Compression Enhancement of HIBASE Technique Using HUFFMAN Coding

by

Ahsan Habib

MASTER OF SCIENCE IN INFORMATION AND COMMUNICATION TECHNOLOGY

Institute of Information and Communication Technology

BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY

February, 2012

The thesis titled **"Compression Enhancement of HIBASE Technique Using HUFFMAN Coding",** submitted by Ahsan Habib, Roll No. 10063111P, Session: October 2006 has been accepted as satisfactory for the partial fulfillment of the requirements for the degree of Master of Science in Information and Communication Technology on February 25, 2012.

Board of Examiners

Candidate's declaration

It is hereby declared that this thesis or any part of it has not been submitted elsewhere for the award of any degree or diploma, does not contain any unlawful statements.

Signature

(Ahsan Habib)

Acknowledgement

First of all, I want to express my sincere gratitude to my supervisor, Professor Dr. Abu Sayed Md. Latiful Hoque, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka for making the research as an art of exploiting new ideas and technologies on top of the existing knowledge in the field of database compression. He provides me moral courage and excellent guideline that make it possible to complete the work. His profound knowledge and expertise in this field provide me many opportunities to learn new things to build my career as a researcher.

I also express my gratitude to Professor Dr. S. M. Lutful Kabir, Director of Institute of Information and Communication Technology (IICT), BUET for providing me enough lab facilities to make necessary experiments of my research. I am grateful to my Board of Examiners, their valuable comments helps me a lot to improve my work.

I am also grateful to my mother, wife, other family members and colleagues for giving continuous support.

Abstract

Storage requirement for database system is a problem for many years. Storage capacity is being increased continually, but the enterprise and service provider data need double storage every six to twelve months. It is a challenge to store and retrieve this increased data in an efficient way. Reduction of the data size without losing any information is known as loss-less data compression.

In this thesis we have presented a loss-less compression technique namely H-HIBASE (further compression of HIBASE technique using HUFFMAN Coding). Due to disk based compression H-HIBASE support very large database with acceptable storage volume. Insertion, deletion and update mechanisms on the architecture have been presented and analyzed. The architecture executes query directly on compressed data and it is capable of executing all types of SQL queries. The experimental evaluation has been performed with synthetic and real data. The experimental result has been compared with DHIBASE and widely used Oracle database. Our target was to handle relations and justify the storage requirements and query time in comparison with DHIBASE and Oracle database.

We evaluated the storage performance in comparison with DHIBASE and Oracle database. The storage performance that has been achieved in H-HIBASE is 25 to 40 percent better than the Oracle database for real and synthetic data. The query performance that has been achieved in H-HIBASE is 10 to 25 percent better than that of DHIBASE.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Storage requirement for database system is a problem for many years. Storage capacity is being increased continually, but the enterprise and service provider data need double storage every six to twelve months [1]. It is a challenge to store and retrieve this increased data in an efficient way. Reduction of the data size without losing any information is known as loss-less data compression. This is potentially attractive in database systems for two reasons:

- Storage cost reduction
- Performance improvement

The reduction of storage cost is obvious. The performance improvement arises as the smaller volume of compressed data may be accommodated in faster memory than its uncompressed counterpart. Only a smaller amount of compressed data needs to be transferred and/or processed to effect any particular operation.

Most of the large databases are often in tabular form. The operational databases are of medium size whereas the typical size of fact tables in a data warehouse is generally huge [2]. These data are write once-read many type for further analysis. Problem arises for high-speed access and high-speed data transfer. The conventional database technology cannot provide such performance. We need to use new algorithms and techniques to get attractive performance and to reduce the storage cost. High performance compression algorithm, necessary retrieval and data transfer technique can be a candidate solution for large database management system. It is difficult to combine a good compression technique that reduces the storage cost while improving performance.

1.1 Background

A number of research works [3, 4, 5, 6] are found on compression based Database Management Systems (DBMS). Commercial DBMS uses compression to a limited extent to improve performance [7]. Compression can be applied to databases at the relation level, page level and the tuple or attribute level. In page level compression methods the compressed representation of the database is a set of compressed tuples. An individual tuple can be compressed and decompressed within a page. An approach to page level compression of relations and indices is given in [8]. The Oracle Corporation introduces disk-block based compression technique [9] to manage large database. Complex SQL (Structured Query Language) queries cannot be carried out on these databases in compressed form.

SQL:2003 [10] supports many different types of operations. Compression based systems like High Compression Database System (HIBASE) [11], Three Layer Model [12] and Columnar Multi Block Vector Structure (CMBVS) [2] have limited number of query statements compared to SQL.

1.2 Problem Definition

The HIBASE (**Hi**gh Compression Data**base** System) [13] approach is a compression technique for Main Memory Database Management System (MMDBMS) [14], supports high performance query operations on relational structure [15]. Disk-based system (DHIBASE) is an extension of HIBASE architecture [15]. The structure stores database in column wise format so that the unnecessary columns need not to be accessed during query processing and also restructuring the database schema will be easy. The dictionary space overhead is excessive for both HIBASE and DHIBASE systems. HIBASE and DHIBASE compression techniques simply replaces the attribute values in a tuple with fixed length code-words [16][17]. However, fixed length coding system is not an optimal compression technique it does not consider the frequency of occurrence of the values. Thus HIBASE and DHIABSE require higher space in the compressed database. This higher storage requirement can be avoided if we use Huffman code-words [18]. Moreover, using Huffman code-word will ensure optimal compression as well as High performance operation [19][20]. As we know Huffman algorithm generates an optimal tree [18][19], hence the compression will be optimized. However, the use of Huffman coding could increase the query complexity in HIBASE and DHIBASE, but this complexity can be reduced by designing proper algorithm.

1.3 Objective

HIBASE compression technique achieves query performance by sacrificing the storage requirement by using equal length codeword. The objectives of the research are to:

- ❖ develop a dictionary by applying the principle of Huffman coding,
- ❖ further compress the relational storage of HIBASE by applying dynamic Huffman coding,
- ❖ develop algorithm to perform query operation on the compressed storage,
- ❖ and, analyze the performance of the proposed system in terms of both storage and queries.

1.4 Research Approach and Methodology

Further compression using Huffman coding reduces each field to just sufficient bits to encode all the values that occur within the domain of that field. Experimental design has been carried out using following steps:

Step 1: The database has been sorted according to the number of occurrences of same values and the sorted database has been used in Huffman algorithm to generate the dictionary. In this dictionary the codeword with number of occurrences has been stored according to particular keyword.

Step 2: A compression algorithm has been developed to compress the database using Huffman dictionary.

Step 3: Algorithm has been developed to process all kinds of SQL queries using the compressed database only. The result has been decompressed using the Huffman dictionary. Analysis of the algorithm has been given.

Step 4: Synthetic and real datasets has been used to analyze the performance of the system. The storage and performance of the proposed system has been compared with the existing HIBASE and DHIBASE systems.

1.5 Organization of the thesis

In chapter 2, a survey of the research in compression methods and query processing in database systems is presented. We have developed a new system which is an extension of HIBASE architecture. The dictionary organization and compressed relational structure of HIBASE are discussed.

Chapter 3 presents the overview of our proposed architecture, Enhancement of HIBASE Using HUFFMAN Coding (H-HIBASE). The structure stores database in column wise format so that the unnecessary columns need not to be accessed during query processing and also restructuring the database schema will be easy. Each attribute is associated with a domain dictionary. Attributes of multiple relations with same domain share the same dictionary. The detailed analysis of query processing of H-HIBASE system has also been given in this chapter. SQL-like query operators in compressed format have been defined and algorithm of each operator and analysis of the algorithms has been provided as well.

Chapter 4 describes the experimental work that has been carried out. Results obtained have been thoroughly discussed.

Chapter 5 presents conclusions and suggestions for future work.

Chapter 2
Literature Survey

The amount of information does not strictly depend on the volume of data. Insight depends on information; the volume of data depends on its own representation. For cost and performance reason the data should be made as concise as possible. Over the last decade computer memory cost has been significantly reduced. But at the same time, storage size of data and information has also been increased. Therefore, storage cost for large-scale databases is still a great problem [5].

Combining compression with data processing provides performance improvement. Database systems need to provide efficient addressability for data, and generally must provide dynamic update. It is difficult to incorporate these features with good compression techniques [21]. Many research works [3, 4, 5, 6] have been done in database systems to exploit the benefit of compression in storage reduction and performance improvement.

2.1 Compression Techniques

Based on the ability of the compressed data to be decompressed into the original data, data compression techniques can be classified as either loss-less or lossy. A loss-less technique means that the compressed data can be decompressed into the original without any loss of information. On the other side, a compression method that cannot reconstruct the original data from the compressed form is called lossy compression. This type of compression is appropriate for compressing image, voice or video data.

Based on how the input data is treated during compression, we can categorize the compression techniques as lightweight or heavyweight scheme. Lightweight scheme compresses a sequence of values. Heavyweight scheme compresses a sequence of bytes. This scheme is based on patterns found in the data, ignoring the boundaries between values, and treating the input data as an array of bytes.

2.1.1 Loss-Less Compression Methods

The loss-less property is essential for many types of application e.g., word-processing and database applications. The loss-less compression methods can be further classified as follows:

- Statistical encoding
- Dictionary Based Methods

Statistical Encoding

Statistical encoding uses the probabilities of occurrence of each character and each group of characters, assigns short codes to frequently occurring characters or groups of characters while assigns longer codes to less frequently encountered characters or groups of characters [22]. The widely used statistical compression methods are Huffman [23] and Shannon-Fano [24,25] encoding. These methods are static and require a prior knowledge of the probability of occurrence of each character in the input string. Performance degrades if the frequency of occurrences changes. Static methods require at least two passes: one pass to determine the probability of occurrences of the input alphabet and the other pass to encode the string. To maintain the efficiency of the resulting code obtained by compressing data, adaptive or dynamic compression schemes have been developed by many researchers [26, 27].

Dictionary Based Methods

In dictionary based compression methods, the encoder operates on-line, inferring its dictionary of available phrases from previous parts of the message and adjusting its dictionary after the transmission of each phrase. This allows the dictionary to be transmitted implicitly, since the decoder simultaneously makes similar adjustment to its dictionary after receiving each phrase. The Lempel-Ziv families of compression methods [28, 29, 30] are of this type and are used in many file storage and archiving systems. These methods perform better than the character-based methods in terms of speed and space. The main drawback of these methods for database applications is the locality of reference. The encoded data using the initial part of the dictionary is not the same as the encoded data using the later part of the dictionary. Therefore the compressed data is not directly addressable in these methods [11]. These techniques require decompression all, or a large amount of the data even if only a small part of that data is required. Alternatively, a complete dictionary is created in advance using the full message. The dictionary is included explicitly as part of the compressed message. This scheme is highly efficient for decompression, and the compressed data can be searched directly [31].

2.1.2 Lossy Compression Methods

All real world measurement of audio-visual data inherently contains a certain amount of noise. If the compression method includes a small amount of additional noise, no harm is done. Compression techniques that result in this sort of degradation are called lossy. This phenomenon is important because lossy compression techniques can give greater compression ration over the loss-less methods. The higher the compression ratio, the more noise added to the data. Lossy compression is advantageous for image, voice and video data because the additional noise has little effect on the user's perception. JPEG (Joint Photographic Expert Group) and MPEG (Moving Picture Expert Group) are standards for compression of image, voice and video data using lossy compression methods.

2.1.3 Lightweight Compression Methods

Lightweight compression techniques work on the basis of some relationship between values, such as when a particular value occurs often, or if we encounter long runs of repeated values. Run length encoding [32], delta encoding and dictionary encoding [5, 6, 33] are some examples of lightweight compression techniques. In a light-weight compression technique, the compression algorithm is simple and fast. The compression and decompression time is more important than the amount of compression.

2.1.4 Heavyweight Compression Methods

LZO (Lempel Ziv Oberhummer) [34] is a modification of the original Lempel Ziv [28] dictionary coding algorithm. [28] works by replacing byte patterns with tokens. Each time the algorithm recognizes a new pattern, it outputs the pattern and then it adds it to a dictionary. The next time it encounters that pattern, it outputs this token from the table. The first 256 tokens are assigned to possible values of a single byte. Subsequent tokens are assigned to larger patterns.

Details on the particular algorithm modifications added by LZO are undocumented, although the LZO code is highly optimized and hard to decipher. LZO is heavily optimized for decompression speed. It provides the following features:

- Decomposition is simple and very fast.
- Requires no memory for decomposition.
- Compression is fast.
- The algorithm is thread safe.
- The algorithm is loss-less.

2.2 Compression on Database Processing

Compression has now become an essential part of many large information systems where large amount of data is processed, stored or transferred. This data may be of any type e.g., voice, video, text, tables etc. No single compression technique is suitable for all types of data. Lossy compression is appropriate for voice or video data where as loss-less compression is suitable for other data types. Cormack [3] has used a modified Huffman code [23] for the IBM IMS database system. Westmann et al. [4] has developed a lightweight compression method based on LZW [30] for relational databases. Moffat et al. [35] use the Run Length Encoding [32] method for a parameterized compression technique for sparse bitmaps of a digital library.

2.2.1 Compression of Relational Structures

We shall certainly get some benefits if we compress relational databases. We can improve the index structures such as B-trees by reducing the number of leaf pages. We can reduce the storage requirements for information. We have reduction in transaction turnaround time and user response time as a result of faster transfer between disk and main memory in I/O bound systems. In addition, since this will also reduce I/O channel loading, the CPU can process many more I/O requests and thus increase channel utilization. We may have better efficiency of backup since copies of the database could be kept in compressed form. This reduces the number of tapes required to store the data and reduces the time of reading from, and writing to, these tapes. The whole or the major portion of processing data in compressed form may be memory resident. Main memory access time is several orders of magnitude faster than the secondary storage access time. This improves performance.

Compression can be applied to databases at the relation level, page level and the tuple or attribute level. In page level compression the database is represented as a set of compressed tuples. An individual tuple can be compressed and decompressed within a page. When a particular tuple is required, the corresponding page is transferred to the memory and decompression is necessary only if the decompressed tuple is required. An approach to page level compression of relations and indexes is given in [8]. The important aspects of the technique are that each compressed data page is independent of the other pages and each tuple can be decompressed based on the information found on that specific page. A compressed tuple can be referred by a page-no and an offset. The degree of compression greatly depends on the range of values in each field for the set of tuples stored on a page.

Wee et al. [36] has proposed a tuple level compression scheme using Augmented Vector Quantization. Vector quantization is a lossy data compression technique used in image and speech coding [37]. However Wee et al. [36] has developed a loss-less method for database compression to improve performance of I/O intensive operations.

A similar compression scheme has been given in [4] with a different approach. The work presented a set of very simple and light-weight compression techniques and shows how a database system can be extended to exploit these compression techniques. Numeric compression is done by suppressing zeros; string compression is done by classical Huffman [23] or LZW [30]. Dictionary-based compression methods, however, are used for any field containing a small number of different values.

2.2.2 HIBASE Architecture

The HIBASE [11] approach is a more radical attempt to model the data representation that is supported by information theory. The architecture represents a relation table in storage as a set of columns, not a set of rows. Of course, the user is free to regard the table as a set of rows. However, the operation of the database can be made considerably more efficient when the storage allocation is by columns.

Table 2.1: Distributor relation

ID	First Name	Last Name	Area
1	Abdul	Bari	Dhaka
2	Abdur	Rahman	Sylhet
3	Md	Alamin	Chittagong
4	Abdul	Gafur	Dhaka
5	Salam	Bari	Sylhet
6	Md	Tuhin	Rajshahi
7	Salam	Mia	Rajshahi
8	Chan	Mia	Dhaka
9	Ghendhu	Mia	Chittagong
10	Abdur	Rahman	Sylhet

The database is a set of relations. A relation is a set of tuples. A tuple in a relation represents a relationship among a set of values. The corresponding values of each tuple belong to a domain for which there is a set of permitted values. If the domains are D_1, D_2, ……., D_n respectively. A relation r is defined as a subset of the Cartesian product of the domains. Thus r is defined as $r \subseteq D_1 \times D_2 \times ....\times D_n$.

An example of a relation is given in Table 2.1 In the conventional database technology, we have to allocate enough space to fit the largest value of each field of the records. When the database designer does not know exactly how large the individual values are, he/she must err on the side of caution and make the field larger than is strictly necessary. In this instance, a designer should specify the width in bytes as shown in Table 2.2 Each tuple is occupying 18 bytes, so that 10 tuples occupy 180 bytes.

Table 2.2: Field length and tuple size for Distributor relation

Attribute No	**Attribute Name**	**Bytes**
0	First Name	6
1	Last Name	6
2	Area	6
Total		**18**

The HIBASE architecture by Cockshot, McGregor and Wilson [11, 38] is a more radical approach to model the data representation. The HIBASE architecture can be derived from a conventional record structure using the following steps:

1. A dictionary per domain is employed to store the string values and to provide integer identifiers for them. This achieves a lower range of identifier, and hence a more compact representation than could be achieved if a single dictionary was provided for the whole database.
2. Replace the original field value of the relation by identifiers. The range of the identifiers is sufficient to distinguish string of the domain dictionary.

In the relational approach, the database is a set of relations [39]. A relation represents a set of tuples. In a table structure, the rows represent tuples and the columns contain values drawn from domains. Queries are answered by the application of the operations of the relational algebra, usually as embodied in a relational calculus-based language such as SQL.

Table 2.3: Compressed table in HIBASE

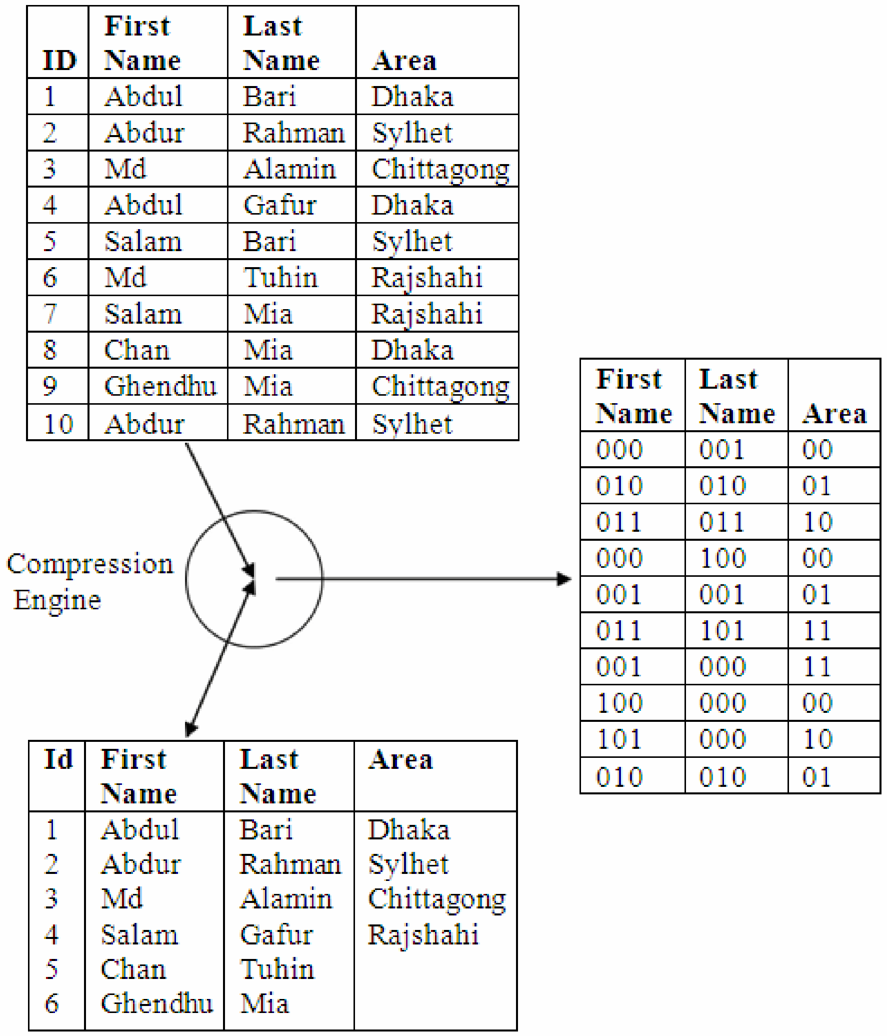
First Name	**Last Name**	**Area**
000	001	00
010	010	01
011	011	10
000	100	00
001	001	01
011	101	11
001	000	11
100	000	00
101	000	10
010	010	01

The objective of the compression architecture is to trade off cost and performance between that of conventional DBMS and main memory DBMS. Costs should be less than the second, and processes faster than the first [11]. The architecture's compact representation can be derived from a traditional record structure in the following steps:

Creating dictionaries: A dictionary for each domain is created which stores string values and provides integer identifiers for them. This achieves a lower range of identifiers, and hence a more compact representation than could be achieved if only a single dictionary was generated for the entire database.

Replacing field values by integer identifiers: The range of the identifiers need only be sufficient to unambiguously distinguish which string of the domain dictionary is indicated. In Fig. 2.1, since there are only 6 distinct First Names, only six identifiers are required. This range can be represented by only a 3-bit binary number.

Therefore in the compressed table each tuple requires only (3 bits: First Name, 3 bits: Last Name, 2 bits: Area) a total of 8 bits instead of the 18 bytes (6 bytes: First Name, 6 bytes: Last Name, 6 bytes: Area) for the uncompressed relation. This achieves a compression of the table by a factor of over 22. The actual compression ratio is somewhat lower due to the space requirements of domain dictionaries. Generally some domains are present in several relations and this reduces the dictionary overhead by sharing them among different attributes. In a domain a specific identifier always refers to the same field value and this fact enables some operations to be carried out directly on compressed table data without examining dictionary entries until string values are essential (e.g. for output). This is not the overall storage; however, we must take account of the space occupied by the domain dictionaries and indexes. Typically, a proportion of domain is present in several relations and this reduces the dictionary overhead by sharing it by different attributes.



ID	First Name	Last Name	Area
1	Abdul	Bari	Dhaka
2	Abdur	Rahman	Sylhet
3	Md	Alamin	Chittagong
4	Abdul	Gafur	Dhaka
5	Salam	Bari	Sylhet
6	Md	Tuhin	Rajshahi
7	Salam	Mia	Rajshahi
8	Chan	Mia	Dhaka
9	Ghendhu	Mia	Chittagong
10	Abdur	Rahman	Sylhet

First Name	Last Name	Area
000	001	00
010	010	01
011	011	10
000	100	00
001	001	01
011	101	11
001	000	11
100	000	00
101	000	10
010	010	01

Id	First Name	Last Name	Area
1	Abdul	Bari	Dhaka
2	Abdur	Rahman	Sylhet
3	Md	Alamin	Chittagong
4	Salam	Gafur	Rajshahi
5	Chan	Tuhin	
6	Ghendhu	Mia	

Fig. 2.1: Compression of Distributor relation

Dictionary structure: All distinct attribute values (lexemes) are stored in an end-to-end format in a string heap. A hashing mechanism is used to achieve a contiguous integer identifier for the lexemes. This reduces the size of the compressed table. It has three important characteristics:

1. It maps the attribute values to their encoded representation during the compression operation: *encode*(*lexeme*) →*token*
2. It performs the reverse mapping from codes to literal values when parts of the relation are decompressed: *decode*(*token*) → *lexeme*.
3. The mapping is cyclic such that *lexeme* = *decode(encode(lexeme))* and also *token* = *encode(decode(token))*.

The structure is attractive for low cardinality data. For high cardinality and primary key data, the size of the string heap grows considerably and contributes very little or no compression.

Column-wise storage of relations: The architecture stores a table as a set of columns (Fig. 2.2), not as a set of rows. This makes some operations on the compressed database considerably more efficient. A column-wise organization is much more efficient for dynamic update of the compressed representation. A general database system must support dynamic incremental update, while maintaining efficiency of access. The processing speed of a query is enhanced because queries specify operations only on a subset of domains. In a column-wise database only the specified values need to be transferred, stored and processed. This requires only a fraction of the data that required during processing by rows.

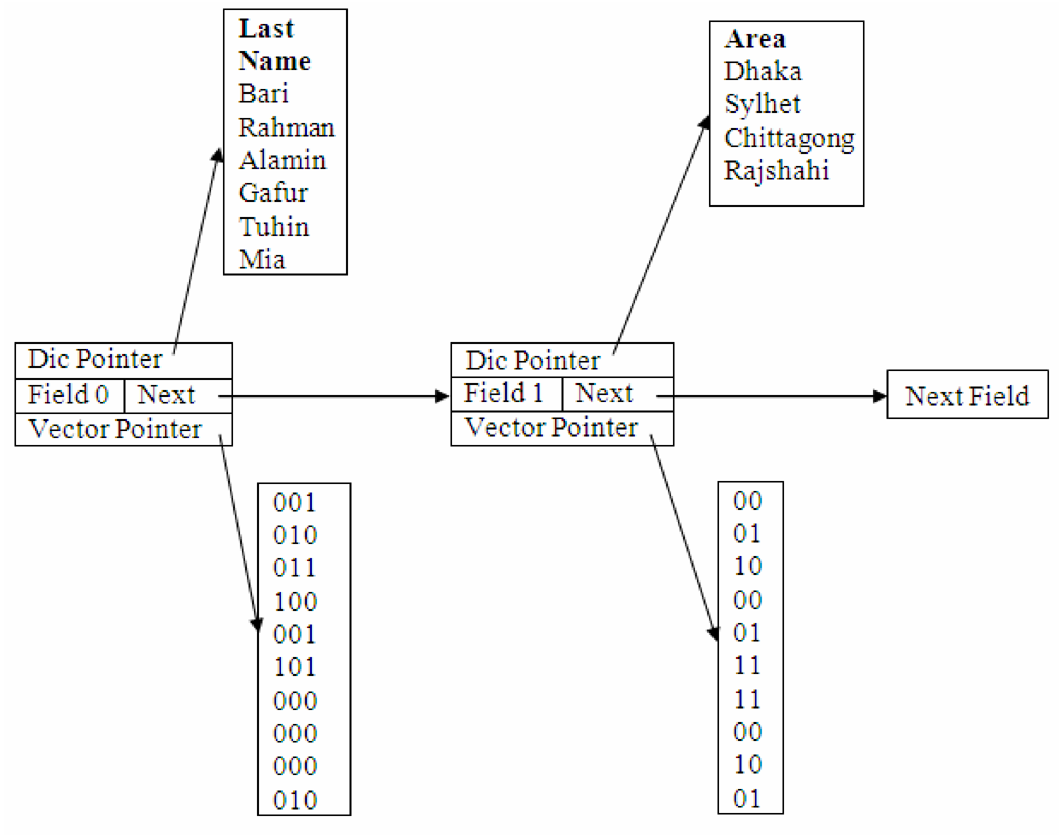


Fig. 2.2: Column-wise storage of a relation

2.2.3 The DHIBASE Technique

Disk-based system (DHIBASE) is an extension of HIBASE architecture [15]. The structure stores database in column wise format so that the unnecessary columns need not to be accessed during query processing and also restructuring the database schema will be easy. Each attribute is associated with a domain dictionary. Attributes of multiple relations of same domain share a single dictionary. DHIBASE have presented a sorting mechanisms according to sorting the compressed database for both string and code order. DHIBASE perform high performance SQL queries on single or multiple tables in compressed form. It has been designed and implemented all basic relational algebra operations e.g. *selection, projection, join* operation, *set* operations, *aggregation*, *insertion, deletion* and *update* on the architecture [15].

2.2.4 The HUFFMAN Technique

In computer science and information theory, Huffman coding is an entropy encoding algorithm used for lossless data compression. The term refers to the use of a variable-length code table for encoding a source symbol (such as a character in a file) where the variable-length code table has been derived in a particular way based on the estimated probability of occurrence for each possible value of the source symbol. It was developed by David A. Huffman while he was a Ph.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes"[18, 40].

Huffman coding is widely used and very effective technique for compressing data;

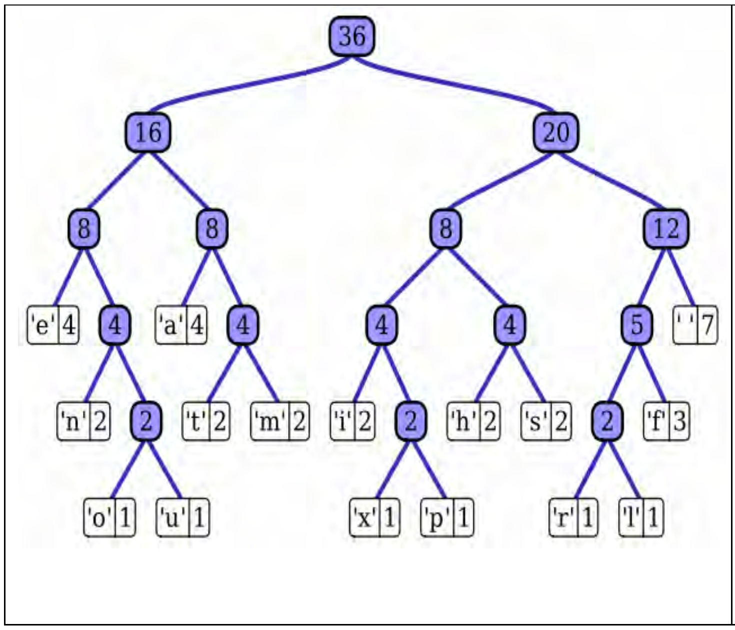
1. Savings of 20% to 90% are typical, depending on the characteristics of the file being compressed.
2. Huffman coding involves the use of variable-length codes to compress long string of text.
3. By assigning shorter codes to more frequent characters, Huffman encoding can compression text by as much as 80%.

The simplest construction algorithm uses a priority queue where the node with lowest probability is given highest priority [40]:

1. Create a leaf node for each symbol and add it to the priority queue.
2. While there is more than one node in the queue:
    a. Remove the two nodes of highest priority (lowest probability) from the queue
    b. Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
    c. Add the new node to the queue.
3. The remaining node is the root node and the tree is complete.

Since efficient priority queue data structures require O(log *n*) time per insertion, and a tree with *n* leaves has 2*n*–1 nodes, this algorithm operates in O(*n* log *n*) time.

Huffman tree generated from the exact frequencies of the text "this is an example of a Huffman tree". The frequencies and codes of each character are shown in Fig. 2.3. Encoding the sentence with this code requires 135 bits, as opposed to 288 bits if 36 characters of 8 bits were used.



Char	Freq	Code
space	7	111
a	4	010
e	4	000
f	3	1101
h	2	1010
i	2	1000
m	2	0111
n	2	0010
s	2	1011
t	2	0110
l	1	11001
o	1	00110
p	1	10011
r	1	11000
u	1	00111
x	1	10010

Fig. 2.3: Construction of a HUFFMAN Tree.

2.2.5 Arithmetic Coding

In Huffman coding, each character is encoded into an integral number of bits. This means that the codes may often be longer than that strictly required for the character. For example a character with probability of occurrence 0.9 can be coded minimally in 0.135 bits (from information-theoretic considerations), but requires 1 full bit in this scheme.

Arithmetic coding attempts to address the above shortcoming of Huffman coding. Here, the compressed version of the input data is represented by the interval between two real numbers of arbitrary precision, (x, y), where $0<=x< y<=1$. At the start the range is initialized to the entire interval [0,1), and this range is progressively refined. During the encoding process, each character is assigned an interval within the current range, the width of the interval being proportional to the probability of occurrence of that character. The range is then narrowed to that portion of the current range which is allocated to this character. So, as encoding proceeds and more data is scanned, the interval needed to represent the data becomes smaller and smaller, and the number of bits needed to specify the interval grows. The more likely characters reduce the range less than the unlikely characters and hence add fewer bits to the compressed data. The implementation details of this scheme are given in [41, 42].

Arithmetic coding also has adaptive and non-adaptive versions, in exactly the same manner as that described previously for Huffman coding.

2.2.6 Three Layer Model

The Three Layer Model was developed by Hoque et al. [12]. This database architecture was designed for storage and querying of structured relational databases, sparsely populated e-commerce data and semi-structured XML. They have proved the system in practice with a variety of data. They have achieved significant improvement over the basic HIBASE model [11] for relational data. Their system performs better than the Ternary model [43] for the sparsely populated data. They compared their results with UNIX utility *compress.* The system performs a factor of two to six more in reduction of data than *compress,* maintaining the direct addressability of the compressed form of data.

The architecture has three layers:

Layer 1: The lowest layer is the vector structures to store the compressed form of data. As queries are processed on the compressed form of data, indexing is allowed on the structure such that we can access any element in the compressed form without decompression. The size of the element can vary during database update. The vector can adapt dynamically as data is added incrementally to the database. This dynamic vector structure is the basic building block of the architecture.

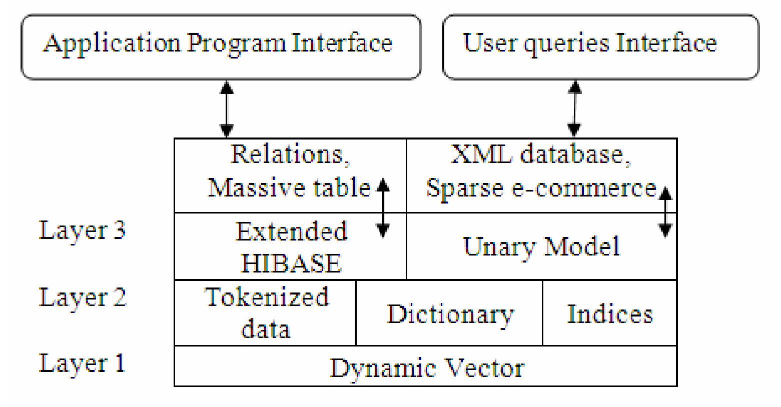


Fig. 2.4: The Three Layer Model

Layer 2: The second layer is the explicit representation of the off-line dictionary in compact form. They have presented a phrase selection algorithm for off-line dictionary method in linear time and space [44].

Layer 3: The third layer consists of the data models to represent structured relational data, sparsely populated data and semi-structured XML.

2.2.7 Columnar Multi Block Vector Structure (CMBVS)

The proposed compression based data management system architecture [2] can be used to handle terabyte level of relational data. The existing compression schemes e.g. Hibase [11] or Three Layer Database Compression Architecture [12] work well for memory resident data and provide good performance. These are low cost solution for highperformance data management system but are not scalable to manage terabyte level of data. CMBVS is a disk based columnar multi-block vector structure that can be used to store relational data in a compressed representation with direct addressability. Parallel data access can be achieved by distributing the vector structure into multiple servers to improve the scalability. The structure is capable of carrying out query directly on the compressed data. This reduces query time drastically. The system has been compared with the conventional relational DBMS. The architecture is significantly efficient in storage reduction and also faster than conventional systems in retrieval time performance.

2.2.8 Compression in Oracle

The Oracle RDBMS has introduced a compression technique [9] for reducing the size of relational tables. This compression algorithm is specifically designed for relational data. Using this compression technique, Oracle is able to compress data much more effectively than standard compression techniques. More significantly, Oracle incurs virtually no performance penalty for SQL queries accessing compressed tables. In fact, Oracle's compression may provide performance gains for queries accessing large amounts of data, as well as for certain data management operations like backup and recovery.

The compression algorithm used in Oracle compresses data by eliminating duplicate values in a database block. The algorithm is a loss-less dictionary-based compression technique. One dictionary (symbol table) is created for each database block. Therefore, compressed data stored in a database block is self-contained. That is, all the information is available within the block to recreate the uncompressed data in that block. This compression technique has been chosen to achieve local optimality of compression ratio. The algorithm is greedy, meaning that it tries to load as many rows as possible into each block. It does not attempt to achieve any form of global compression ratio optimality.

The problem of global compression ratio optimality is highly computationally intensive. If global compression ratio optimality is desired, the entire set of rows to be compressed needs to be buffered before blocks can be populated. For large data warehouses this is not feasible because it would potentially require to buffer terabytes of data, which is not practical.

2.3 Query Processing

Query processing refers to the range of activities involved in extracting data from a database. The basic steps involved in query processing are

1. Parsing and translation
2. Optimization
3. Evaluation

The first action the system must take in query processing is to translate a given query into its internal form. Given a query, there are generally a variety of methods for computing the answer. In SQL, a query could be expressed in several different ways. Each SQL query can itself be translated into one relational-algebra expression in one of several ways. We can execute each relational-algebra operation by many different algorithms. To specify fully how to evaluate a query, we need not only to provide the relational-algebra expression, but also to annotate it with instructions specifying how to evaluate each operation. Annotation may state the algorithm to be used for a specific operation, or the particular index or indices to use. Different query-evaluation plans for a given query have different costs. Based on these cost estimates, a particular plan is accepted. The given query is evaluated with that plan and the result of the query is output.

2.3.1 Uncompressed Query Processing

Parsing of query languages differs little from parsing of traditional programming languages. Main parsing techniques were covered in [45], but here optimization is presented from a programming language point of view. A excellent description of external sorting algorithms, including an optimization that create initial runs that are (on the average) twice the size of the memory, is described in [46].

Query optimization: Much work has been done in query optimization. Access-path selection in the System R optimizer is described in [47], which was one of earliest relational query optimizers. Volcano, an equivalence-rule based query optimizer, is described in [48]. Query processing in Starburst is described in [49]. Query optimization in Oracle is briefly outlined in [50].

The SQL language poses several challenges for query optimization, including the presence of duplicates and nulls, and the semantics of nested sub-queries. Extension of relational algebra to duplicates is described in [51]. Optimization of nested sub-queries is discussed in [52].

Multi-query optimization, which is the problem of optimizing the execution of several queries as group, is described in [53]. If an entire group of queries is considered, it is possible to discover common sub-expressions that can be evaluated once for the entire group. Optimization of a group of queries and the use of common sub-expressions are considered in [54]. Optimization issues in pipelining with limited buffer space combined with sharing of common sub-expression are discussed in [55].

Join operation: In the mid 1970s, database systems used only nested-loop join and merge join. These systems, which were related to the development of System R, determined that either the nested-loop join or merge join nearly always provided the optimal join method [56]; hence, these two were the only join algorithms implemented in System R. The System R study did not include an analysis of hash join algorithms. Today hash join algorithms are considered to be highly efficient.

Hash join algorithms were initially developed for parallel database systems. Hash join techniques are described in [57], and extensions including hybrid hash join are described in [58]. Hash join techniques that can adapt to the available memory is important in systems where multiple queries may be running at the same time. This issue is described in [59]. The use of hash joins and hash teams, which allow pipelining of hash joins by using the same partitioning for all hash joins in a pipeline sequence in the Microsoft SQL Server is presented in [60].

Aggregation: An early work on relational algebra expressions with aggregate functions is found in [61]. More recent work in this area includes [62]. Optimization of queries containing outer joins is described in [63].

Views: A survey of materialized view maintenance is presented in [64]. Optimization of materialized view maintenance plans is described in [65]. Query optimization in the presence of materialized views is addressed in [66].

2.3.2 Compressed Query Processing

Very few systems execute queries directly on compressed data without any decompression. HIBASE Architecture [11], Three Layer Model [12], Columnar Multi Block Vector Structure (CMBVS) [2] are the systems that execute queries directly on compressed data (Fig. 2.5). The query is translated to compressed form and then processed directly against the compressed relational data. Less data needs to be manipulated and this is more efficient than the conventional alternative of processing an uncompressed query against uncompressed data.

The final answer will be converted to a normal uncompressed form. However, the computational cost of this decompression is low because the amount of data to be decompressed is only a small fraction of the processed data. All these systems are capable of executing queries on single compressed relation. Queries on multiple compressed relations have not been designed so far.

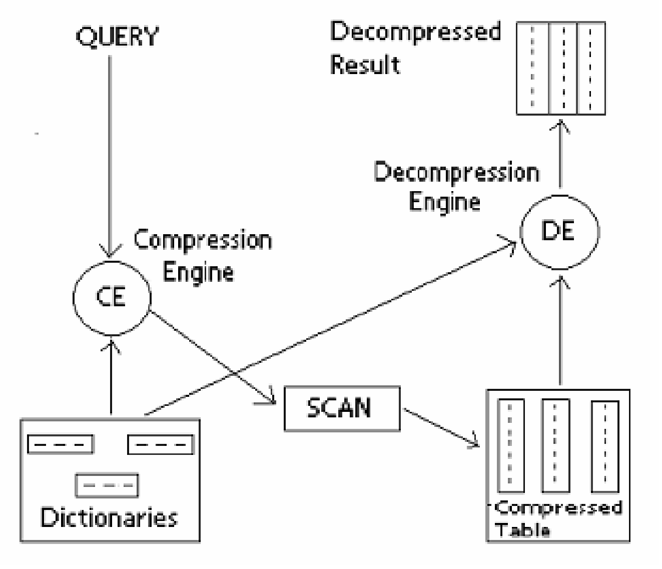


Fig. 2.5: Querying a database in compressed form

Compression technique used in Oracle is different than that used in Hibase Architecture [11], Three Layer Model [12] and CMBVS [2]. As separate symbol table is created for each database block, the compressed data is not directly addressable in compressed form.

Therefore, it is hard to implement queries directly on multiple compressed relations. In fact, Oracle's compression algorithm is particularly well suited for data warehouses environment, which contains large volumes of historical data with heavy query workloads. The system is targeted mostly for read-only applications where simple queries are involved.

2.4 Summary

This chapter described different types of existing compression techniques, compression of relational database, development in query processing both in uncompressed and compressed form. We have thoroughly discussed the HIBASE architecture and HUFFMAN architecture because we taken these model as the basis of our architecture (which we call H-HIBASE). But there are fundamental differences between HIBASE and H-HIBASE. Differences are observable in storage structure of compressed data, number and types of query processing.

Chapter 3
H-HIBASE Architecture

This chapter describes the details of the proposed H-HIBASE architecture for storage of compressed relational data. The chapter also illustrates the details of the query processing techniques of the proposed system. We have developed the system for single processor system. Query is evaluated directly on compressed data.

3.1 H-HIBASE: Enhancement of HIBASE Model Using HUFFMAN Coding

The basic HIBASE architecture is memory based. We have developed a more general architecture (Fig. 3.1) that supports both memory and disk based operations. We have made two assumptions:

a. The architecture stores relational database only
b. Single processor system architecture

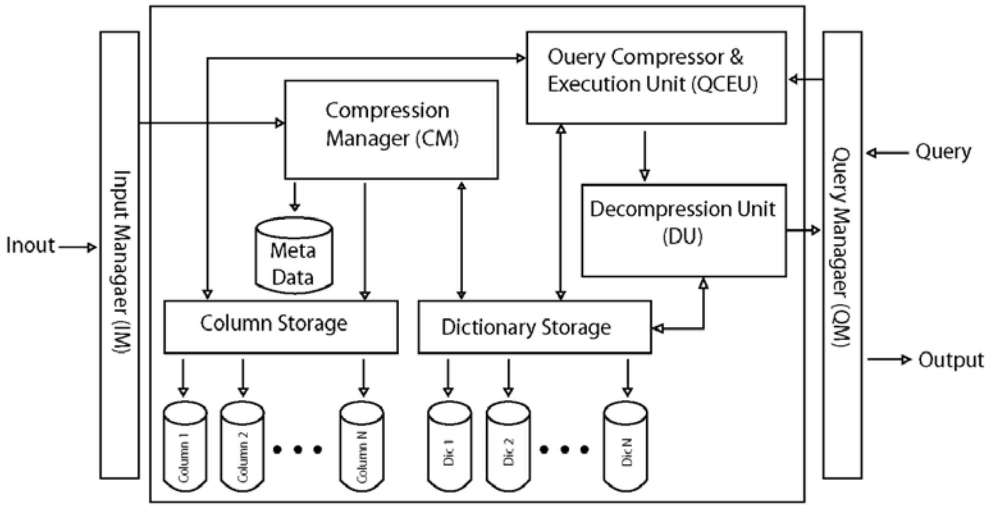


Fig. 3.1: H-HIBASE Architecture

Although the architecture is designed for single processor system, it can easily be expanded for other architectures. The Input Manager (IM) takes input from different sources and passes to Compression Manager (CM). CM compresses the input, make necessary update to appropriate dictionaries and stores the compressed data into respective column storage.

Query Manager (QM) takes user query and passes to Query Compression and Execution Unit (QCEU). QCEU translates the query into compressed form and then applies it against compressed data. Then it passes the compressed result to Decompression Unit (DU) that converts the result into uncompressed form.

Each compressed column is stored across multiple disk blocks. Each disk block has fixed size. Compressed data are stored in end-to-end position in disk block. No data is split over two disk blocks. The total database is kept into the main memory when the database is small enough to be placed into the memory. For large databases, recently active parts are placed into the main memory. The last disk block of each column and each dictionary is always kept into main memory.

All insertions are committed in this memory block. The Insert or Update operation in H-HIBASE model requires ’string’ look up in the dictionary. Efficient decoding [9] from code to ‘string’ may be achieved by two ‘table look-up’ operations. So we do not need to search the entire dictionary in the worst case. If the ’string’ is present in the dictionary the operation does not need a reorganization of the vector structure. If the ’string’ is not present in the dictionary it is inserted into the dictionary. This insertion might result an increase of the element width. In this case the operation requires a reorganization of the vector structure. Deletion is performed by replacing the desired record with the last record and then reducing number of records by one. Dictionary entries are not deleted.

3.2 H-HIBASE: Analysis

As we know Huffman algorithm generates an optimal tree, hence the compression has been optimized. Moreover high performance has been ensured as most repeated attribute values get more weight and has been entered first in the dictionary i.e. domain dictionary values has been sorted in such a way that frequently occurred values has been accessed first then the rare values. Fig. 3.2 shows the whole analysis at a glance. It has been shown in Fig. 3.2 that five steps are necessary to complete whole process. Steps are explained below:

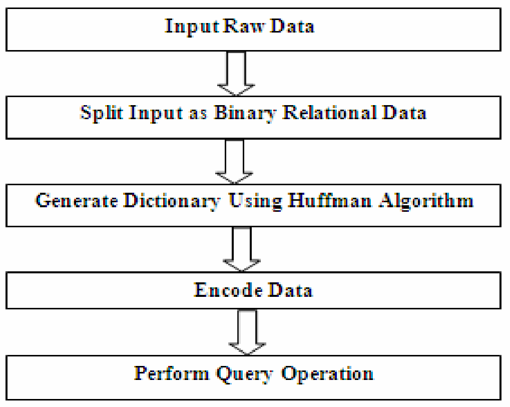

Input Raw Data
Split Input as Binary Relational Data
Generate Dictionary Using Huffman Algorithm
Encode Data
Perform Query Operation

Fig. 3.2: Experimental Design at a glance.

Step 1: Take synthetic and real data as input (Consider the database shown in table 3.1):

Table 3.1: Distributor Relation

ID	First Name	Last Name	Area
1	Abdul	Bari	Dhaka
2	Abdur	Rahman	Sylhet
3	Md	Alamin	Chittagong
4	Abdul	Gafur	Dhaka
5	Salam	Bari	Sylhet
6	Md	Tuhin	Rajshahi
7	Salam	Mia	Rajshahi
8	Chan	Mia	Dhaka
9	Ghendhu	Mia	Chittagong
10	Abdur	Rahman	Sylhet

The H-HIBASE approach is a more radical attempt to model the data representation that is supported by information theory. The architecture represents a relation table in storage as a set of columns, not a set of rows. Of course, the user is free to regard the table as a set of rows. However, the operation of the database can be made considerably more efficient when the storage allocation is by columns.

Step 2: Split the relational database as binary relational databases (Shown in table 3.2):

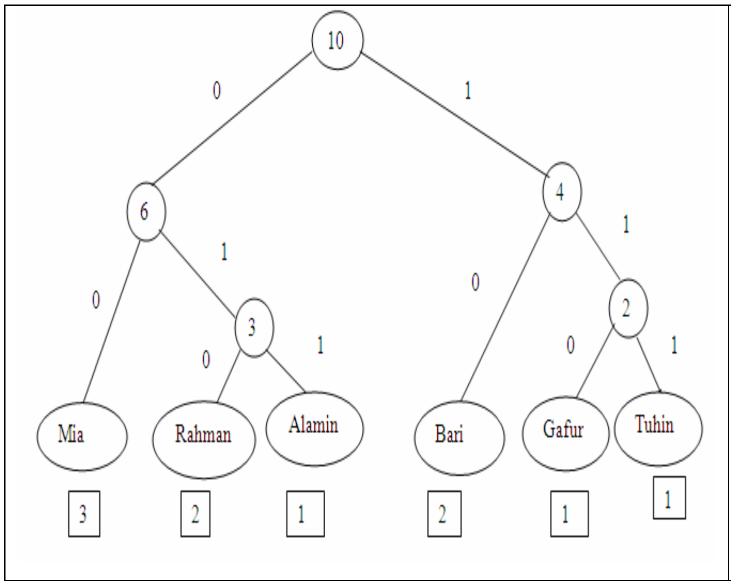
Table 3.2: Binary Relational Database

ID	First Name	ID	Last Name	ID	Area
1	Abdul	1	Bari	1	Dhaka
2	Abdur	2	Rahman	2	Sylhet
3	Md	3	Alamin	3	Chittagong
4	Abdul	4	Gafur	4	Dhaka
5	Salam	5	Bari	5	Sylhet
6	Md	6	Tuhin	6	Rajshahi
7	Salam	7	Mia	7	Rajshahi
8	Chan	8	Mia	8	Dhaka
9	Ghendhu	9	Mia	9	Chittagong
10	Abdur	10	Rahman	10	Sylhet

Binary relational database is a database with two columns in each table and it is very efficient where column wise searching is regular. The Table 3.1 has been split to three Binary relation tables which are shown in Table 3.2.

Step 3: Generate dictionary using HUFFMAN algorithm (Shown in Table 3.3):

The range of the identifiers need only be sufficient to unambiguously distinguish which string of the domain dictionary is indicated. In Fig. 3.3, since there are only 6 distinct Last Names, only six variable length codeword are required. This range can be represented by only a 3-bit or 2 bit binary number.



Name	Freq	Code
Mia	3	00
Rahman	2	010
Bari	2	10
Alamin	1	011
Gafur	1	110
Tuhin	1	111

Fig. 3.3: Construction of Huffman Tree for Last Name column.

A dictionary for each domain has been created which stores string values and provides Huffman codeword for them. This achieves a lower range of codeword, and hence a more compact representation could be achieved.

Table 3.3: H-HIBASE Dictionary

Index	First Name	Codeword
1, 4	Abdul	110
5, 7	Salam	111
2, 10	Abdur	00
3,6	Md	01
8	Chan	100
9	Ghendhu	101

Index	Last Name	Codeword
7,8,9	Mia	10
1,5	Bari	111
2, 10	Rahman	00
3	Alamin	010
4	Gafur	011
6	Tuhin	110

Index	Area	Codeword
1,4,8	Dhaka	10
2,5,10	Sylhet	11
3,9	Chittagong	00
6,7	Rajshahi	01

In Table 3.3 three different dictionaries are shown which has been generated by Huffman principle. From above table it has been shown that most repetitive values give smaller codeword, hence further compression has been achieved. In the above table it has been shown that Mia most repeated string in the LastName column need just two bits (10) in the dictionary. Whereas Tuhin the less repeated string need three bits (110) to represent it in the dictionary.

Step 4: Develop algorithm to encode data (Shown in Table 3.4):

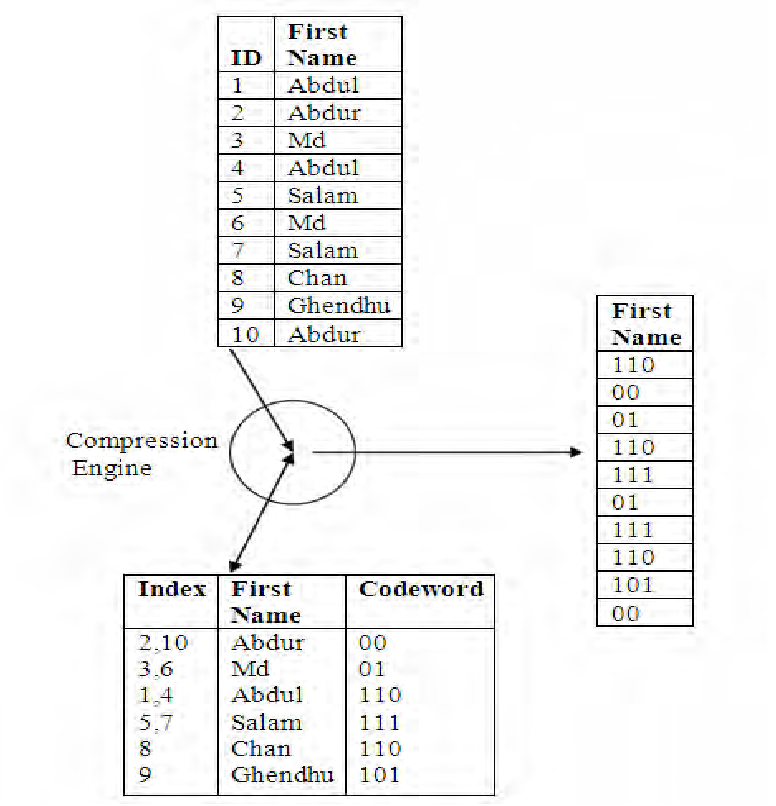
Table 3.4: H-HIBASE Storage

First Name	Last Name	Area
110	111	10
00	00	11
01	010	00
110	011	10
111	111	11
01	110	01
111	10	01
100	10	10
101	10	00
00	00	11

In the compressed table each tuple requires only (First Name required minimum 2 maximum 3 bits, Last Name required minimum 2 maximum 3 bits, Area required 2 bits) a total of maximum 8 bits. This achieves a compression of the table by a factor of over 15. This is not the overall storage; however, we must take account of the space occupied by the domain dictionaries. Typically, a proportion of domain is present in several relations and this reduces the dictionary overhead by sharing it by different attributes.

Step 5: Develop algorithm to perform query operation on the compressed storage:

After encoding data it is challenging to retrieve those codes from compressed storage. Fig. 3.4 shows how compressed data can be accessed. Dictionary access and compressed storage access are necessary to perform every query. From the following figure it has been shown that the searched value has been look in the dictionary first, if it is available than calculate its position and length. According to the position and length it can easily be accessed from the compressed storage.



ID	First Name
1	Abdul
2	Abdur
3	Md
4	Abdul
5	Salam
6	Md
7	Salam
8	Chan
9	Ghendhu
10	Abdur

First Name
110
00
01
110
111
01
111
110
101
00

Index	First Name	Codeword
2,10	Abdur	00
3,6	Md	01
1,4	Abdul	110
5,7	Salam	111
8	Chan	110
9	Ghendhu	101

Fig. 3.4: Compression of Distributor relation

Column-wise storage of relations: The architecture stores a table as a set of columns (Fig. 3.5), not as a set of rows. This makes some operations on the compressed database considerably more efficient. A column-wise organization is much more efficient for dynamic update of the compressed representation. A general database system must support dynamic incremental update, while maintaining efficiency of access. The processing speed of a query is enhanced because queries specify operations only on a subset of domains. In a column-wise database only the specified values need to be transferred, stored and processed. This requires only a fraction of the data that required during processing by rows.

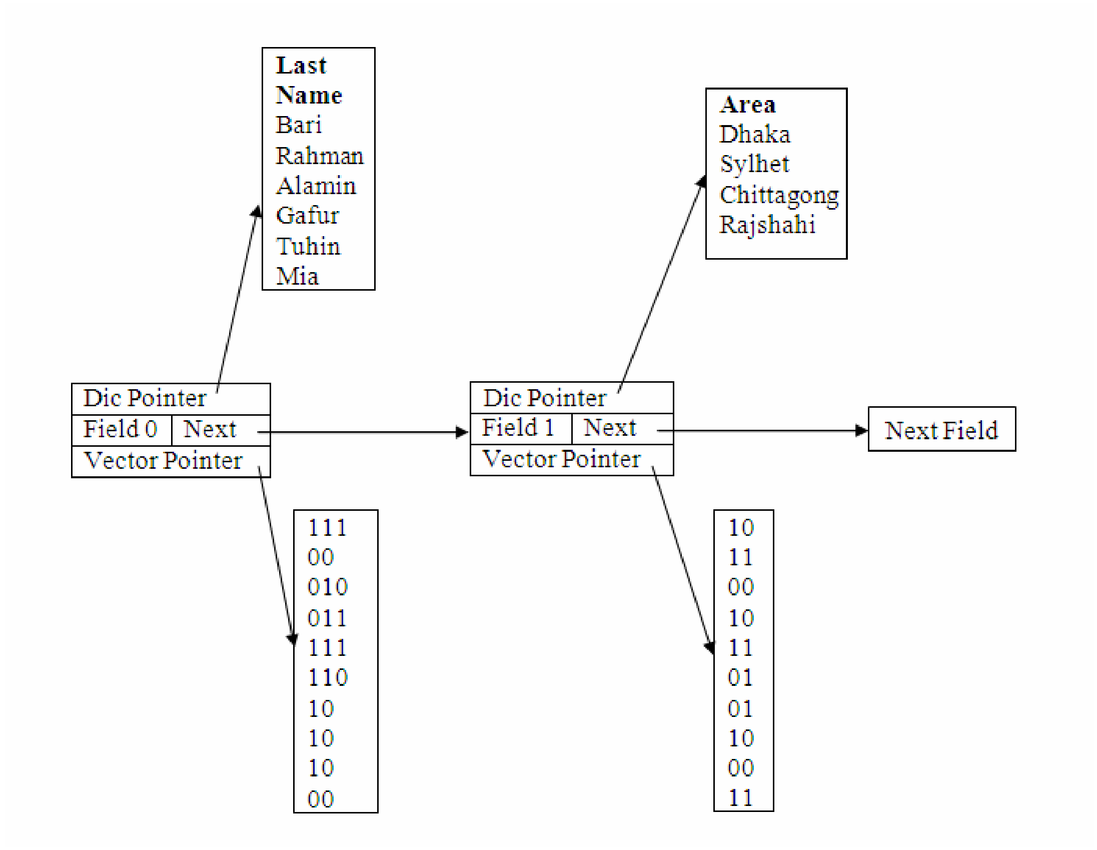


Fig. 3.5: Column-wise storage of a relation

3.3 H-HIBASE: Storage Complexity

HIBASE:

$S_{Ci} = n * C_i$ bits

Where S_{Ci} = space needed to store column i in compressed form n = number of records in the relation

C_i = number of bits needed to represent i^{th} attribute in compressed form

= $\lfloor lg(m) \rceil$; where m is no of entries in the corresponding domain dictionary

Total space to store compressed table, $S_{HIBASE} = \sum_{i=1}^{p} S_{C_i}$ bits; where p is the number of column

If we assume that domain dictionaries occupy an additional 25% of S = 1.25 S, then total space in compressed relation, $S_{CRHIBASE} = 1.25\ S_{HIBASE}$

H-HIBASE:

$S_{H\text{-}HIBASE} = \sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}$ bits

a_{ij} represents the number of bits in a particular position of two dimensional matrix, where i is the number of row and j is the number of column. From equation it has been shown that the first iteration counts all bits within a row and second iteration counts all columns. Hence total bits of entire storage have been counted by the equation.

If we assume that domain dictionaries occupy an additional 25% of S = 1.25 S, then total space to store the compressed relation, $S_{CRH\text{-}HIBASE} = 1.25\ S_{H\text{-}HIBASE}$

Compression Enhancement:

Compression Enhancement = $((S_{CRHIBASE} - S_{CRH\text{-}HIBASE})*100 / S_{CRHIBASE})$ %

3.4 H-HIBASE: Implementation

3.4.1 H-HIBASE Dictionary

To translate to and from the compressed form it is necessary to go through a dictionary. A dictionary is a list of values that occur in the domain. Huffman dictionary is comparable to Huffman table where two pieces of information has been stored namely lexeme and token. Lexeme corresponds to discrete values in a domain whereas token corresponds to code-word. Short code-words have been placed first for a domain dictionary which ensures faster dictionary access. Hence there has been a significant improvement in database performance during compression, decompression and query operations. As Huffman coding gives more weight to most repeated value, it is likely to have shortest code-word to most repeated value. Huffman algorithm have been generated the position of values in the dictionary as well. Table 3.3 shows dictionaries for distributor relationship. The Huffman dictionary has generated as per following algorithm.

Algorithm 3.1: Huffman(C)

```
HUFFMAN (C)
1. n ← |C|
2. Q ← C
3. for i ← 1 to n -1
4.        do allocate a new node z
5.             left[z]  ← x ← EXTRACT-MIN (Q)
6.             right[z] ← y ← EXTRACT-MIN (Q)
7.             f[z] ← f[x] + f[y]
8.             INSERT (Q, z)
9. return EXTRACT-MIN (Q)
```

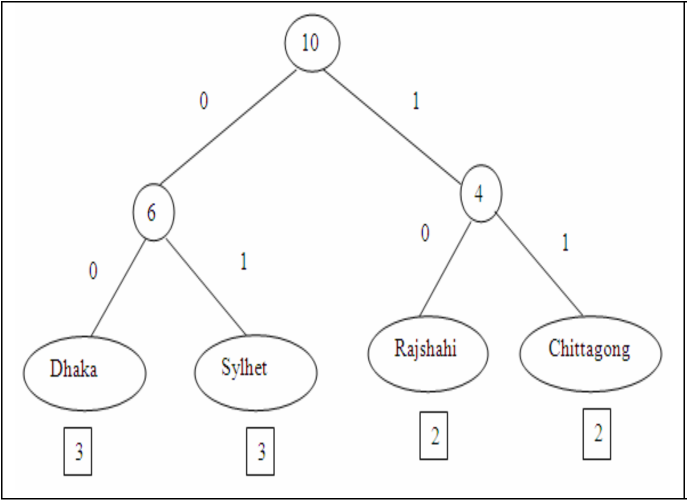
In the pseudocode that follows, we assume that C is a set of n strings and each string c € C is an object with a defined frequency f[c]. The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of |C| leaves and performs a sequence of |C| - 1 “meaning” operations to create the final tree. A min-priority queue Q, keyed on f, is used to identify the two least-frequent objects to merge together. The result of the merger of two objects is a new object whose frequency is the sum of the frequencies of the two objects that were merged [67].

In algorithm 3.1 n is the initial queue size, line 2 initializes the min-priority queue Q with the character in C. The for loop in line 3-8 repeatedly extracts the two nodes x and y of lowest frequency from the queue, and replaces them in the queue with a new node z representing their merger. The frequency of z is computed as the sum of the frequencies of x and y in line 7. The node z has x as its left child and y as its right child. After n-1 mergers, the node left in the queue-the root of the code tree returned in line 9.

The for loop in lines 3-8 is executed exactly n-1 times, and since each heap operation requires time O (lg n), the loop contributes O (n lg n) to the running time. Thus, the total running time of HUFFMAN on a set of n characters is O (n lg n).

3.4.2 H-HIBASE: Encoding

Consider a set of source symbols S = { s_0, s_1, ….. , s_{n-1}}= {Dhaka, Sylhet, Chittagong, ….. , Rajshahi} with frequencies W = { w_0, w_1, ….. , w_{n-1}} for w_0>=w_1>=…..>=w_{n-1}, where the symbol s_i has frequency w_i. Using the Huffman algorithm to construct the Huffman tree T, the codeword c_i, 0<=i<=n-1, for symbol s_i can then can be determined by traversing the path from the root to the left node associated with the symbol s_i, where the left branch is corresponding to '0' and the right branch is corresponding to '1'. Let the level of the root be zero and the level of the other node is equal to summing up its parents level and one. Codeword length l_i for s_i can be known as the level of s_i.



Division	Freq	Code
Dhaka	3	00
Sylhet	3	01
Rajshahi	2	10
Chittagong	2	11

Fig. 3.6: Construction of Huffman Tree for Division column.

The wighted external path length $\sum w_i l_i$ is minimum. For example, the Huffman tree corresponding to the source symbols { s_0, s_1, ……..,s_7} with the frequencies {3, 3, 2, 2} is shown in the Fig.3.6. the codeword set C{c_0,c_1,…….,c_7} is derived as {10, 11, 00, 01}. In addition, the codeword set compose of a space with 2^d addresses, where d=2 is the depth of the Huffman tree.

In the following, the detailed algorithm to generate the intervals is presented. For each Huffman tree, the required storage for the interval representation is n entries. Each entry contains two fields: address and symbol. The length of address is d bits, and the storage complexity is O (n).

Both C and C++ allow integer members to be stored into memory spaces smaller than the compiler would ordinarily allow [68]. These space-saving structure members are called *bit fields*, and their width in bits can be explicitly declared. The following structure has three bit-field members: kingdom, phylum, and genus, occupying 2, 6, and 12 bits respectively.

```
struct taxonomy {
    unsigned kingdom: 2;
    unsigned phylum: 4;
    unsigned genus: 12;
            };
```

To store codeword we have declared an array of structure with bit field where data can be stored with 1 bit storage. This structure have 32 members variable named a,b,c,…..,z,A,B,…,F and every member can be stored 1 bit. To put databits in this structure we have a function named putvalue (index_of_structure, data_variable, databit) which store bit into the structure after reading the input from the dictionary.

Algorithm 3.2: Encode (index, name, databit, frequency)

```
ENCODE (Huffman_Dictionary hd)
1. Input: Huffman_Dictionary ( index, name, databit, frequency)
2. Output: Encoded Bit Stream
3. BEGIN
4. for i ←1 to total_number_of_rows
5.     for j ← 0 to codeword [i].lenght
6.         putvalue (index_of_structure, data_variable, databit)
```

```
7.      if (data_variable == 'z')
8.          data_variable ← 'A'
9.      else if (data_variable == 'F')
10.             data_variable ← 'a'; index_of_structure ++
11.     else data_variable ++
12. END
```

In algorithm 3.2 it has been shown that index, frequency and codeword of a particular record has been read from the dictionary first. After that it store in the storage bitwise with the repetition of number of frequencies. And this process has continued until the last record of the dictionary. The required storage for the interval representation is n entries and the storage complexity is O (n).

3.4.3 Query Operation: Selection

To search a value in the compressed storage it is necessary to access the dictionary first. The start position of the searched value has been calculated from the dictionary by a function named findstartposition (searchedvalue). The end position of the searched value can also be calculated by another function named findendposition (searchedvalue). By using start and end position of searched value it can easily be found from the array.

Algorithm 3.3: Searching (Searched value)

```
SEARCH (string searchedvalue)
1.  Input: The Searched Value
2.  Output: The matching interval
3.  BEGIN
4.    for  I ← 1 to number_of_coderword_in_dictionary
5.        if (inputdata=userdata)
6.            position ← i
7.    sp ← findstartposition (position)
8.    ep ← findendposition (position)
9.    if the searched codeword is matched between the codeword of sp and ep
10.     print Found
```

```
11.    else
12.        print Not found
13. END
```

The details algorithm is listed above. The time complexity for decoding is O (n).

3.4.4 Query Operation: Insertion

To insert a new record in the database multiple action is required. First of all data has been inserted in the input file, dictionary has been updated by using function HUFFMAN (C), storage has been refreshed with the function named ENCODE (Huffman_Dictionary).

Algorithm 3.4: Insertion (Inserted value)

```
INSERT (string InsertedValue)
1. Take inserted value as input
2. BEGIN
3.    Insert a new raw as the last tuple of input file
4.    Call HUFFMAN (C)
5.    Call ENCODE(Huffman_Dictionary)
6. END
```

3.4.5 Query Operation: Deletion

To delete a record from the database multiple actions is required. First of all data has been deleted from the input file, dictionary has been updated according to the new file by using function HUFFMAN (C), storage has been refreshed with the function named ENCODE (Huffman_Dictionary).

Algorithm 3.5: Deletion (Deleted value)

```
DELETE (string DeletedValue)
1. Take deleted value as input
2. BEGIN
3.    Search deleted item in the input database
```

```
4.      if found delete the item by left shifting
5.          Call HUFFMAN (C)
6.          Call ENCODE(Huffman_Dictionary)
7.      else print "data cannot be deleted"
8.  END
```

3.4.6 Query Operation: Update

To update data in the database multiple actions is required. First of all data has been updated from the input file, dictionary has been updated according to the new file by using function HUFFMAN (C), storage has been refreshed with the function named ENCODE (Huffman_Dictionary).

Algorithm 3.6: Update (Old value, New value)

```
UPDATE (string Oldvalue, string  Newvalue )
1.  Take updated value with old value as input
2.  BEGIN
3.  Search old value in the input file
4.      If found update the input file by replacing new value with the old    value
5.          Call HUFFMAN (C)
6.          Call ENCODE (Huffman_Dictionary)
7.      Else print "data can not be updated"
8.  END
```

3.4.7 Query Operation: Aggregate Function

Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value. SQL provides five different built-in aggregate functions: count, max, min, sum and avg. The input of sum and avg must be a collection of numbers, but other operators can operate on collections of non-numeric data types, such as strings, alpha-numeric, as well.

For aggregation queries we have considered the following relation:
account(account_no, branch_name, balance)

3.4.7.1 Count

Select branch_name, count (branch_name) from account.

Algorithm 3.7: Count ()

```
COUNT ()
1. Initialize count=0
2. Read dictionary
3. Loop until finish the number of tuple
4.        Count++
5. Print Count
```

Algorithm 3.7 is indicated that record has been counted from the dictionary, for each frequency it increases count by 1 until reach the last frequency.

3.4.7.2 Sum/Avg

Select branch_name, sum (balance) from account.

Algorithm 3.8: Sum ()/Avg ()

```
SUM/AVG ()
1. Read dictionary
2. Initialize sum=0
3. Put number in an array
4. For 1 to size of array (count)
5.        Sum=sum + number
6. Print sum
```

In the above algorithm it has been shown that the every number has been added with the previous number in the array, loop continues until reach the last entry.

3.4.7.2 Max/Min

Select branch_name, max (balance) from account.

Algorithm 3.9: Max ()/ Min ()

```
MAX/MIN ()
```

```
1. Read dictionary
2. Initialize maximum=0
3. Put number in an array
4. For 1 to size of array (count)
5.         If (current value>maximum value)
6.         Maximum = current value
7. Print maximum
```

Algorithm 3.9 has been used to find the maximum number. In the algorithm it has been shown that any larger number replaced by smaller number in the array.

3.5 Summary

In this chapter we have presented an attractive compression-based architecture, called H-HIBASE. Due to disk based compression H-HIBASE support very large database with acceptable storage volume. Insertion, deletion and update mechanisms on the architecture have been presented and analyzed. The architecture executes query directly on compressed data and it is capable of executing most of SQL queries. Algorithms of query operators given in this chapter have been thoroughly analyzed.

Chapter 4
Result and Discussion

The objective of the experimental work is to verify the applicability and feasibility of the proposed H-HIBASE architecture. The experimental evaluation has been performed with synthetic and real data. The experimental results are compared with DHIBASE and widely used Oracle 10g. Our target was to handle relations and justify the storage requirements and query time in comparison with DHIBASE and Oracle 10g.

4.1 Experimental Environment

H-HIBASE has been tested on a machine with 1.73 GHz Pentium IV processor and 1 GB of RAM, running on Microsoft Windows XP. Five different relations have been created for synthetic data which are given below. Each query has been executed five times and the average execution time has been taken.

4.1.1 Data Set

Data set for Synthetic data:

A random data generator has been used to generate synthetic data and large number of records has been inserted into each table. There are five tables in synthetic data set are given below, where first attribute of each table is the primary key. Synthetic data generator has been generated 21035, 21120, 21214, 31422, 30455 records for *Distributor, Customer, Item, Employee* and *Store* relations respectively. To store same number of records using Oracle 10g it required 1 MB disk space. Table 4.1 shows overall compression rate of different technique for different number of record. Compression rate has been calculated with respect to DHIBASE and Oracle 10g. Table 4.1 also shows overall CF's for different relations, CF's has been calculated with respect to Oracle 10g. It has been observed that the proposed system outperforms Oracle 10g by a factor of 11 to 13. Table 4.1 shows that the H-HIBASE has greater compression capability than DHIBASE with enhancement rate between 9% to 15% as well.

Distributor (d_id, fname, lname, area)

Customer (c_id, name, street, city)

Item (i_id, type, description)

Employee (e_id, name, department)

Store (s_id, location, type)

Data set for real data:

Billing Management Software has been used to manage different types of Bills like water bill, electricity bill. Different types of report like daily bill, monthly bill, yearly bill has been produced by this software. Real data set of this software has been shown below. Storage requirement in different technique for real data has been shown in table 4.2. It has been observed that the proposed system has better than Oracle 10g by a factor of 4 to 5. Table 4.2 also shows that the H-HIBASE has greater compression capability than DHIBASE which is more than 30%.

Electricbill (issue_no, meterno, presentreading, surcharge, shop_id, tannent_id, bill_month, unit_rate number, issue_date, paid_date, demand_charge, meter_charge, last_date, ref_no, consume, vat, previousreading)

Electricbill_for_shop (meter_no, meter_rgd, prv_surcharge, prv_vat, prv_demand, prv_from_date, prv_to_date, shop_id, tannent_id, paid_date, last_date, prv_metercharge, unit_rate, prv_consume, issue_date, is_due, max_rgd, ref_no, prv_arrear, meter_charge, demand_charge)

Floor (floor_id, floor_name)

Rate (rate_id, rate_title, charge)

Shop (shop_number, shop_name, shop_rent, shop_floor_number, shop_id, tannent_id)

Tennant (tannent_id, tannent_name, account_no, address, phone)

Utility_bill (issue_no, bill_month, last_date, paid_date, shop_id, tannent_id, issue_date, ref_no)

Utility_bill_detail (issue_no, utility_id, bill_amt, surcharge)

Utility_bill_for_shop (utility_id, default_amount, prv_amount, prv_surcharge, prv_from_date, prv_to_date, shop_id, tannent_id, last_date, paid_date, is_due, ref_no)

Utility_setup (utility_id, utility_title, default_bill)

4.1.2 Data Generation Algorithm

A random data generator has been used to generate synthetic data and large number of records have been inserted into each table. There are five tables in synthetic data set has been given below, where first attribute of each table is the primary key. Our synthetic data generator has been generated 21035, 21120, 21214, 31422, 30455 records for *Distributor, Customer, Item, Employee* and *Store* relations respectively.

Algorithm 4.1: Synthetic Data Generator:

```
1     InsertRandomData (RowCount)
2      BEGIN
         LOOP
4         COL1 ← VAR1 ← dbms_randon.string('L', 10)
5         COL2 ← VAR2 ← dbms_randon.string('L', 10)
              .
              .
8         COLN ← VARN ← dbms_randon.string('L', 10)
9            LOOP
10             IF mod(counter,50)=0 THEN
11                  REPEAT step 4 to 8
12             END IF
13             InsertData (COL1, COL2, ...,COLN)
14             Counter ← counter+1
15             Exit when counter>=rowcount
16           END LOOP
          END LOOP
17      END
```

From the above algorithm it has been observed that the random data generation function has been generated an amount of random data for a column, which has been inserted into the database table.

4.2 Storage Requirement

4.2.1 Synthetic Data

Table 4.1 shows the storage for some relation in different technique. It has also indicated the Compression Factor (CF) for H-HIBASE and Oracle database system. It shows compression enhancement of H-HIBASE with compare to DHIBASE as well.

Table 4.1: Compression achieved in different techniques for synthetic data (KB)

Relation	Record	Oracle 10g.	DHIBASE	H-HIBASE	Overall CF	Enhancement Rate (%)
Distributor	21035	1024	96.28	81.84	12.51	14.99
Customer	21120	1024	96.65	82.17	12.47	14.98
Item	21214	1024	97.10	82.54	12.40	14.99
Employee	31422	1024	95.88	86.30	11.86	10.02
Store	30455	1024	92.93	83.64	12.24	9.99

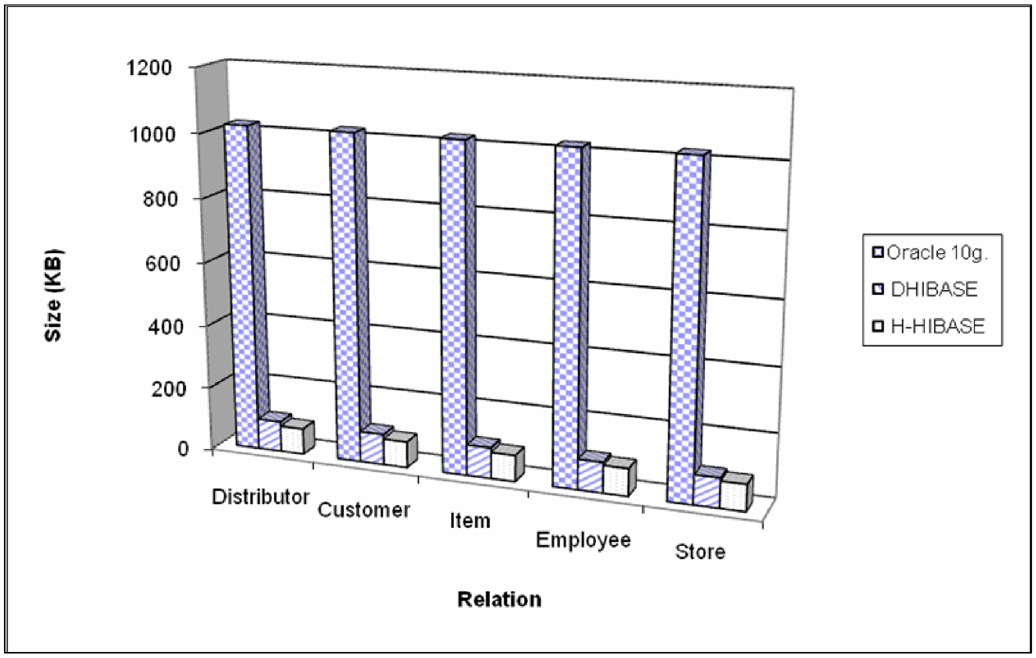


Fig. 4.1: Storage in H-HIBASE, DHIBASE and Oracle 10g.

Fig. 4.1 shows the storage comparison among H-HIBASE, DHIBASE and Oracle 10g. In the figure it has been indicated that DHIBASE can be compressed the oracle storage with the rate of 90%, whereas H-HIBASE can be compressed the oracle storage with the rate of 92%.

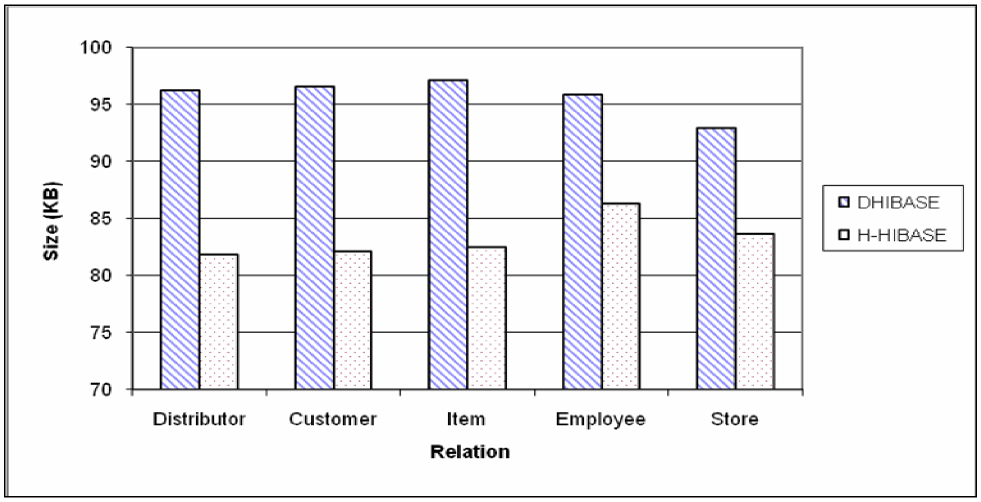


Fig. 4.2: Storage in H-HIBASE and DHIBASE

Fig. 4.2 indicates the storage comparison between H-HIBASE and DHIBASE. In the figure it has been indicated that H-HIBASE has better compression capability with the rate 30%. This is because DHIBASE has been used fixed length coding whereas H-HIBASE has been used variable length Huffman coding which need a reduced amount of storage.

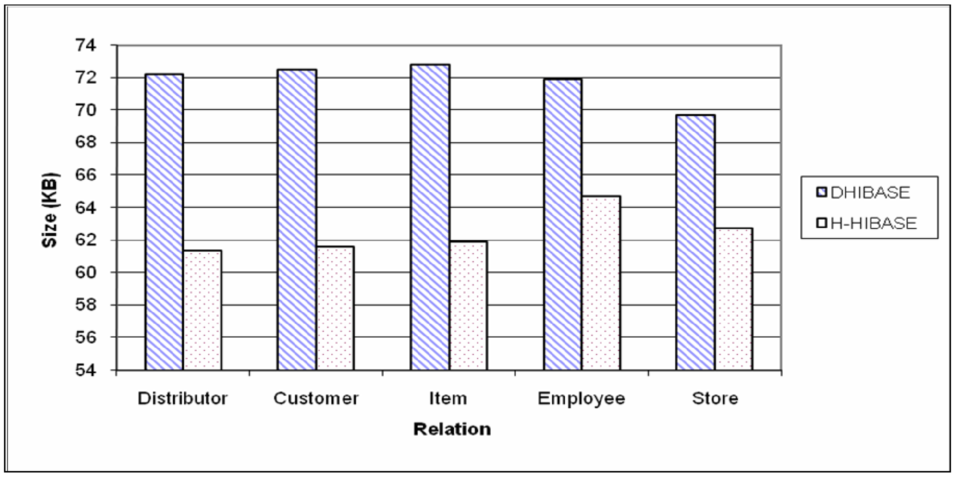


Fig. 4.3: Code size in DHIBASE and H-HIBASE

The storage of code size in DHIBASE and H-HIBASE technique is shown in Fig. 4.3. From the figure, H-HIBASE produces minimum number of code to store entire relation than that of DHIBASE system. DHIBASE has needed around 70 KB to store code size, whereas H-HIBASE has needed around 60 KB to store code size for the same number of records.

4.2.2 Real Data

Real data set has been shown below. Storage requirement in different technique for real data has been shown in table 4.2. It has been observed that the proposed system has better than Oracle 10g by a factor of 4 to 5. Table 4.2 also shows that the H-HIBASE has greater compression capability than DHIBASE which is more than 30%. Higher storage requirement has been avoided by using Huffman code-words in H-HIBASE technique. Moreover high performance has been ensured as most repeated attribute values get more weight and entered first in the dictionary i.e. domain dictionary values sorted in such a way that frequently occurred values accessed first than the rare values.

Table 4.2: Compression achieved in different techniques for real data (KB)

Relation	Column	Record	Oracle 10g.	DHIBASE	H-HIBASE	Enrichment Rate (%)
Electric bill	17	1505019	381424	126347.42	88152.89	30.23
Electric_bill_for_shop	17	570				
Floor	3	8				
Rate	3	8				
Shop	6	583				
Tennant	6	292				
Utility_bill	8	1550987				
Utility_bill_detail	4	6206961				
Utility_bill_for_shop	13	2266				
Utility_setup	3	6				

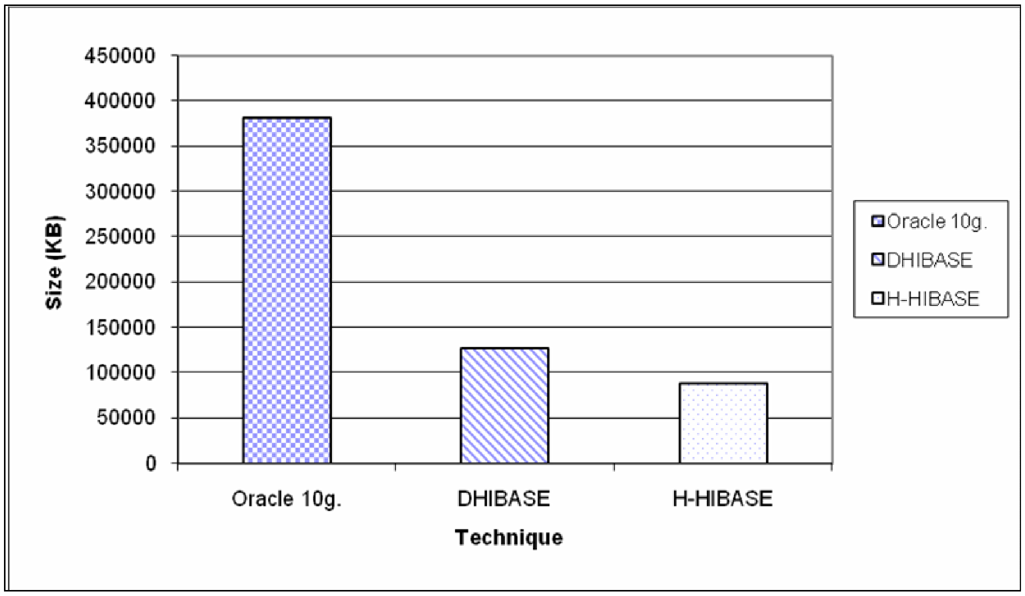


Fig. 4.4: Storage of real data in H-HIBASE, DHIBASE and Oracle 10g.

Fig. 4.4 shows the comparison of storage size among Oracle database, DHIBASE, and H-HIBASE. To store same number of record it is required approximately 380 MB, 125 MB, and 85 MB in Oracle 10g, DHIBASE, and H-HIBASE respectively. H-HIBASE technique has more compression capability than any other existing systems.

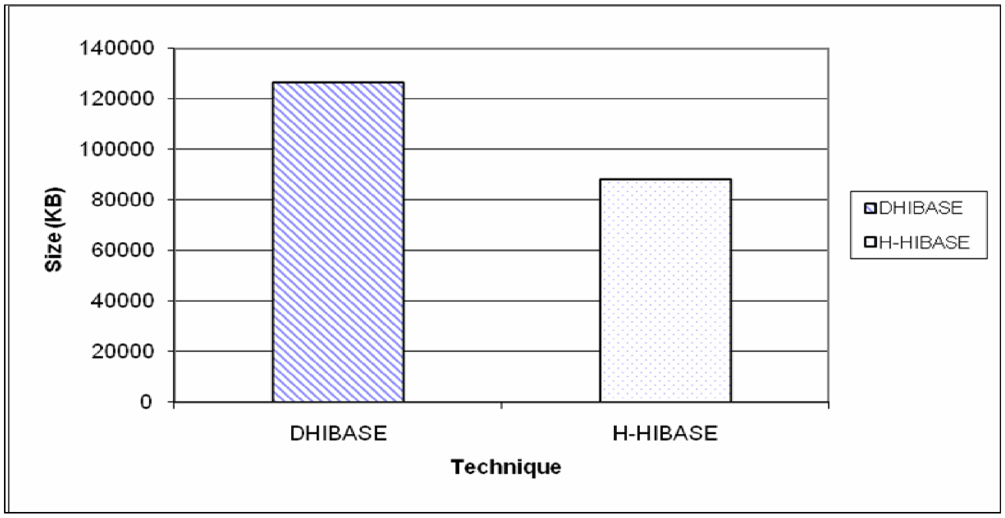


Fig. 4.5: Storage of Real Data in H-HIBASE and DHIBASE

Fig. 4.5 indicates the storage comparison between H-HIBASE and DHIBASE. In this figure H-HIBASE has better compression capability with the rate of more than 30%.

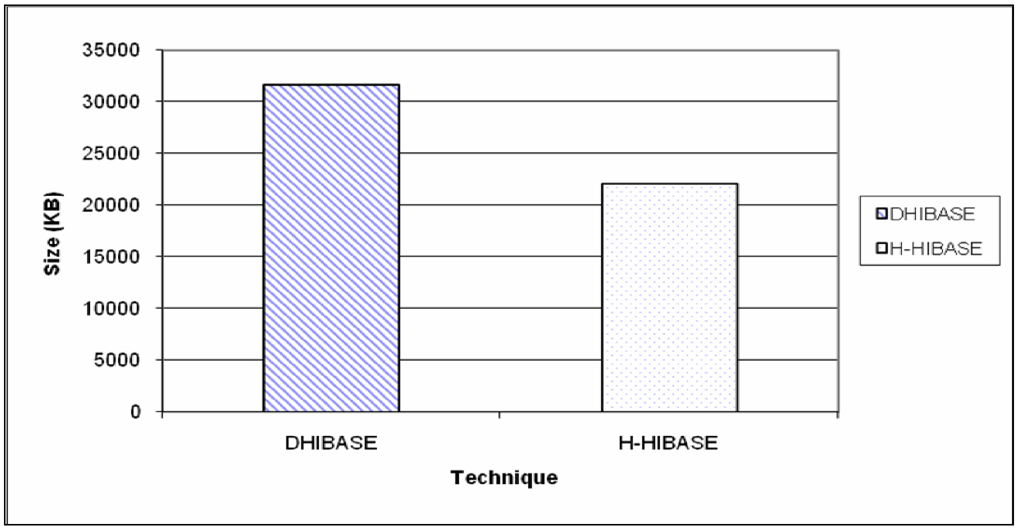


Fig. 4.6: Code size in DHIBASE and H-HIBASE

Fig. 4.6 shows the code size comparison between H-HIBASE and DHIBASE. H-HIBASE is space efficient, this is because DHIBASE has used fixed length coding whereas H-HIBASE has used variable length Huffman coding. Variable length coding required smaller amount storage than fixed length coding.

4.3 Query Performance

To assess query performance, we carried out queries on both DHIBASE and H-HIBASE. The performed queries and obtained results are described in the following sub-sections. In all cases Distributor relation contains 0.1, 0.4, 0.7, 1.0 million records. Item, Employee, Store and Customer relations contain 1000, 2000, 100, 10000 records respectively. All queries executed in H-HIBASE system are directly applied on compressed data. Given query is first converted into compressed form and compressed query is executed.

4.3.1 Single Column Projection

We have executed the following query and the result is shown in figure 4.7.

select Name from Customer

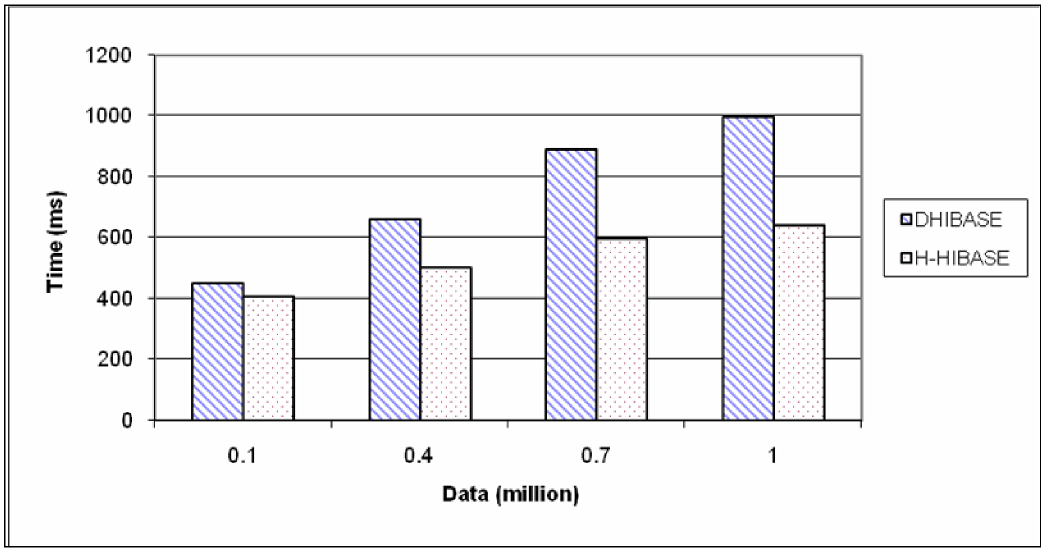


Fig. 4.7: Single column projection

Figure 4.7 shows that H-HIBASE is faster than that of DHIBASE in case of projection operation. This is obvious because H-HIBASE stores data in compressed form with minimum storage. Therefore, to find a particular record it required to search a smaller amount space. This is the main reason of speed-gain in H-HIBASE system.

4.3.2 Two Column Projection

We have executed the following query and the result is shown in figure 4.8.

select Name, Street from Customer

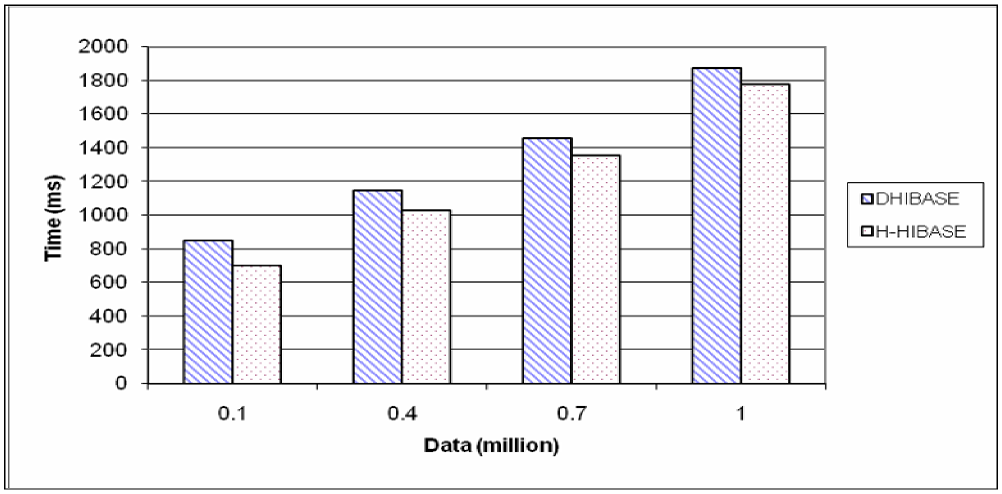


Fig. 4.8: Two column projection

Figure 4.8 shows that H-HIBASE is quicker than that of DHIBASE in case of two column projection operation. This is obvious because H-HIBASE stores data in compressed form with minimum storage. The processing speed of a query is enhanced because queries specify operations only on a subset of domains. In a column-wise database only the specified values need to be transferred, stored and processed. This requires only a fraction of the data that required during processing by rows.

4.3.3 Three Column Projection

We have executed the following query and the result is shown in figure 4.9.

select Name, Street, City from Customer

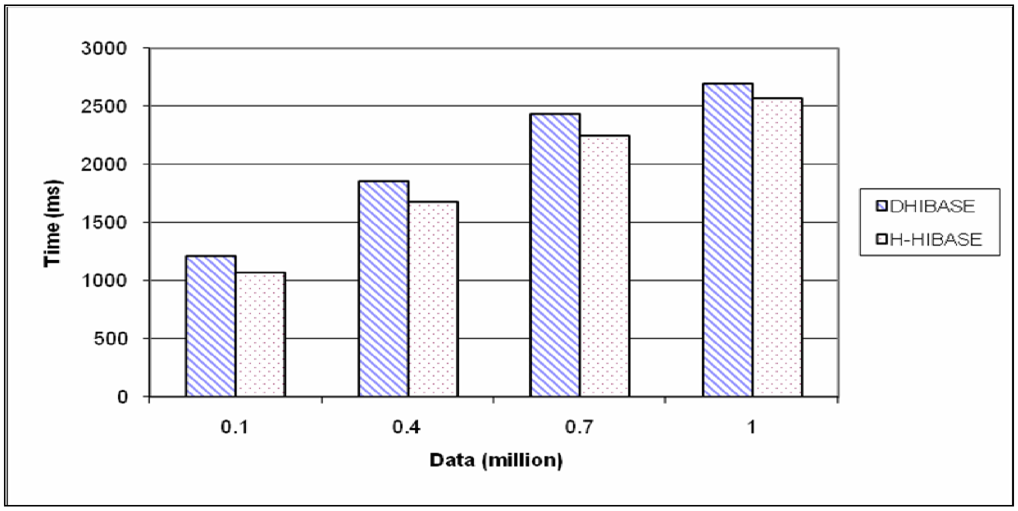


Fig. 4.9: Three column projection

Figure 4.9 shows that H-HIBASE has better performance than that of DHIBASE in case of three column projection operation. H-HIBASE stores data in compressed form with minimum storage. In three column projection it has taken more execution time than single column projection operation. It requires more access time to collect data from the dictionary.

4.3.4 Full Table Scan

We have executed the following query and the result is shown in figure 4.10.

*select * from Customer*

Fig. 4.10 shows that the performance of H-HIBASE and DHIBASE shows similar result of three column projection which is shown in Fig. 4.9. This is because the *Customer* relation has three columns.

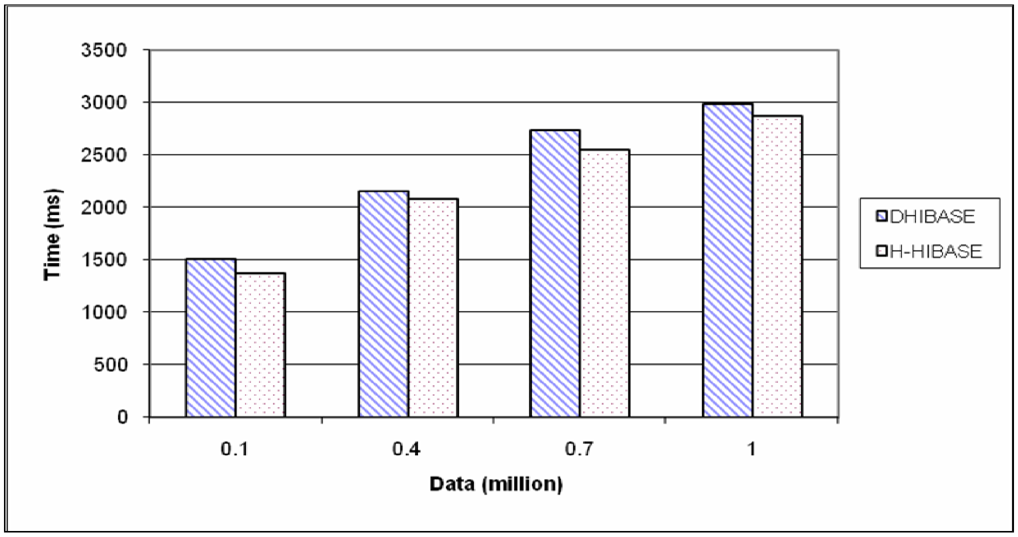


Fig. 4.10: Full Table Scan

4.3.5 Single Predicate Selection

We have executed the following query and the result is shown in figure 4.11.

select Name from Customer where City = "Dhaka"

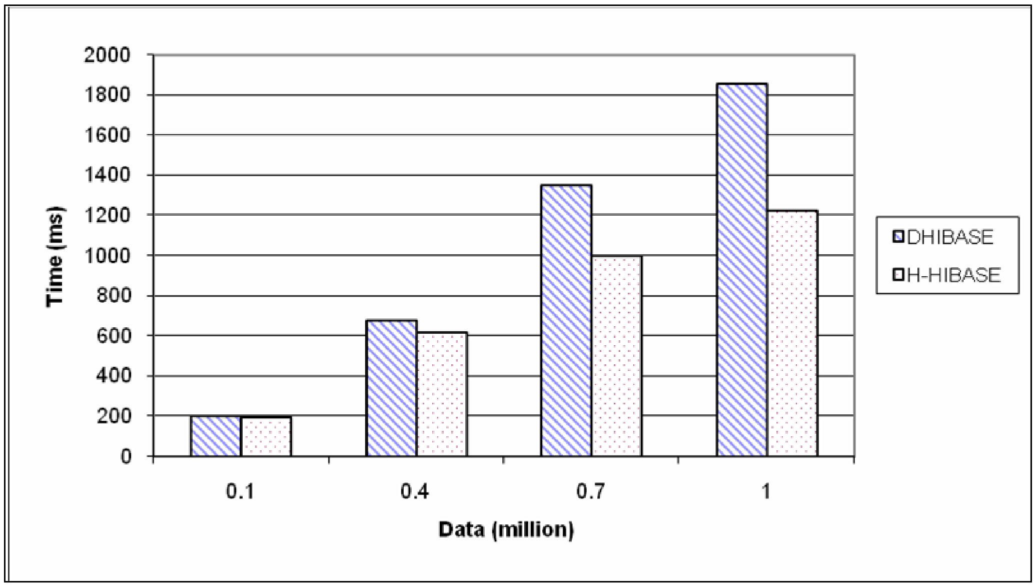


Fig. 4.11: Single predicate selection

Figure 4.11 shows that H-HIBASE does not better performance than DHIBASE in case 0.1 million to 0.4 million records but faster in case 0.7 million and 1.0 million records. In case of 1 million records, it reads most repetitive values first from the dictionary, and takes a reduced amount of time to access it from the storage. The processing speed of predicate selection query is enhanced because queries specify operations only on a subset of domains. In a column-wise database only the specified column need to be accessed. This requires only a fraction of the data that required during processing by rows.

4.3.6 Five Percent Selectivity

We have executed the following query and the result is shown in figure 4.12.

*select * from distributor where rownum < (((select count(*) from distributor)/100) * 5)*

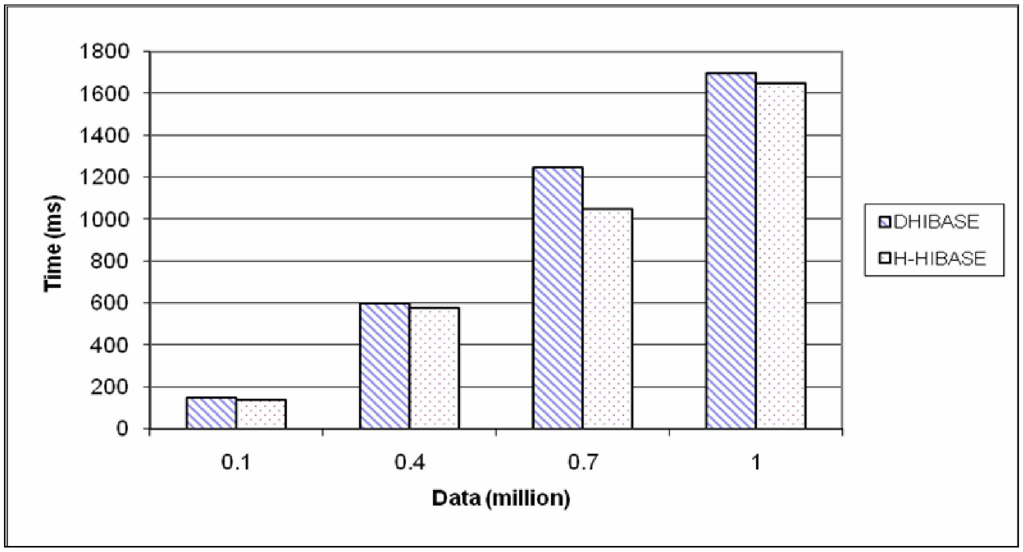


Fig. 4.12: 5% selectivity

Figure 4.12 shows that H-HIBASE performs better performance than DHIBASE in case of 5% selectivity. This is because within this 5% data there are large number of repetition.

4.3.7 Ten Percent Selectivity

We have executed the following query and the result is shown in figure 4.13.

*select * from distributor where rownum < (((select count(*) from distributor)/100) * 10)*

Figure 4.13 show that H-HIBASE show better performance than DHIBASE in case of 10% selectivity, but it is not as efficient as 5% selectivity, because of lower repetition.

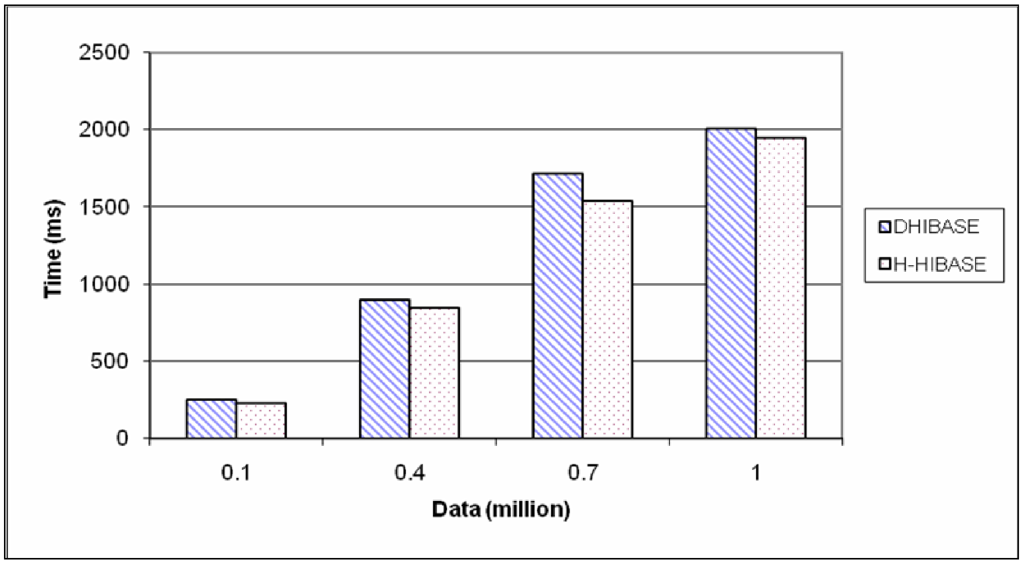


Fig. 4.13: 10% selectivity

4.3.8 Aggregate Function: Count

For aggregation queries we have considered the following relation:
account(account_no, branch_name, balance).

We have executed the following query and the result is shown in figure 4.14.

Select branch_name, count (branch_name) from account.

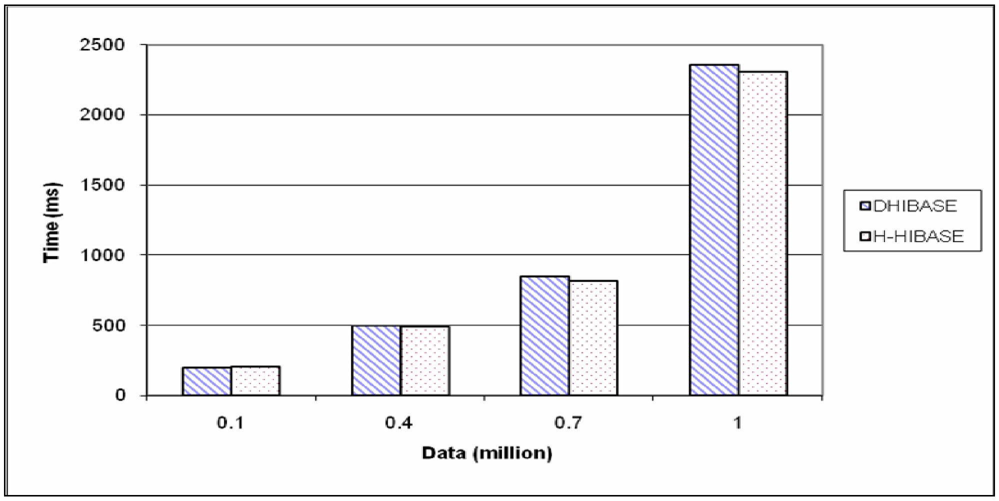


Fig. 4.14: Aggregate function: Count

We assume that the relation *account* is already sorted by account_no according to dictionary code. In case of 0.1 to 1 million records, H-HIBASE read all distinct values from dictionary to calculate the result. Hence the performance is almost same with DHIBASE.

4.3.9 Aggregation: Max/ Min/ Sum/ Avg

We have calculated the following queries and the result is shown in figure 4.15
select account_no, max (balance) from account group by account_no
select account_no, sum (balance) from account group by account_no
select account_no, avg (balance) from account group by account_no

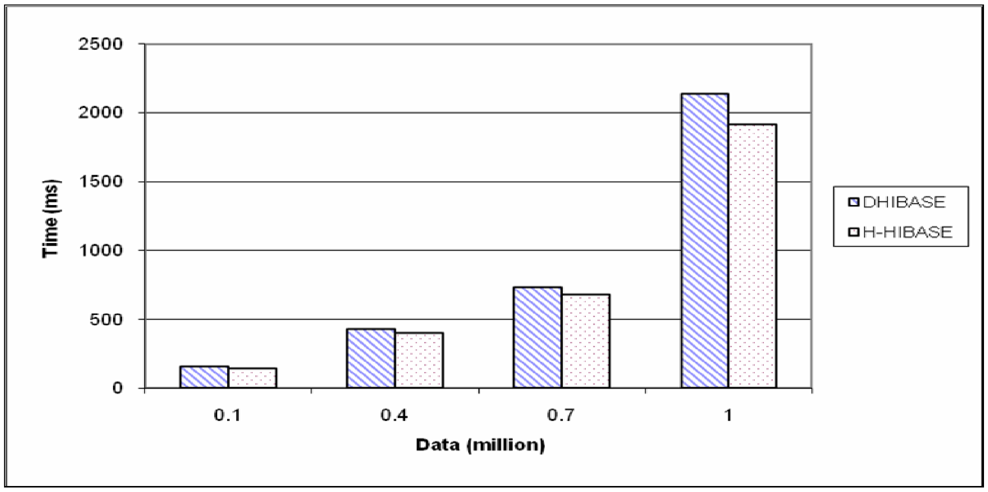


Fig. 4.15: Aggregate function: Max/Min/Sum/Avg.

H-HIBASE read all distinct values from dictionary to calculate the result. In case of fewer amounts of data performance is almost same in both techniques, and when storage increases H-HIBASE perform better because of its repetition.

4.4 Worst Case Analysis

In the above analysis we have observed that the H-HIBASE technique perform better than that of DHIBASE in case of storage and query time. It just because H-HIBASE has been used Huffman Principle to generate its dictionary and H-HIBASE store its code-words into column wise format into the disk. As Huffman Principles generate shorter code-word for most repetitive values, hence compression has been optimized where repetition is regular. If there is no repetition than Huffman generate a codeword for every string value and require more space to store these bigger length code-words in the disk. In the worst case scenario, the performance of H-HIBASE is same with DHIBASE; both of them require same space to store all random code-words.

4.5 Summary

In this chapter we have presented the experimental evaluation of the H-HIBASE architecture. We evaluated the storage performance in comparison with DHIBASE and Oracle 10g. The storage performance that is achieved in H-HIBASE is 25 to 40 percent better than the Oracle 10g for real and synthetic data. It has also been shown that the storage performance that is achieved in H-HIBASE is 15 to 35 percent better than the DHIBASE. The query performance that is achieved in H-HIBASE is 10 to 25 percent better than that of DHIBASE.

Chapter 5
Conclusion and Future Research

Database compression is attractive for two reasons: storage cost reduction and performance improvement. Both are essential for management of large databases. Direct addressability of compressed data is necessary for faster query processing. It is also important for queries to be processed in compressed form without any decompression. Literature survey shows that compression techniques used in memory resident databases are not suitable for large databases when database cannot fit into memory. We have improved the basic HIBASE model and DHIBASE model for disk support. We have also improved query processing capability of the basic system. We have defined a number of operators for querying compression-based relational database system, designed algorithms for these operators and thoroughly analyzed these algorithms.

5.1 Fundamental Contributions of the Thesis

- ❖ The main contribution of this research is to develop a compression technique that is enhancement of HIBASE technique using HUFFMAN coding (H-HIBASE) with better compression capability.

- ❖ Compressed data are stored using the H-HIBASE architecture with disk support. This overcomes the scalability problems of the memory resident DBMS.

- ❖ Considerable storage reduction has been achieved using the H-HIBASE architecture. The experimental results show that H-HIBASE architecture is 15 to 35 times space efficient than that of HIBASE and DHIBASE.

- ❖ We have designed algorithms for most of the relational algebra operations that support most of the commercial database systems. Experimental results show that the H-HIBASE system has better performance for insertion, deletion, update operations on single relation compared to Oracle database. In case of selection operation, H-HIBASE is significantly better than DHIBASE.

5.2 Future Research

The H-HIBASE architecture has been implemented in a single processor system and achieved significant performance improvement over existing compression based systems. H-HIBASE is disk based database compression architecture. The future expansion of this research is to explore the following issues:

- ❖ The architecture can be used for parallel database environment to achieve scalable performance for data warehouse application.
- ❖ We have not considered any back-up and recovery mechanism for H-HIBASE architecture. These features may be included.
- ❖ To achieve concurrent access to H-HIBASE architecture, a multi-threaded algorithm can be considered to support multi-user DBMS.
- ❖ To achieve faster query performance for multiple relation join, set and aggregation operations, better algorithm may be designed.

Bibliography

[1] Tashenberg, C. B., “Data management isn’t what it was,” Data Management Direct Newsletter, May 24, 2002.

[2] Rouf, M. A., “Scalable storage in compressed representation for terabyte data management,” M. Sc. Thesis, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh, 2006.

[3] Cormack, G. V., “Data compression on a database system,” Communication of the ACM, Vol-28, No. 12, pp 1336–1342, 1985.

[4] Helmer, S., Westmann, T., Kossmann, D. and Moerkotte, G., “The implementation and performance of compressed databases,” SIGMOD Record, Vol-29, No. 3, pp 55–67, 2000.

[5] Roth, M. A. and Horn, S. J. V., “Database compression,” SIGMOD Record, Vol22, No. 3, pp 31–39, 1993.

[6] Graefe, G. and Shapiro, L., “Data compression and database performance,” ACM/IEEE-CS Symposium on Applied Computing, pp 22-27, April 1991.

[7] Oracle Corporation, “Table compression in Oracle 9i: a performance analysis, an Oracle whitepaper,” http://otn.oracle.com/products/bi/pdf/o9ir2_ compression_performance_twp.pdf.

[8] Ramakrishnan, R., Goldstein, J. and Shaft, U., “Compressing relations and indexes,” Proceedings of the IEEE Conference on Data Engineering, pp 370–379, Orlando, Florida, USA, February 1998.

[9] Poess, M. and Potapov, D., “Data compression in Oracle,” Proceedings of the 29 VLDB Conference, pp 937-947, Berlin, Germany, September 2003.

[10] Silberschatz, A., Korth, H. F. and Sudarshan, S., “Database system concepts,” 5th Edition, McGraw-Hill, 2006.

[11] McGregor, D., Cockshott, W. P. and Wilson, J., “High-performance operations using a compressed architecture,” The Computer Journal, Vol-41, No. 5, pp 283– 296, 1998.

[12] Hoque, A. S. M. L., “Compression of structured and semi-structured information,” Ph. D. Thesis, Department of Computer and Information

Science, University of Strathclyde, Glasgow, UK, 2003.

[13] Cockshott, W. P., McGregor, D. and Wilson, J., "High-Performance Operation Using a Compressed Database Architecture," The Computer Journal, Vol. 41, No. 5, pp. 285-296, 1998.

[14] Lehman, T. J., Carey, M. J., "A Study of Index Structures for Main Memory Database Management Systems," Proceedings of the Twelfth International Conference on Very Large Databases, pp. 294-303, August 1986.

[15] Bhuiyan, M. M., Hoque, A. S. M. L., "High Performance SQL Queries on Compressed Relational Database," Journal of Computers, Vol. 4, No. 12, pp 1263-1274, December 2009.

[16] Hoque, A. S. M. L., "Compression-Based Models for Processing of Structured and Semi-stuctured Data," Proceedings of the Workshop DATESO 2003, pp 1-20, ISBN 80-248-0330-5, 2003.

[17] Alom, B. M. M., Henskens, F. and Hannaford, M., "Single Vector Large Data Cardinality Structure to Handle Compressed Database in a Distributed Environment," Springer Berlin Heidelberg 2009, pp 147-160, ISBN 978-3-642-05200-2, 2009.

[18] Huffman, D. A., "A Method for the Construction of Minimum Redundancy Codes," Proceedings of the I.R.E., vol. 40, pp. 1098-1101, September 1952.

[19] Kavousianos, X., Kalligeros, E. and Nikolos, D., "Multilevel Huffman Coding: An Efficient Test-Data Compression Method for IP Cores," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 26, No. 6, pp 1070-1083, June 2007.

[20] Hashemian, R., "Memory Efficient and High-Speed Search Huffman Coding," IEEE Transactions on Communication, Vol. 43, No. 10, pp 2576-2581, October 1995.

[21] McGregor, D. R. and Hoque, A. S. M. L., "Improved compressed data representation for computational intelligence systems," In UKCI-01, Edinburgh, UK, September 2001.

[22] Held, G. and Marshel, T. R. , "Data and Image Compression," Number 0-47195247-8, John Wiley and Sons Ltd, West Sussex, England, 1996.

[23] Huffman, D.A., "A method for the construction of minimum-redundancy code," In Proceedings of IRE, Vol-40, No. 9, pp 1098–1101, 1952.

[24] Fano, R.M., "The transmission of information," In Research Laboratory for Electronics, MIT Technical Report, (65), 1949.

[25] Shannon, C.E., “A mathematical theory of communications,” In Bell system Technical Journal, Vol-27, pp 379–423 and 623–656, 1948.

[26] Vitter, J.S., “Design and analysis of dynamic Huffman code,” In Journal of the ACM, Vol-34, No. 4, pp 825–845, October 1987.

[27] Gallager, R.G., “Variations on a theme by Huffman,” In IEEE Transaction on Information Theory, Vol-24, No. 6, pp 668–674, November 1978.

[28] Lampel, A. and Ziv, J., “A universal algorithm for sequential data compression,” In IEEE Transaction on Information Theory, Vol-23, pp 337–343, 1977.

[29] Lampel, A. and Ziv, J., “Compression of individual sequences via variable rate coding,” In IEEE Transaction on Information Theory, Vol-24, pp 530–536, 1978.

[30] Welch, T. A, “A technique for high-performance data compression,” In IEEE Computer, Vol-17, No. 6, pp 8–19, 1984.

[31] Larson, N. J. and Moffat, A., “Off-line dictionary-based compression,” In Proceedings of the IEEE Data Compression Conference, Snowbird, Utah, March 2000.

[32] Golomb, S. W., “Run-length encodings,” In IEEE Transaction on Information Theory, Vol-12, No. 3, pp 399-401, 1966.

[33] Chen, Z., Gehrke, J. and Korn, F., “Query optimization in compressed database systems,” In SIGMOD ’01: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, pp 271–282, ACM Press, 2001.

[34] Oberhummer, M. F. X. J. , “LZO: A real-time data compression library,” 2002, http://www.oberhumer.com/opensource/lzo/lzodoc.php.

[35] Moffat, A. and Zobel, J., “Parameterised compression for sparse bitmap,” In Proceedings of the 15 Annual International SIGIR 92, pp 274–285, ACM, 1992.

[36] Wee, K. N. and Ravishankar, C. V., “Relational database compression using augmented vector quantization,” In Proceedings of the 11th International Conference on Data Engineering, pp 540–550, Taipei, Taiwan, 1995. IEEE.

[37] Cuperman, V. and Gersho, A., “Vector quantization: A pattern-matching

technique for speech coding," In IEEE Communication Magazine, Vol. 21, pp 15–21, December 1983.

[38] Kotsis, N., Wilson, J., Cockshott, W. P. and McGregor, D., "Data compression in database systems," In Proceedings of the IDEAS, pp 1–10, July 1998.

[39] Codd, E. F., "A relational model of data for large shared data banks," Communication of the ACM, Vol-13, No. 6, pp 377–387, 1970.

[40] Short history of HUFFMAN Coding: http://en.wikipedia.org/wiki/ Huffman_ coding last access on 31 July 2011.

[41] Jones, D. W, "Application of Splay Trees to Data Compression," Communication of ACM, August 1988.

[42] Witten, I. H, "Arithmetic Coding for Data Communication," Communication of ACM, June 1987.

[43] Xu, Y., Agrawal, R. and Somani, A., "Storage and querying of e-commerce data," In Proceedings of the 27 VLDB Conference, pp 149–158, Roma, Italy, 2001.

[44] Wilson, J., Hoque, A. S. M. L. and McGregor, D. R., "Database compression using an off-line dictionary method," ADVIS, LNCS, Vol-24, pp 11–20, October 2002.

[45] Aho, A. V., Sethi, R. and Ullman, J. D., "Compilers: Principles, Techniques, and Tools," Addison Wesley, 1986.

[46] Knuth, D. E., "The art of computer programming," Vol-3, Addison Wesley, Sorting and Searching, 1973.

[47] Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A. and Price, T. G., "Access path selection in a relational database system," Proc. of the ACM SIGMOD Conf. on Management of Data, pp 23-34, 1979.

[48] Graefe, G. and McKenna, W., "The Volcano optimizer generator," Proc. of the International Conf. on Data Engineering, pp 209-218, 1993.

[49] Haas, L. M., Freytag, J. C., Lohman, G. M. and Pirahesh, H., "Extensible query processing in Starburst," Proc. of the ACM SIGMOD Conf. on Management of Data, pp 377-388, 1989.

[50] Oracle 8 concepts manual, Oracle Corporation, Redwood Shores, 1997.

[51] Dayal, U., Goodman, N. and Katz, R. H., "An extended relational algebra with

control over duplicate elimination," Proc. of the ACM Symposium on Principles of Database Systems, 1982.

[52] Seshadri, P., Pirahesh, H. and Lueng, T. Y. C., "Complex query decorrelation," Proc. of the International Conf. on Data Engineering, pp 450-458, 1996.

[53] Roy, P., Seshadri, S., Sudarshan, S. and Bhobhe, S., "Efficient and extensible algorithms for multi-query optimization," Proc. of the ACM SIGMOD Conf. on Management of Data, 2000.

[54] Hall, P. A. V., "Optimization of a single relational expression in a relational database system," IBM Journal of Research and Development, vol-20, No. 3, pp 244-257, 1976.

[55] Dalvi, N. N., Sanghai, S. K., Roy, P. and Sudarshan, S., "Pipelining in multi-query optimization," Proc. of the ACM Symposium on Principles of Database Systems, 2001.

[56] Blasgen, M. W. and Eswaran, K. P., "On the evaluation of queries in a relational database system," IBM Systems Journal, Vol-16, pp 363-377, 1976.

[57] Kitsuregawa, M., Tanaka, H. and MotoOka, T., "Application of hash to a database machine and its architecture," New Generation Computing, No. 1, pp 62-74, 1983.

[58] Shapiro, L. D., "Join processing in database systems with large main memories," ACM Transactions on Database Systems, Vol-11, No. 3, pp 239-264, 1986.

[59] Davison, D. L. and Graefe, G., "Memory-contention responsive hash joins," Proceedings of VLDB Conference, 1994.

[60] Graefe, G., Bunker, R. and Cooper, S., "Hash joins and hash teams in Microsoft SQL Server," Proceedings of VLDB Conference, pp 86-97, 1998.

[61] Klug, A., "Equivalence of relational algebra and relational calculus query languages having aggregate functions," ACM Press, Vol-29, No. 3, pp 699-717, 1982.

[62] Chaudhuri, S. and Shim, K., "Including group-by in query optimization," In Proceedings of VLDB Conference, 1994.

[63] Galindo-Legaria, C., "Outerjoins as disjunctions," Proc. of the ACM SIGMOD Conf. on Management of Data, 1994.

[64] Gupta, A. and Mumick, L. S., “Maintenance of materialized views: problems, techniques and applications,” IEEE Data Engineering Bulletin, Vol-18, No. 2, 1995.

[65] Mistry, H., Roy, P., Sudarshan, S. and Ramamritham, K., “Materialized view selection and maintenance using multi-query optimization,” Proc. of the ACM SIGMOD Conf. on Management of Data, 2001.

[66] Dar, S., Jagadish, H. V., Levy, A. and Srivastava, D., “Answering queries with aggregation using views,” Proceedings of VLDB Conference, 1996.

[67] Coreman, T. H., Leiserson, C. E., Rivest, R. L., Stein, C., “Introduction to Algorithms,” second edition, page 387-388.

[68] Declaring and Using Bit Fields in Structures: http://publib.boulder.ibm.com /infocenter/macxhelp /v6v81/index.jsp?%20 topic=%2Fcom.ibm.vacpp6m.doc %2Flanguage%2Fref%2Fclrc03defbitf.htm last access on 31 July 2011.