu, K., Otoo, E.J., and Shoshani, A., ‘Compressing Bitmap Indexes for Faster Search Operations’, In Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM), IEEE Computer Society Press, pp. 99-108, 2002.

Wu, K., Otoo, E.J., and Shoshani, A., ‘On the Performance of Bitmap Indices for High Cardinality Attributes’, In Proceedings of the International Conference on Very Large Data Bases VLDB'2004, Toronto, Canada. Morgan Kaufmann, pp. 24 – 35, 2004.

Wu, K., Shoshani, A., and Otoo, E. J., ‘Word aligned bitmap compression method, data structure, and apparatus’, US Patent 6,831,575. 2004.

Wu, K.-L. and Yu, P.S, ‘Range-Based Bitmap Indexing for High Cardinality Attributes with Skew’, In Proceedings of the 22nd International Computer Software and Application Conference (COMPSAC), pp. 61–67, 1998.

Wu, M. C. , ‘Query optimization for selections using bitmaps’, In Proceedings of the ACM Conference on Management of Data (SIGMOD), ACM, New York, pp. 227-238, 1999.

Wu, M.C. and Buchmann, A. P., ‘Research Issues in Data Warehousing’, In Datenbanksystem in Bro, Technik und Wissenschaft, pp. 61-82, 1997.

Wu, M.C., and Buchmann, A.P., ‘Encoded Bitmap Indexing for Data Warehouses’, In Proceedings of the International Conference on Data Engineering, Orlando, Florida, IEEE Computer Society Press, pp. 220-230, 1998.

Yao, S. B., ‘Approximating block Accesses in Database Organizations’, Communication of the ACM, Vol. 20, No. 4, pp. 260-261, 1978.

Zaki, M. J. and Wang, J. T. L., ‘Special issue on bioinformatics and biological data management’, Information Systems, Vol. 28, pp.241–367, 2003.

Ziv, J., and Lempel, A., ‘A universal algorithm for sequential data compression’, IEEE Transactions on Information Theory, Vol.23, No.3, pp.337-343, 1977.

Appendix - A

A.1 Code for WAH and BBC Compression Algorithms

```
#define WAH_RUN_0 "0000000000000000000000000000000"
#define WAH_RUN_1 "1111111111111111111111111111111"
#define BBC32_0   "00000000000000000000000000000000"
#define BBC32_1   "11111111111111111111111111111111"
#define BBC24_0   "000000000000000000000000"
#define BBC24_1   "111111111111111111111111"
#define BBC16_0   "0000000000000000"
#define BBC16_1   "1111111111111111"
#define BBC8_0    "00000000"
#define BBC8_1    "11111111"

#define COUNT 10000000
#include<stdio.h>
#include<time.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#include "compress.h"

int wah_counter = 0;

char array[32];


void reverseString(char *string)
{
      int left = 0;
      char temp;
      int right = strlen(string) - 1;
      while(left < right){
            temp = string[left];
            string[left] = string[right];
            string[right] = temp;
            left++;
            right--;
      }
}

void decimalToBinary(int number,char *binNum,int length)
{
      int i = 0;
      while(number){
            binNum[i++] = number % 2 + 48;
            number = number / 2;
      }
      binNum[i] = 0;
      for(i = strlen(binNum); i < length; i++)
            binNum[i] = '0';
      binNum[i] = 0;
      reverseString(binNum);
      return;
}
```

```
int binaryToDecimal(char *strNum)
{
      int i = strlen(strNum);
      int index = 0;
      int number = 0;
      while(i--){
            if(strNum[i] - 48){
                  number += (int)pow(2,index);
            }
            index++;
      }
      return number;
}

void compress_wah(char *input_file_name, char *output_file_name)
{
      bool run_0=false;
      bool run_1=false;
      int counter=0;
      FILE *fp = fopen(input_file_name,"r");
      if(!fp){
            perror("Error in opening input file");
            exit(1);
      }
      FILE *fo = fopen(output_file_name,"w");
      if(!fo){
            fclose(fp);
            perror("Error in opening output file");
            exit(1);
      }

      char temp[32];
      int n = 0;
      clock_t start = clock();
      while(!feof(fp)){
            n = fread(array,1,sizeof(array) - 1,fp);
//          if(!feof(fp)){
                  if(n < 0){
                        perror("Error in reading file");
                        fclose(fp);
                        fclose(fo);
                        exit(2);
                  }
                  array[n] = 0;
                  if(n < 31){
                        //active word
                        int len = 31 - strlen(array);
                        for(int i = 0; i < len + 1; i++)
                              fprintf(fo,"%s","0");
                        fprintf(fo,"%s",array);
                        clock_t finish = clock();
                        printf("Time taken = %2.7lf
seconds\n",(double)(finish - start)/CLOCKS_PER_SEC);
                        int size = ftell(fo);
                        long usize = ftell(fp);
                        printf("Uncompressed File Size = %d bytes\n",
usize/8);
                        printf("Compressed Size = %d bytes\n",size/8);
                        printf("Percentage Compressed %2.2lf%%
\n",100*(double)(usize-size)/usize);
                        break;
                  }
                  if(!strcmp(array,WAH_RUN_0))
                  {
                        if(run_1)
                        {
                              //write run1 1
                              decimalToBinary(counter,temp,30);
                  //          temp[31]=0;
                              fprintf(fo,"%s%s","11",temp);
                              run_1=false;
                              counter=1;
                        }
                        else if(run_0)
                              counter++;
                        else
                              counter=1;
                        run_0=true;
                  }
                  else if(!strcmp(array,WAH_RUN_1))
                  {
                        if(run_0)
                        {
                              //write run 0
                              decimalToBinary(counter,temp,30);
                  //          temp[31]=0;
                              fprintf(fo,"%s%s","10",temp);
                              run_0=false;
                              counter=1;
                        }
                        else if(run_1)
                              counter++;
                        else
                              counter=1;
                        run_1=true;
                  }
                  else
                  {
                        if(run_1)
                        {
                              //write run1 1
                              decimalToBinary(counter,temp,30);
                  //          temp[31]=0;
                              fprintf(fo,"%s%s","11",temp);
                              run_1=false;
                        }
                        if(run_0)
                        {
                              //write run 0
                              decimalToBinary(counter,temp,30);
                  //          temp[31]=0;
                              fprintf(fo,"%s%s","10",temp);
                              run_0=false;
                        }
                        //write literal
                        fprintf(fo,"%s%s","0",array);
                        counter=0;
                        run_0=false;
                        run_1=false;
                  }

            }
//    }
      fclose(fp);
      fclose(fo);
}


int get_literal_count(FILE *fp ,char **literal_buffer)
{
      //int loc = ftell(fp);
      //bool condition = true;
      char temp[9];
      int n = 0;
      strcpy(*literal_buffer,"");
      int literal_count = 0;
      bool first = true;
      while(true){
            n = fread(temp,1,8,fp);
            temp[n] = 0;
            if(n < 8){
                  fseek(fp,-(n),SEEK_CUR);
                  break;
            }
            if(!strcmp(temp,BBC8_0) || !strcmp(temp,BBC8_1)){
                  fseek(fp,-8,SEEK_CUR);
                  break;
            }
            strcat(*literal_buffer,temp);
            if(first){
                  first = false;
                  char *ptr = strstr(temp,"0");
                  ptr = strstr(ptr + 1,"0");
                  if(!ptr)
                        return -(strstr(temp,"0")-temp);
                  ptr = strstr(temp,"1");
                  ptr = strstr(ptr + 1,"1");
                  if(!ptr)
                        return -(strstr(temp,"1")-temp);
            }
            literal_count++;
            if(literal_count > 15)
                  break;
      }
      //fseek(fp,loc,SEEK_SET);
      return literal_count;
}
```

```
void set_run_3_4_header(int counter, char **array)
{
      counter = counter - 4;
      int bit_size = (int)ceil((double)log(counter) / log(2));
      bit_size = (int)ceil((double)bit_size / 7);
      *array = new char[bit_size * 8 + 1];
      char *temp=new char[bit_size * 9];
      strcpy(*array,"");
      decimalToBinary(counter,temp,0);
      int i = 0;
      while(strlen(temp + i) > 7){
            strcat(*array,"1");
            strncat(*array,temp + i,7);
            i += 7;
      }
      int len = 7 - strlen(temp + i);
      while(len >= 0){
            strcat(*array,"0");
            len--;
      }
      strcat(*array,temp + i);
      delete temp;
      return;
}

void compress_bbc(char *input_file_name, char *output_file_name)
{
      int counter=0;
      FILE *fp = fopen(input_file_name,"r");
      if(!fp){
            perror("Error in opening input file");
            exit(1);
      }
      FILE *fo = fopen(output_file_name,"w");
      if(!fo){
            fclose(fp);
            perror("Error in opening output file");
            exit(1);
      }

      char temp[33];
      char literal_array[5];

      char *literal_buffer = new char[121];
      int n = 0;
      char *run_header;
      int run_fill_size = 0;
      int literal_length = 0;
      clock_t start = clock();
      while(!feof(fp)){
            n = fread(temp,1,32,fp);
            if(n < 0){
                  perror("Error in reading");
                  exit(2);
            }
            temp[n] = 0;
            if(n < 32){
                  fprintf(fo,"%s%s","10000100",temp);
                  // code if read less than 32 bits
                  break;
            }
            else{
                  if(!strcmp(temp,BBC32_0)){
                        run_fill_size = 4;
                        while(true){
                              n = fread(temp,1,8,fp);
                              temp[n] = 0;
                              if(n < 8){

                                    fprintf(fo,"%s","00100001");

      set_run_3_4_header(run_fill_size,&run_header);
                                    fprintf(fo,"%s%s",run_header,temp);
                                    delete run_header;
                                    break;
                              }
                              if(!strcmp(temp,BBC8_0))
                                    run_fill_size++;
                              else{
                                    fseek(fp,-8,SEEK_CUR);

      literal_length=get_literal_count(fp,&literal_buffer);
                                    if(literal_length<=0)
                                    {
                                          //run type 4
                                          literal_length=-
literal_length;

      decimalToBinary(literal_length,literal_array,3);

      fprintf(fo,"%s%s","00010",literal_array);

      set_run_3_4_header(run_fill_size,&run_header);
                                          fprintf(fo,"%s",run_header);
                                          delete run_header;

      //fprintf(fo,"%s%s%s","01011",literal_array);
                                    }
                                    else
                                    {
                                          //run type 3

      decimalToBinary(literal_length,literal_array,4);

      fprintf(fo,"%s%s","0010",literal_array);

      set_run_3_4_header(run_fill_size,&run_header);

      fprintf(fo,"%s%s",run_header,literal_buffer);
                                          delete run_header;

                                    }
```

```
                              }
                        }
                        //run 3 or 4
                  }
                  else if(!strcmp(temp,BBC32_1)){

                        run_fill_size = 4;
                        while(true){
                              n = fread(temp,1,8,fp);
                              temp[n] = 0;
                              if(n < 8){

                                    fprintf(fo,"%s","00110001");

      set_run_3_4_header(run_fill_size,&run_header);
                                    fprintf(fo,"%s%s",run_header,temp);
                                    delete run_header;
                                    break;
                              }
                              if(!strcmp(temp,BBC8_1))
                                    run_fill_size++;
                              else{
                                    fseek(fp,-8,SEEK_CUR);

      literal_length=get_literal_count(fp,&literal_buffer);
                                    if(literal_length<=0)
                                    {
                                          //run type 4
                                          literal_length=-
literal_length;

      decimalToBinary(literal_length,literal_array,3);

      fprintf(fo,"%s%s","00011",literal_array);

      set_run_3_4_header(run_fill_size,&run_header);
                                          fprintf(fo,"%s",run_header);
                                          delete run_header;

      //fprintf(fo,"%s%s%s","01011",literal_array);
                                    }
                                    else
                                    {
                                          //run type 3

      decimalToBinary(literal_length,literal_array,4);

      fprintf(fo,"%s%s","0011",literal_array);

      set_run_3_4_header(run_fill_size,&run_header);

      fprintf(fo,"%s%s",run_header,literal_buffer);
                                          delete run_header;

                                    }
```

```
                                }
                          }
                          //run 3 or 4


                          //run 3 or 4
                    }
                    else{
                          if(!strncmp(temp,BBC24_0,24)){
                                fseek(fp,-8,SEEK_CUR);
                                literal_length =
get_literal_count(fp,&literal_buffer);
                                if(literal_length<=0)
                                {
                                      literal_length=-literal_length;

      decimalToBinary(literal_length,literal_array,3);

      fprintf(fo,"%s%s","01011",literal_array);
                                }
                                else
                                {

      decimalToBinary(literal_length,literal_array,4);

      fprintf(fo,"%s%s%s","1011",literal_array,literal_buffer);
                                // run 1 or 2 for length = 24 bits
                                }
                          }
                          else if(!strncmp(temp,BBC24_1,24)){
                                fseek(fp,-8,SEEK_CUR);
                                literal_length =
get_literal_count(fp,&literal_buffer);
                                if(literal_length<=0)
                                {
                                      literal_length=-literal_length;

      decimalToBinary(literal_length,literal_array,3);

      fprintf(fo,"%s%s","01111",literal_array);
                                }
                                else
                                {

      decimalToBinary(literal_length,literal_array,4);

      fprintf(fo,"%s%s%s","1111",literal_array,literal_buffer);
                                //run 1 or 2 for length = 24 bits
                                }
                          }
                          else if(!strncmp(temp,BBC16_0,16)){
                                fseek(fp,-16,SEEK_CUR);
                                literal_length =
get_literal_count(fp,&literal_buffer);
                                if(literal_length<=0)
                                {
                                      literal_length=-literal_length;
      decimalToBinary(literal_length,literal_array,3);

      fprintf(fo,"%s%s","01010",literal_array);

                                }
                                else
                                {

      decimalToBinary(literal_length,literal_array,4);

      fprintf(fo,"%s%s%s","1010",literal_array,literal_buffer);
                                //run 1 or 2 for length = 16 bits
                                }
                          }
                          else if(!strncmp(temp,BBC16_1,16)){
                                fseek(fp,-16,SEEK_CUR);
                                literal_length =
get_literal_count(fp,&literal_buffer);
                                if(literal_length<=0)
                                {
                                      literal_length=-literal_length;

      decimalToBinary(literal_length,literal_array,3);

      fprintf(fo,"%s%s","01110",literal_array);

                                }
                                else
                                {

      decimalToBinary(literal_length,literal_array,4);

      fprintf(fo,"%s%s%s","1110",literal_array,literal_buffer);
                                }
                                //run 1 or 2 for length = 16 bits
                          }
                          else if(!strncmp(temp,BBC8_0,8)){
                                fseek(fp,-24,SEEK_CUR);
                                literal_length =
get_literal_count(fp,&literal_buffer);
                                if(literal_length<=0)
                                {
                                      literal_length=-literal_length;

      decimalToBinary(literal_length,literal_array,3);

      fprintf(fo,"%s%s","01001",literal_array);

                                }
                                else
                                {

      decimalToBinary(literal_length,literal_array,4);

      fprintf(fo,"%s%s%s","1001",literal_array,literal_buffer);
                                }
                                //run 1 or 2 for length = 8 bits
                          }
                          else if(!strncmp(temp,BBC8_1,8)){
                                fseek(fp,-24,SEEK_CUR);
                                literal_length = 
get_literal_count(fp,&literal_buffer);
                                if(literal_length<=0)
                                {
                                      literal_length=-literal_length;

      decimalToBinary(literal_length,literal_array,3);

      fprintf(fo,"%s%s","01101",literal_array);

                                }
                                else
                                {

      decimalToBinary(literal_length,literal_array,4);

      fprintf(fo,"%s%s%s","1101",literal_array,literal_buffer);
                                }
                                //run 1 or 2 for length = 8 bits
                          }
                          else{
                                fseek(fp,-32,SEEK_CUR);
                                literal_length = 
get_literal_count(fp,&literal_buffer);
                                if(literal_length<=0)
                                {
                                      literal_length=-literal_length;

      decimalToBinary(literal_length,literal_array,3);

      fprintf(fo,"%s%s","01000",literal_array);

                                }
                                else
                                {

      decimalToBinary(literal_length,literal_array,4);

      fprintf(fo,"%s%s%s","1000",literal_array,literal_buffer);
                                }
                                //fseek(fp,-24,SEEK_CUR);
                                //literal
                          }
                    }
              }
      }
      clock_t finish = clock();
      printf("Time taken = %2.7lf seconds\n",(double)(finish - 
start)/CLOCKS_PER_SEC);
      int size = ftell(fo);
      long usize = ftell(fp);
      printf("Uncompressed File Size = %d bytes\n", usize/8);
      printf("Compressed Size = %d bytes\n",size/8);
      printf("Percentage Compressed %2.2lf%% \n\n",100*(double)(usize-
size)/usize);

}

int main()
{

      printf("\nDoing WAH Compression\n\n");
      compress_wah("input_data.txt","wah.txt");
      printf("\nDoing BBC Compression\n\n");
      compress_bbc("input_data.txt","bbc.txt");
      return 0;
}
```

A.2 Code for Gray Code Ordering Algorithm

```
#include<stdio.h>
#include<stdlib.h>

int rows,cols;

void findcols(FILE *fp)
{
      cols=0;
      char c;
      while(1)
      {
            fscanf(fp,"%c",&c);
            if(c==' ')
            continue;

            else if(c=='\n')
            goto final;

            else
            cols++;
      }

final:
rewind(fp);
}

void findrows(FILE *fp)
{
    rows=0;
      char c;
      while(1)
      {
            if(feof(fp))
            goto final;

            fscanf(fp,"%c",&c);

            if(c=='\n')
                rows++;
```

```
     }

final:
rows++;
rewind(fp);
}

void reversing(int i,int j)
{
      if(i>=j)
            return;

      FILE *f1;
      FILE *f2;

      int temp1;
      int temp2;

      int k;
      char c,ctemp;

      f1=fopen("o1.txt","r+");
      f2=fopen("o1.txt","r+");


      temp1=(2*cols+1)+((2*cols+2)*(i-2))+2;
      temp2=(2*cols+1)+((2*cols+2)*(j-2))+2;

      fseek(f1,temp1,1);
      fseek(f2,temp2,1);

      while(1)
      {
            if(i==j)
            break;

            for(k=1;k<=cols;k++)
            {
                  fscanf(f2,"%c",&c);
                  ctemp=c;
                  fseek(f2,-1,1);
                  fscanf(f1,"%c",&c);
                  fprintf(f2,"%c",c);
                  fseek(f1,-1,1);
                  fprintf(f1,"%c",ctemp);

                  if(k==cols)
                  break;

                  fseek(f1,1,1);
                  fseek(f2,1,1);
            }

            fseek(f1,-(2*cols-1),1);
            fseek(f2,-(2*cols-1),1);
            i++;
            j--;

            if(i>j)
            break;

            fseek(f1,(2*cols+2),1);
            fseek(f2,-(2*cols+2),1);

      }


      fclose(f1);
      fclose(f2);
}

void gcsort(int start,int end,int b)
{
      if(start>=end)
            return;

      FILE *f1;
      FILE *f2;

      int i;
      int j;

      char n1,n2,c,ctemp;

      int k;
      int temp1;
      int temp2;

      i=start;
      j=end;

      f1=fopen("o1.txt","r+");
      f2=fopen("o1.txt","r+");

      temp1=(2*cols+1)+((2*cols+2)*(i-2))+(2*b);
      temp2=(2*cols+1)+((2*cols+2)*(j-2))+(2*b);

      fseek(f1,temp1,1);
      fseek(f2,temp2,1);

      while(i<j)            //positioning f1,f2 to swap loc
      {

            while(1)
            {
                  if(j==start)
                  {
                        fscanf(f2,"%c",&n2);
                        break;
                  }
```

```
          fscanf(f2,"%c",&n2);

          if(n2=='0')
                break;

          j--;
          fseek(f2,-(2*cols+3),1);


    }


    while(1)
    {
          if(i==end)
          {
                fscanf(f1,"%c",&n1);
                break;
          }

          fscanf(f1,"%c",&n1);

          if(n1=='1')
          break;

          i++;
          fseek(f1,(2*cols+1),1);

    }

    fseek(f1,-1,1);
    fseek(f2,-1,1);

    if(i<j)
    {
          fseek(f1,-(2*b-2),1);
          fseek(f2,-(2*b-2),1);

          for(k=1;k<=cols;k++)
          {
                fscanf(f2,"%c",&c);
                ctemp=c;
                fseek(f2,-1,1);
                fscanf(f1,"%c",&c);
                fprintf(f2,"%c",c);
                fseek(f1,-1,1);
                fprintf(f1,"%c",ctemp);

                if(k==cols)
                break;

                fseek(f1,1,1);
                fseek(f2,1,1);
          }
          fseek(f1,-(2*(cols-b)+1),1);
          fseek(f2,-(2*(cols-b)+1),1);
    }
```

```
	}

	if(b<cols)
	{
		gcsort(start,j,b+1);
		gcsort(j+1,end,b+1);
		reversing(j+1,end);
	}

	fclose(f1);
	fclose(f2);
}




void main()
{


	FILE *f1;

	f1=fopen("o1.txt","r+");

	findrows(f1);
	findcols(f1);

	fclose(f1);

	gcsort(1,rows,1);
}
```

A.3 Code for Multi-Component Encoding

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>

void main()
{
	clrscr();
	int cols,rows,i,j,k,base1,base2,rem,quo,num,r;
	FILE *fr,*fw;
	char c,ch1,ch2,ch3;

	fr=fopen("SING.CPP","r");
	fw=fopen("king.cpp","w");

	cols=0;
	while(1)
	{
		fscanf(fr,"%c",&c);
		if(c=='0' || c=='1')
		{
```

```
            cols++;
      }
      if(c=='\n')
      break;
}

printf("%2d",cols);


/*We reduce the size of each rows of bitmap from 12 bits in each
  row to 7 bits using multicomponent equality encoded index */

  rewind(fr);

  rows=0;
  while(1)
  {
      c=fgetc(fr);

      if(c==EOF)
      break;

      if(c=='\n')
      rows++;
  }
 rows=rows+1;
 printf("%2d",rows);

 rewind(fr);

 base1=4;  //NOTE base1 is larger of 2 bases
 base2=3;


 for(i=1;i<=rows;i++)
 {
      for(j=cols-1;j>=0 ;j--)
      {
            fscanf(fr,"%c",&c);
            fscanf(fr,"%c",&c);
            if(c=='1')
            {
                  num=j;
            }
      }
      quo=num/base1;
      rem=num%base1;

      ch1='0';
      ch2='1';
      ch3=' ';
      for(k=base2-1;k>=0;k--)
      {
            fprintf(fw,"%c",ch3);
            if(k==quo)
            {
                  fprintf(fw,"%c",ch2);
```

```
                        }
                        else
                        {
                              fprintf(fw,"%c",ch1);
                        }
                  }

                  for(r=base1-1;r>=0;r--)
                  {
                        fprintf(fw,"%c",ch3);
                        if(r==rem)
                        {
                              fprintf(fw,"%c",ch2);
                        }
                        else
                        {
                              fprintf(fw,"%c",ch1);
                        }
                  }
                  fscanf(fr,"%c",&c);
                  fprintf(fw,"%c",'\n');
            }

}
```

A.4 Code for Synthesizing Student Records

```
#include<stdio.h>
#include<stdlib.h>


main(int argc,char *argv[])

{char *year[]={"2000","2001","2002","2003","2004","2005"};
char *A[]={"A1","A2","A3","A4","A5","A6","A7","A8"};
char *B[]={"B1","B2","B3","B4","B5"};
char *C[]={"C2","C5","C6","C7"};
char *dual1[8][5];
char *dual2[8][8];
char *dual3[4][4];
char *sing1[8];
char *sing2[5];
char *all[130];
char *last[999];
int a,i,c,d,count,j,l,ran,k,ran2,ran3,ran4;
char name[16]="",idno[12]="";
char
*host[]={"RM","BD","KR","GN","SK","VY","BG","VK","AK","RP","ML","MB"};
FILE *f1,*f2,*f3,*f4,*f5,*f6;
char *dig[]={"0","1","2","3","4","5","6","7","8","9"};
f1=fopen(argv[1],"w");
f2=fopen(argv[2],"w");
f3=fopen(argv[3],"w");
f4=fopen(argv[4],"w");
f5=fopen(argv[5],"w");
f6=fopen(argv[6],"w");
```

```
/*generating the disciplines*/

for(i=0;i<=7;i++)
for(j=0;j<=4;j++)
{
dual1[i][j]=(char *)malloc(4);
strcat(dual1[i][j],B[j]);
strcat(dual1[i][j],A[i]);
}

for(i=0;i<=7;i++)
for(j=0;j<=7;j++)
{if(i!=j)
{
dual2[i][j]=(char *)malloc(4);
strcat(strcat(dual2[i][j],A[i]),A[j]);

}
}

for(i=0;i<=3;i++)
for(j=0;j<=3;j++)
{if (i!=j)
{
dual3[i][j]=(char *)malloc(4);
strcat(strcat(dual3[i][j],C[i]),C[j]);
}
}

for(i=0;i<=7;i++)
{
sing1[i]=(char *)malloc(4);
strcat(strcat(sing1[i],A[i]),"PS");
}

for (j=0;j<=4;j++)
{
sing2[j]=(char *)malloc(4);
strcat(strcat(sing2[j],B[j]),"TS");
}

k=0;

/* moving all the disciplines into one array pointer*/


for(i=0;i<=7;i++)
for(j=0;j<=4;j++)
{
all[k]=(char *)malloc(sizeof(dual1[i][j]));
strcat(all[k], dual1[i][j]);
free(dual1[i][j]);
k=k+1;
}

for(i=0;i<=7;i++)
for(j=0;j<=7;j++)
```

```
{if (i!=j)
{
all[k]=(char *)malloc(sizeof(dual2[i][j]));
strcat(all[k],dual2[i][j]);
free(dual2[i][j]);
k=k+1;
}
}

for(i=0;i<=3;i++)
for(j=0;j<=3;j++)
{if(i!=j)
{
all[k]=(char *)malloc(sizeof(dual3[i][j]));
strcat(all[k],dual3[i][j]);
free(dual3[i][j]);
k=k+1;
}
}

for(i=0;i<=7;i++)
{
all[k]=(char *)malloc(sizeof(sing1[i]));
strcat(all[k],sing1[i]);
free(sing1[i]);
k=k+1;
}
for (j=0;j<=4;j++)
{
all[k]=(char *)malloc(sizeof(sing2[j]));
strcat(all[k],sing2[j]);
free(sing2[j]);
k=k+1;
}

printf("discipline gen finished\n");
l=0;
/*generating the id last three*/
for(i=0;i<=9;i++)
for(j=0;j<=9;j++)
for(k=0;k<=9;k++)
{if(i!=0||j!=0||k!=0)
{
last[l]=(char *)malloc(3);
strcpy(last[l],dig[i]);
strcat(strcat(last[l],dig[j]),dig[k]);
l=l+1;
}
}

/*
for(i=0;i<=998;i++)
printf("%s\n",last[i]);
*/
```

```
//==============================================================
/*generating file 1  */

count=0;
c=0;
d=15;
for(i=0;i<2000000;i++)
 {
if (count<d)
  {  a=97+(int)(26.0*rand()/(RAND_MAX+1.0));
name[count]=(char)a;
 count=count+1;
 }
else
{if(c>99)
break;
name[count]='\0';
ran=0+(int)(12.0*rand()/(RAND_MAX+1.0));
ran2=0+(int)(6.0*rand()/(RAND_MAX+1.0));
ran3=0+(int)(998.0*rand()/(RAND_MAX+1.0));
ran4=0+(int)(121.0*rand()/(RAND_MAX+1.0));
strcpy(idno,year[ran2]);
strcat(idno,all[ran4]);
strcat(idno,last[ran3]);
strcat(idno,"\0");
fprintf(f1,"%s %s %s\n",idno,name,host[ran]);
count=0;
d=(int)(1+(int)(15.0*rand()/(RAND_MAX+1.0)));
c=c+1;
}
}
fclose(f1);
printf("finished generating 1st file\n");
}
```

A.5 Code for Binning Algorithms

```
//query.h

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

extern int b_num,b_width,rec_no,max,min;
int **bin_ranges;
int numberofbins;
int rec_read;
int temp;
int b_num,b_width,rec_no,max,min;
FILE *OutFile;

struct ind
{
      int seek;
      int recNO;
      };
```

```
typedef struct ind inde;

void *binning(char *[]);
unsigned int power(int,int);
void queries(char *[],unsigned int *(*)[]);
void categorize(unsigned int *(*)[],int,FILE *);
void create_indexFile(char *[]);
void print_record(int,int,int,char *[],int,int,int);
int id_num(char *);
int compar(const void *,const void *);
int bitcount(unsigned int x);
void Print_Binary(int);
void Print_Bin_Byte(int);
```

//main.c

```
#include "query.h"

main(int argc,char *args[])
{
int id,rm,temp;
int i=0;
unsigned int* (*bin)[];
void *tmp;

bin=binning(args);
tmp=bin;
queries(args,bin);
}
```

//Index of data file

```
#include<stdio.h>
#include <string.h>
#include <stdlib.h>

struct ind
{
      int seek;
      int recNO;
      };
typedef struct ind inde;

main(int argc, char *argv[])
{
      inde i_w, i_r;
      char ch,id[40],name[40],bhawan[2];
      int rec_read = 2,record=0,n=0;
      FILE *recFile = fopen(argv[1],"r");
      FILE *indexFile = fopen("index","w");

      i_w.seek = ftell(recFile);
      i_w.recNO = 1;
      fwrite(&i_w,sizeof(i_w),1,indexFile);
      while((ch=getc(recFile))!=EOF)
      {
            if(ch=='\n')
            {
                  i_w.seek = ftell(recFile) + 1;
                  i_w.recNO = rec_read;
                  rec_read++;
                  fwrite(&i_w,sizeof(i_w),1,indexFile);
                  }
            }
            fclose(indexFile);

///RETRIEVAL OFRECORDS
      printf("TOTAL RECORDS = %d", rec_read-2);
rread:
      printf("\nEnter the Record to be retrieved :");
      scanf("%d",&record);
      if(rec_read-2 < record || record <= 0)
      {
            printf("RECORD DOESN'T EXIST\n");
            printf("\npress 1 to try again or 0 to exit:");
            scanf("%d",&n);
            if(n == 0) exit(1);
            else goto rread;
            }
      fopen("index","r");
      do
      {
            fread(&i_r,sizeof(i_r),1,indexFile);
            }while(i_r.recNO!=record);

      fseek(recFile,i_r.seek-1,0);
      if(record == 1) rewind(recFile);
      fscanf(recFile,"%s %s %s",id,name,bhawan);
      printf("%s  %s  %s\n",id,name,bhawan);
      printf("\npress 1 to try again or 0 to exit:");
      scanf("%d",&n);
      if(n == 1) {fclose(indexFile);goto rread;}
      else exit(1);
      fclose(recFile);
      fclose(indexFile);
      }
```

Biography of the Supervisor

Dr. Navneet Goyal is an Associate Professor in the Department of Computer Science and Information Systems at BITS, Pilani. He is also Assistant Chief of Computer Assisted House Keeping Unit at BITS, Pilani.

Dr. Goyal obtained his doctorate from Indian Institute of Technology, Roorkee in 1995. After completing his Ph.D, he joined BITS-Pilani in 1995, where he has been involved in teaching, research and administration. He has published more than 15 research papers in national and international journals in Applied Mathematics, Databases, Data Warehousing, and Data Mining. He presently teaches courses on database systems, data warehousing, and data mining to undergraduate and graduate students at BITS. He also holds a PG Diploma in Health Systems Management from Tulane University, USA.

Biography of the Candidate

Yashvardhan Sharma has completed his first degree from BITS-Pilani, India with first division in the year 1999 and M.E. (Software Systems) from BITS-Pilani with first class in the year 2001. He has a teaching experience of over 8 years to undergraduate and graduate students at BITS-Pilani. Currently he is working as Lecturer in Computer Science and Information Systems group at BITS-Pilani. His areas of interest include Data Warehousing Performance Enhancing Techniques, Data Mining, Object Oriented Software Engineering and Application Programming.