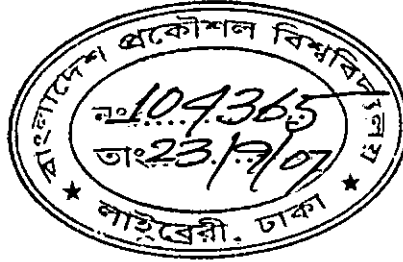
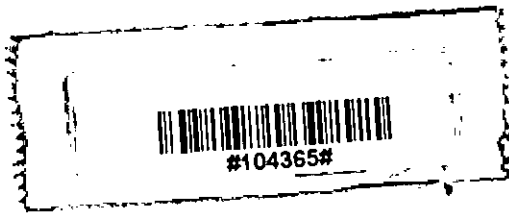


High Performance Queries on Multiple Tables for Compressed Form of Relational Database

By
Mohammad Masumuzzaman Bhuiyan
Roll No. 040405004 F



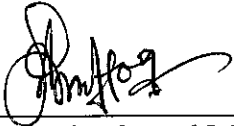



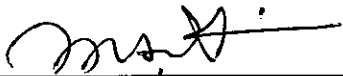
A thesis submitted to the Department of Computer Science and Engineering in
partial fulfillment of the requirements for the degree of MASTER OF
SCIENCE IN COMPUTER SCIENCE AND ENGINEERING



Department of Computer Science and Engineering
BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY, DHAKA
September 2007

The thesis “**High Performance Queries on Multiple Tables for Compressed Form of Relational Database**”, submitted by Mohammad Masumuzzaman Bhuiyan, Roll No. 040405004 F, Session: April 2004, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka, has been accepted as satisfactory for the partial fulfillment of the requirements for the degree of Master of Science in Engineering (Computer Science and Engineering) and approved as to its style and contents for the examination held on 15th September, 2007.

Board of Examiners

- | | |
|--|--------------------------|
| 
_____ | Chairman
(Supervisor) |
| 1. Dr. Abu Sayed Md. Latiful Hoque
Associate Professor, Department of CSE
BUET, Dhaka-1000 | |
| 
_____ | Member
(Ex-officio) |
| 2. Dr. Muhammad Masroor Ali
Professor and Head, Department of CSE
BUET, Dhaka-1000 | |
| 
_____ | Member |
| 3. Dr. Mohammad Mahfuzul Islam
Assistant Professor, Department of CSE
BUET, Dhaka-1000 | |
| 
_____ | Member |
| 4. Dr. Reaz Ahmed
Assistant Professor, Department of CSE
BUET, Dhaka-1000 | |
| 
_____ | Member
(External) |
| 5. Dr. Mohammad Shorif Uddin
Professor, Department of CSE
Jahangirnagar University, Dhaka | |
- R

Declaration

It is hereby declared that the work presented in this thesis or any part of this thesis has not been submitted elsewhere for the award of any degree or diploma, does not contain any unlawful statements and does not infringe any existing copyright.

Signature

M.Z. Bhuiyan

(Mohammad Masumuzzaman Bhuiyan)

Table of Contents

Declaration	i
Table of Contents	ii
List of Figures	iv
List of Tables	v
Acknowledgement	vi
Abstract	vii
Chapter 1 Introduction	1
1.1 Background	2
1.2 Problem Definition	2
1.3 Objective	2
1.4 Organization of the Thesis	3
Chapter 2 Literature Survey	4
2.1 Compression Techniques	4
2.1.1 Loss-Less Compression Methods	5
2.1.2 Lossy Compression Methods	6
2.1.3 Lightweight Compression Methods	6
2.1.4 Heavyweight Compression Methods	6
2.2 Compression on Database Processing	7
2.2.1 Compression of Relational Structures	7
2.2.2 HIBASE Architecture	9
2.2.3 Three Layer Model	12
2.2.4 Columnar Multi Block Vector Structure (CMBVS)	13
2.2.5 Compression in Oracle	13
2.3 Query Processing	14
2.3.1 Uncompressed Query Processing	14
2.3.2 Compressed Query Processing	16
2.4 Summary	17
Chapter 3 DHIBASE Architecture and Compressed Query Processing	18
3.1 DHIBASE: Disk Based HIBASE Model	18
3.2 DHIBASE: Storage Complexity	19
3.3 DHIBASE: Insertion	20
3.4 DHIBASE: Deletion	20
3.5 DHIBASE: Update	21
3.6 Sorting of compressed relation according to code values	21
3.7 Sorting of compressed relation according to string values	23
3.8 Joining of compressed relations	24
3.9 SQL queries on compressed data	25
3.9.1 Queries on single compressed relation	25
3.9.1.1 Projection	26
3.9.1.2 Selection with single predicate	26
3.9.1.3 Selection with multiple predicates	27
3.9.1.4 Selection with range predicate	27

3.9.2 Queries on multiple compressed relations	28
3.9.2.1 Projection	28
3.9.2.2 Selection with single predicate	28
3.9.2.3 Selection with multiple predicates	28
3.9.2.4 Set Union	28
3.9.2.5 Set Intersection	29
3.9.2.6 Set Difference	30
3.9.3 Aggregation on compressed relation	31
3.9.3.1 Count	31
3.9.3.2 Max/ Min	32
3.9.3.3 Sum/ Avg	32
3.10 Summary	33
Chapter 4 Results and Discussion	34
4.1 Experimental Environment	34
4.2 Storage Requirement	36
4.3 Query Performance	37
4.3.1 Single Column Projection	37
4.3.2 Single Predicate Selection	38
4.3.3 Double Predicate Selection	38
4.3.4 Range Predicate Selection	39
4.3.5 Sorting of Relation	40
4.3.6 Aggregation: Count	41
4.3.7 Aggregation: Max/ Min/ Sum/ Avg	41
4.3.8 Natural Join	42
4.3.9 Set Union	43
4.3.10 Set Union/ Set Difference	44
4.3.11 Bulk Update	45
4.4 Summary	46
Chapter 5 Conclusion and Future Research	47
5.1 Fundamental Contributions of the Thesis	47
5.2 Future Research	48
Bibliography	49

List of Figures

<u>Fig. No.</u>	<u>Title</u>	<u>Page No.</u>
Fig. 2.1	Compression of CUSTOMER relation	10
Fig. 2.2	Column-wise storage of a relation	11
Fig. 2.3	The Three Layer Model	12
Fig. 2.4	Querying a database in compressed form	16
Fig. 3.1	DHIBASE Architecture	18
Fig. 4.1	Storage comparison between DHIBASE and SQL Server	36
Fig. 4.2	Single column projection	37
Fig. 4.3	Single predicate selection	38
Fig. 4.4	Double predicate selection	39
Fig. 4.5	Range predicate selection	40
Fig. 4.6	Sorting of <i>Sales</i> selection	40
Fig. 4.7	Aggregation: Count	41
Fig. 4.8	Aggregation: Max/ Min/ Sum/ Avg	42
Fig. 4.9	Natural join	42
Fig. 4.10	Set union	43
Fig. 4.11	Set intersection/ Set difference	44
Fig. 4.12	Bulk update	45

List of Tables

<u>Table No.</u>	<u>Title</u>	<u>Page No.</u>
Table 2.1	CUSTOMER: a simple relation	9
Table 4.1	Item relation	34
Table 4.2	Employee relation	34
Table 4.3	Store relation	35
Table 4.4	Customer relation	35
Table 4.5	Sales relation	35

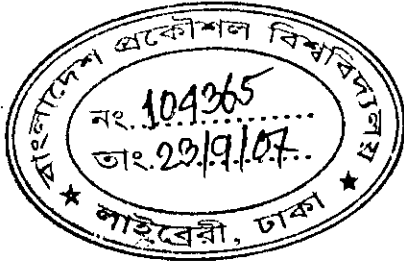
Acknowledgement

I want to express my cordial gratitude to my supervisor, Dr. Abu Sayed Md. Latiful Hoque, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka for making this research as a way of exploring new ideas in the field of database compression. He grows inside me the curiosity and also provides me the proper guidelines that make it possible to complete the research. His profound knowledge and expertise in this field help me to earn relevant knowledge in the field of database compression.

Finally, I would like to acknowledge all of my family members for their heartiest wishes for completing the research.

Abstract

Loss-less data compression is potentially attractive in database application for storage reduction and performance improvement. The existing compression architectures work well for small memory resident database. Some other techniques use disk-based compression and therefore, can support large database. But all these systems can execute a limited number of queries. Moreover, they cannot perform queries based on multiple tables. We have developed a disk based compression architecture that uses dictionary based compression. Each column is stored separately in compressed form. String data are compressed and numeric data may or may not be compressed based on the discretion of the database designer. We have compared our system with widely used Microsoft SQL Server. The experimental result shows that the proposed system requires 10 to 20 times less space. As the system is column oriented, schema evolution is easy. We have also defined a number of query operators on compressed database. We have implemented natural join, selection with range predicate, set operations and all aggregation functions. These complex queries have not been explored in existing compression based systems. Other than selection queries our system outperforms Microsoft SQL Server with respect to query time. The performance of selection queries could be improved by introducing indices. Also the system is appropriate for parallel computation by distributing the compressed columns to separate processors.



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 recently 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

Many different techniques are now available for database compression [3, 4, 5, 6]. But these systems primarily focus on compression and have limited focus on efficient query processing. These systems support insertion, deletion, update, projection and selection operation on single table. Most of the database applications require supporting queries on multiple tables. But these systems do not support queries on multiple tables, sorting of compressed relations, join operations, aggregations and other complex queries. So size reduction should not be the sole objective of a compressed database system. Efficient query processing must be maintained and at the same time, all query operations must be implemented.

1.3 Objective

The objective of this research is to design a compression-based architecture that can be used for efficient querying of compressed relational database. This objective will be achieved in three steps:

- Defining of a number of operators for querying compression-based relational database system,
- Designing algorithms for these operators,
- Evaluating algorithms in terms of time and space.

The features of the proposed architecture are as follows:

- Records are directly addressable in compressed storage into memory or disk block.
- Reduced disk transfer time due to compression.
- Decompression overhead will be minimized as only the final result will be translated in uncompressed form.

1.4 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 disk-based 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, Disk Based HIBASE (DHIBASE). 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.

Chapter 4 gives the detailed analysis of query processing of DHIBASE system. SQL-like query operators in compressed format have been defined and algorithm of each operator and analysis of the algorithms are also provided.

Chapter 5 describes the experimental work that has been carried out. Results obtained are thoroughly discussed.

Chapter 6 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 [14]. Many research works [4, 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 [15]. The widely used statistical compression methods are Huffman [16] and Shannon-Fano [17,18] 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 [19, 20].

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 [21, 22, 23] 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 [24].

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 [25], delta encoding and dictionary encoding [5, 6, 26] 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) [27] is a modification of the original Lempel Ziv [21] dictionary coding algorithm. [21] 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:

- Decompression is simple and very fast.
- Requires no memory for decompression.
- 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. Compression methods e.g., Huffman [16], Lempel-Ziv [21, 22], LZW [23] etc cannot be used where efficient searching in the compressed data space is required. Cormack [3] has used a modified Huffman code [16] for the IBM IMS database system. Westmann et al. [4] has developed a lightweight compression method based on LZW [23] for relational databases. Moffat et al. [28] use the Run Length Encoding [25] 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. [29] 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 [30]. However Wee et al. [29] 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 [16] or LZW [23]. Dictionary-based compression methods, however, are used for any field containing a small number of different values.

2.2.2 HIBASE Architecture

The HIBASE architecture by Cockshot, McGregor and Wilson [11, 31] is a more radical approach to model the data representation.

In the relational approach, the database is a set of relations [13]. 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 2.1: CUSTOMER: a simple relation

Customer Name	Street	City	Status
Beauty	South	Gazipur	Married
Kalam	South	Dhaka	Single
Johan	North	Gazipur	Married
Anika	West	Gazipur	Married
Zoinal	South	Dhaka	Married
Belal	South	Gazipur	Married
Shamim	West	Dhaka	Married
Johan	West	Gazipur	Single

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 7 distinct customer names, only seven 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: Customer name, 2 bits: Street, 2 bits: City, 2 bits: Status) a total of 9 bits instead of the 25 bytes (6 bytes: Customer name, 5 bytes: Street, 7 bytes: City, 7 bytes: Status) 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). It may be noted that dictionary entries start with 1 not 0, this is because a 0 in a field will indicate a null or missing value.

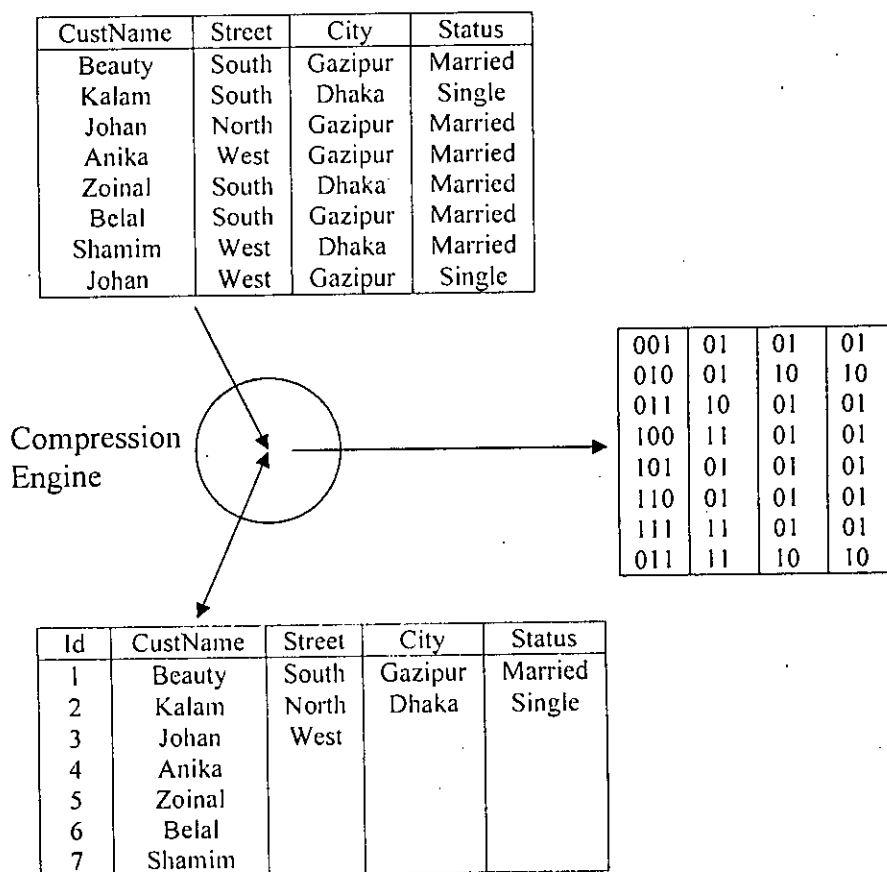


Fig. 2.1: Compression of CUSTOMER relation

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: $encode(lexeme) \rightarrow token$
2. It performs the reverse mapping from codes to literal values when parts of the relation are decompressed: $decode(token) \rightarrow lexeme$.
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.

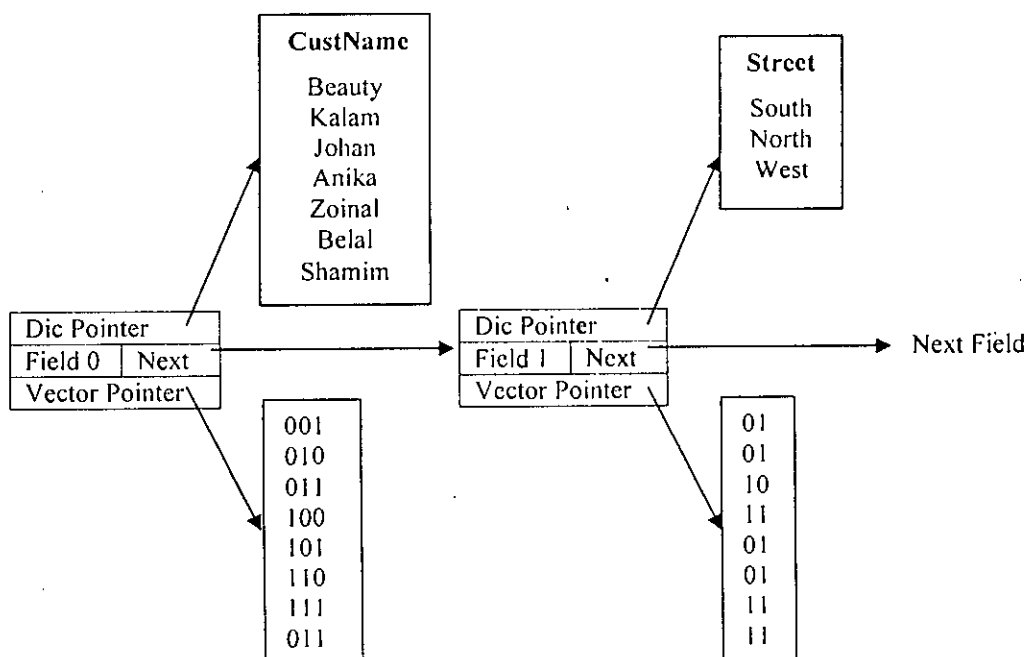


Fig. 2.2: Column-wise storage of a relation

2.2.3 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 [32] 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.

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 [33].

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

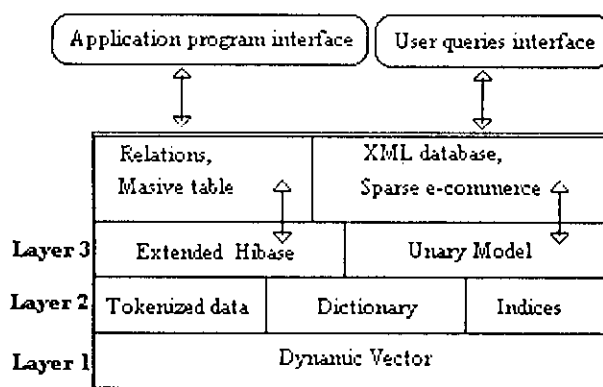


Fig. 2.3: The Three Layer Model

2.2.4 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 high-performance 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.

2.2.5 Compression in Oracle

The Oracle RDBMS recently 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 [34], 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 [35].

Query optimization: Much work has been done in query optimization. Access-path selection in the System R optimizer is described in [36], which was one of earliest relational query optimizers. Volcano, an equivalence-rule based query optimizer, is described in [37]. Query processing in Starburst is described in [38]. Query optimization in Oracle is briefly outlined in [39].

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 [40]. Optimization of nested sub-queries is discussed in [41].

Multi-query optimization, which is the problem of optimizing the execution of several queries as group, is described in [42]. 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 [43]. Optimization issues in pipelining with limited buffer space combined with sharing of common sub-expression are discussed in [44].

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 [45]; 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 [46], and extensions including hybrid hash join are described in [47]. 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 [48]. 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 [49].

Aggregation: An early work on relational algebra expressions with aggregate functions is found in [50]. More recent work in this area includes [51]. Optimization of queries containing outer joins is described in [52].

Views: A survey of materialized view maintenance is presented in [53]. Optimization of materialized view maintenance plans is described in [54]. Query optimization in the presence of materialized views is addressed in [55].

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. 2.4). 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.

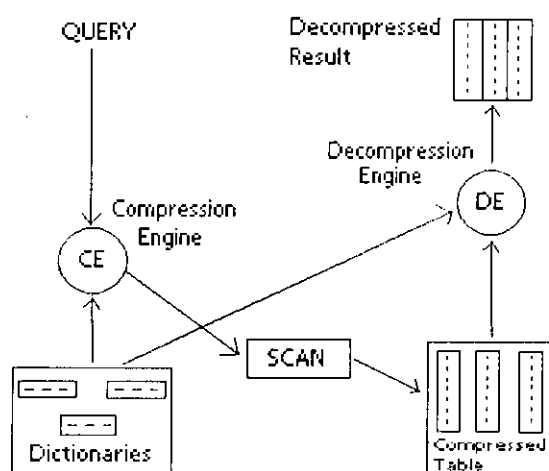


Fig. 2.4: Querying a database in compressed form

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.

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 because we taken this model as the basis of our architecture (which we call DHIBASE). But there are fundamental differences between HIBASE and DHIBASE. Differences are observable in storage structure of compressed data, number and types of query processing.

Chapter 3

DHIBASE Architecture and Compressed Query Processing

This chapter describes the details of the proposed DHIBASE architecture for storage of compressed relational data. The chapter also describes the details of the query processing techniques of the proposed system. We develop the system for single processor system. Query is evaluated directly on compressed data.

3.1 DHIBASE: Disk Based HIBASE Model

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

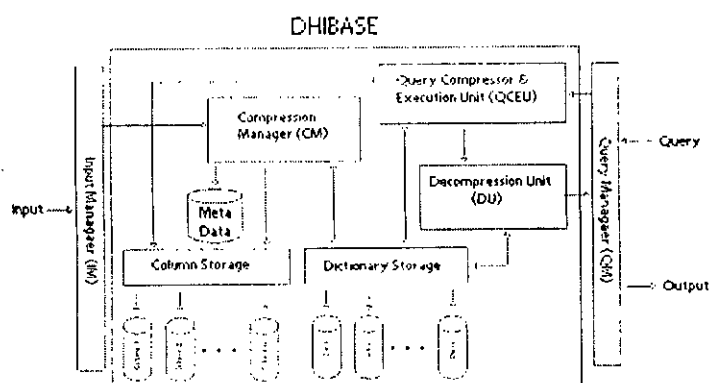


Fig. 3.1: DHIBASE Architecture

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 & 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 disk based 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 DHIBASE: Storage Complexity

$$S_{C_i} = n * C_i \text{ bits}$$

Where S_{C_i} = space needed to store column i in compressed form

n = no of records in the relation

C_i = no of bits needed to represent i^{th} attribute in compressed form

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

Total space to store all compressed columns, $S_{CC} = \sum_{i=1}^p S_{C_i}$ bytes; no of column is p

If we assume that domain dictionaries will occupy an additional 25% of S_{CC} , then

Total space to store the compressed relation, $S_{CR} = 1.25 S_{CC}$

Total space to store the uncompressed relation, $S_{UR} = \sum_{i=1}^p n \times x_i \times 8$ bits; x_i is the size (in bytes) of i^{th} attribute in uncompressed form.

Compression factor, $CF = \frac{S_{UR}}{S_{CR}} = \frac{p \times (n \times x \times 8)}{1.25 \times p \times n \times c} = \frac{x \times 8}{1.25c}$; assuming that all attributes have equal width in both compressed and uncompressed form.

3.3 DHIBASE: Insertion

Insertion of a value into a field of a record needs the search for compressed code of the given lexeme in the dictionary. If the lexeme is not in the dictionary then we have to add that lexeme to the dictionary which may require widening of the compressed column. Using hash structure, the search requires 2 seeks and 2 blocks transfers. As the last block of the dictionary is in memory, the insertion of the lexeme in the dictionary requires no seeks or block transfers. The complexities of widening operation are given bellow.

Let, B1 and B2 memory blocks are allocated to initial compressed column and widened compressed column respectively. Total block transfers are $b1 = \left\lceil \frac{n \lg m}{8b} \right\rceil$ and $b2 =$

$\left\lceil \frac{n (\lg m + 1)}{8b} \right\rceil$ respectively; n is the no of tuples in given relation, m is total no of initial

entries in corresponding dictionary, b is disk block size (in bytes) and we increase width

of the field by 1 bit whenever widening is necessary. Total no of seeks is $\frac{b1}{B1} + \frac{b2}{B2}$.

3.4 DHIBASE: Deletion

Deletion is performed by replacing the desired record with the last record and then reducing no of records by one. Dictionary entries are not deleted. Deletion from last memory block needs no seeks or no block transfers. Replacing the desired record by last one requires 2 seeks and 2 blocks transfer for each field.

3.5 DHIBASE: Update

If the new lexeme is not in the dictionary then we have to insert the new lexeme in the dictionary, which requires all operations of insertion as described in section 3.3. Updating of the desired field of the desired record requires 2 seeks and 2 blocks transfer.

3.6 Sorting of compressed relation according to code values

Sorting of table according to code values is required for some queries like natural join. As table is stored in column-wise format, an Auxiliary Column (AC) will be used to help the sorting process. AC has so many rows as the number of records in the table. Initially each row of AC will contain its own position in the column. That is, the i^{th} row of AC contains i . We first sort the column that contains the sorting attribute. Let us consider the following table which will be sorted by *CustName*.

Table 3.1: A relation to be sorted

CustName	Street	City	Status
Johan	North	Gazipur	Married
Kalam	South	Dhaka	Single
Anika	West	Gazipur	Married
Beauty	South	Gazipur	Married

The actual storage (based on dictionaries of Fig. 2.1) is given in Fig. 3.2.

CustName	Street	City	Status	AC
3	2	1	1	1
2	1	2	2	2
4	3	1	1	3
1	1	1	1	4

Fig. 3.2: Initial content of Auxiliary Column (AC)

During sorting of *CustName* column AC will be reordered so that after *CustName* column is sorted, i^{th} row of AC will contain the initial position of i^{th} row of *CustName* column (Fig. 3.3).

CustName	AC
1	4
2	2
3	1
4	3

Fig. 3.3: Content of AC after sorting of *CustName*

This AC will be used to sort other columns of the table. We make a copy of the desired data column. Now we scan the AC. From current row i of AC we find row no j of duplicated column that should be stored in row i of desired data column. Now we copy row j of duplicated column to row i of data column. When we finish scanning AC, the data column is sorted. Now duplicated column is deleted. In this way we sort other columns. When all other columns are sorted, AC will be deleted. The above sorting method is given in Algorithm 1.

Algorithm 1: Sorting according to code values

Sub SortByCode(byte t, byte c) //sorts t^{th} table by its c^{th} column

$n =$ no of records in table t

Read all disk blocks of column c of table t into $Data()$

For each record no i of table t , set $AC(i) = i$

call ShellSortCode($Data$, AC , n)

Write all blocks in $Data()$ to disk

For each column $c1$ other than c , call SortCol($Data$, AC , $c1$, n)

Delete $Data()$, $AC()$

End Sub

Sub SortCol($Data()$, $AC()$, c , n)

Read all disk blocks of column c in $Data()$

For each entry t in $AC()$, Write the t^{th} compressed record stored in $Data()$ to t^{th} location of $D()$

Write all blocks of $D()$ to disk

End Sub

Sub ShellSortCode($Data()$, $AC()$, n)

sorts all of n compressed records stored in $Data()$, whenever i^{th} and k^{th} compressed records are interchanged, the same record no's of $AC()$ are also interchanged.

End Sub

Let, the column containing the sorting attribute has B disk blocks and M memory blocks are available for sorting disk data. Using external merge sort total no of block transfers to sort the column is $B * (2 \lceil \log_{M-1}(B/M) \rceil + 1)$ and total no seeks is $2 * \lceil B/M \rceil + \lceil B/M \rceil (2 * \log_{M-1}(B/M) - 1)$. No of disk blocks to store the Auxiliary Column (AC) is $\lceil \frac{n \lg n}{8b} \rceil$; $n =$ no of records and $b =$ block size in bytes.

3.7 Sorting of compressed relation according to string values

The procedure for sorting a table according to string values is similar to that of sorting according to code values. The only difference is in the method of comparison of two values. Sorting by codes requires comparing only code values stored in the table and needs not checking the corresponding dictionary entries. But sorting by string sorts the table according to dictionary entries of the corresponding code values. This forces to look up strings in the dictionary and then compare those strings. If the strings are very long, the comparison time will be considerably long. To reduce the comparison time we create a dictionary (we call this *SortDic*, shown in Fig. 3.4) for each string domain that stores the sorting position of each code value of the dictionary.

Id	CustName	Sort Pos
1	Beauty	2
2	Kalam	5
3	Johan	4
4	Anika	1
5	Zoinal	7
6	Belal	3
7	Shamim	6

Fig. 3.4: Dic and SortDic of *CustName*

Let us consider the entries 101 (Zoinal) and 111 (Shamim) in *CustName* column. According to the *SortDic(CustName)*, $SortDic(101) = 7$ and $SortDic(111) = 6$. Therefore, '111(shamim)' < '101(Zoinal)'. So with *SortDic* we can avoid long string comparison and thus have faster sorting process. Of course, we have additional space to store *SortDic*. *SortDic* only contains integer values. So they themselves may be stored in compressed form. Hence a particular *SortDic* consumes a small amount of space and it may be completely kept into main memory to avoid disk access. Also *SortDic* may be created offline to reduce processing time. Algorithm II describes the technique of sorting by string.

Algorithm II: Sorting according to string values

Sub *SortByString*(byte *t*, byte *c*) //sorts t^{th} table by its c^{th} column

n = no of records in table *t*

Read all disk blocks of column *c* of table *t* into *Data*()

For each record no i of table t , set $AC(i) = i$
 call $ShellSortString(Data, AC, n)$
 Write all blocks in $Data()$ to disk
 For each column cl other than c , call $SortCol(Data, AC, cl, n)$
 Delete $Data(), AC()$

End Sub

Sub $ShellSortString(Data(), AC(), n)$

same as $ShellSortCode()$; but instead of comparing i^{th} and k^{th} compressed records in $Data()$, u^{th} and v^{th} entries of corresponding $SortDic$ are compared. Here u and v are integer values of i^{th} and k^{th} entries in $Data()$

End Sub

The $SortDic$ is also stored in compressed form. If the domain dictionary has m entries then the no of disk blocks to store $SortDic$, $N = \lceil \frac{m \lg m}{8b} \rceil$; b = block size in bytes. If we

assume that we have N memory blocks to store the entire $SortDic$ then all estimates are same as in Algorithm I.

3.8 Joining of compressed relations

In DHIBASE architecture, columns of same domain share the same single dictionary. Therefore, the same data has same compressed code in all positions in all tables. So we can join two tables based on compressed codes without checking the exact 'string' values. Algorithm III describes the natural join of two tables.

Algorithm III: Natural join of $T1$ and $T2$ based on attribute A

Sub $NaturalJoin(T1, C1, T2, C2)$ // $T2$ has total records less than that
 //of $T1$. $T1$ has column $C1$ and $T2$ has column $C2$ for attribute A
 Sort $T1$ according to code values of column $C1$
 Sort $T2$ according to code values of column $C2$
 Read all disk blocks of column $C1$ of $T1$ into $D1()$
 Read all disk blocks of column $C2$ of $T2$ into $D2()$
 Set l to both m and j
 For each record no k of table $T1$ {
 While k^{th} entry of $D1() = j^{th}$ entry of $D2()$ And $j \leq NoOfRecords(T2)$ {
 Set k^{th} entry of $D1()$ to m^{th} position of $D()$

Set k to mth position of AC1()

Set j to mth position of AC2()

Increment m and j

Write all blocks of D() to disk

For each column k of T1 other than C1, Call Adjust(T1, k, AC1, D1())

For each column k of T2 other than C2, Call Adjust(T2, k, AC2, D2())

Delete D(), D1(), D2(), AC1(), AC2()

End Sub

Sub Adjust(T, C, AC(), Data())

Read all disk blocks of column C of T into Data()

For each kth entry t of AC(), set tth entry of Data() to kth position of D()

Write all blocks of D() to disk

End Sub

We assume that relations T1 and T2 are already sorted. T1 has B_1 disk blocks and has n_1 tuples and the no's for T2 are B_2 and n_2 respectively. M_1 and M_2 are the no of memory blocks allocated to store disk blocks of T1 and T2 respectively. Total no of disk block transfers is $B_1 + B_2$. If, on average, p_1 tuples of T1 are matched to p_2 tuples of T2 then total disk seeks is $2 * \max(\lceil n_1 / (M_1 * p_1) \rceil, \lceil n_2 / (M_2 * p_2) \rceil)$. No of disk blocks to store

$AC1()$ and $AC2()$ are $\lceil \frac{n_1 \lg n_1}{8b} \rceil$ and $\lceil \frac{n_2 \lg n_2}{8b} \rceil$ respectively where b is block

size in bytes.

3.9 SQL queries on compressed data

This section describes query processing on compressed data. Section 3.9.1 describes query processing on single compressed relation, section 3.9.2 describes query processing on multiple compressed relations and section 3.9.3 describes aggregation functions on compressed table.

3.9.1 Queries on single compressed relation

Queries on single relation are generally projection, selection and aggregation operations. Following two sub-sections describe projection and selection in details. Aggregation is described in section 3.9.3.

3.9.1.1 Projection

Select C From T. The algorithm is given below:

Algorithm IV: Projection of single column

Read all disk blocks of Dic(C) into D()

For each disk block b of column c {

 For each record r in b {

 Print rth entry of D()

Delete D()

$$\text{no of seek} = 1, \text{ no of disk block read} = \left\lceil \frac{n \lg m}{8b} \right\rceil$$

here n = no of records and m = no of entries in the domain dictionary.

3.9.1.2 Selection with single predicate

Select C From T Where C_j = 'xx'. The algorithm is given below:

Algorithm V: Single predicate selection

Part-1:

v = code for 'xx' in domain Dic(C_j)

Set i = 1

For each disk block b of column C_j {

 For each record r = v in block b {

 Set r to ith position of AC()

 Increment i

Part-2:

Read all disk blocks of Dic(C) into D()

For each each entry t in AC(), Print tth entry of D()

Using hash structure to search domain dictionary we can find the code for a given 'string' in 2 disk seeks and 2 disk-block reads. To select tuples of selection column we need 1 disk seek and $\left\lceil \frac{n \lg m}{8b} \right\rceil$ blocks transfer. The same no is needed for each projected

column. No of disk blocks to store AC() is $\left\lceil \frac{n \lg n}{8b} \right\rceil$.

3.9.1.3 Selection with multiple predicates

Select C From T Where $C_j = 'xx'$ And $C_k = 'yy'$. The algorithm is given below:

Algorithm VI: Multiple-predicate selection

Part-1 of Algorithm V

$u =$ code for 'yy' in domain $Dic(C_k)$

For each entry t of $AC()$, If t^{th} record of column C_k is not u , remove t from $AC()$

Part-2 of Algorithm V

Selection with q predicates: to search code in domain dictionaries we need $q * 2$ disk seeks and $q * 2$ blocks read. To create $AC()$ another $q * 1$ seeks and $q * \left\lceil \frac{n \lg m}{8b} \right\rceil$

blocks read are needed. $AC()$ needs $\left\lceil \frac{n \lg n}{8b} \right\rceil$ blocks for storage.

3.9.1.4 Selection with range predicate

Select C From T Where $C \geq 'xx'$ And $C \leq 'yy'$. The algorithm is given below:

Algorithm VII: Range-predicate selection

$k =$ code for 'xx' in $Dic(C)$

$j =$ code for 'yy' in $Dic(C)$

$t1 = k^{\text{th}}$ entry of $SortDic(C)$

$t2 = j^{\text{th}}$ entry of $SortDic(C)$

Now execute the query:

Select C From T Where $SortDic(C) \geq t1$ And $SortDic(C) \leq t2$

$SortDic(C)$ is stored in compressed form. We assume that the total $SortDic(C)$ is in memory which consumes $\left\lceil \frac{m \lg m}{8b} \right\rceil$ memory blocks where m is the no of entries in

$Dic(C)$. In worst case when the edge values of the given range are not in $Dic(C)$, we have to search the entire dictionary which requires 1 seek and D blocks transfer ($Dic(C)$ has D

blocks). The modified query further needs 1 seek and a total of $\lceil \frac{n \lg m}{8b} \rceil$ blocks transfer where n is the total no of records in table T .

3.9.2 Queries on multiple compressed relations

Queries on multiple relations are set operations and queries based on the join of two or more relations. Natural join has already been described in section 3.8. The following subsections describe different queries on multiple relations.

3.9.2.1 Projection

Select T1.C From T1, T2. The algorithm is given below:

T = NaturalJoin(T1, T2)

Select C' From T

3.9.2.2 Selection with single predicate

Select T1.C From T1, T2 Where T1.C_j = 'xx'. The algorithm is given below:

T = NaturalJoin(T1, T2)

Select C' From T Where C'_j = 'xx'

3.9.2.3 Selection with multiple predicates

Select T1.C From T1, T2 Where T1.C_j = 'xx' And T2.C_k = 'yy'. The algorithm is given below:

T = NaturalJoin(T1, T2)

Select C' From T Where C'_j = 'xx' And C'_k = 'yy'

3.9.2.4 Set Union

The SQL operation **union** operates on relations and corresponds to relational-algebra operation \cup . The relations participating the **union** operation must be compatible, that is, they must have the same set of attributes. The **union** operation automatically eliminates duplicates.

(Select C1 From T1) Union (Select C2 From T2). The algorithm is given below:

Algorithm VIII: Set union

Read all disk blocks of C1 of T1 into D1(), Read all disk blocks of C2 of T2 into D2()

Output will be stored into D()

While both D1() and D2() have more elements {

t = next value from D1()

while D1() has more values and t = next value of D1() {

move to next value of D1()

While D2() has more values and t = next value of D2() {

move to next value of D2()

insert t into D()

If D1() is finished then {

Insert distinct values remaining in D2() into D()

Else

Insert distinct values remaining in D1() into D()

Write all blocks of D() to disk

Let, B1, B2 and B memory blocks are allocated to C1, C2 and output column respectively. Total block transfers are $b1 = \left\lceil \frac{n_1 \lg m}{8b} \right\rceil$, $b2 = \left\lceil \frac{n_2 \lg m}{8b} \right\rceil$ and $b = \left\lceil \frac{n \lg m}{8b} \right\rceil$

respectively. n_1 , n_2 and n are the no of tuples in T1, T2 and output respectively, and m is total no of entries in corresponding dictionary. Total no of seeks is $\frac{b1}{B1} + \frac{b2}{B2} + \frac{b}{B}$.

3.9.2.5 Set Intersection

The **intersect** operation, like **union** operation, also operates on relations. The operation corresponds to relational-algebra operation \cap . The relations participating the **intersect** operation must the same set of attributes. The **intersect** operation automatically eliminates duplicates.

(Select C1 From T1) Intersection (Select C2 From T2). The algorithm is given below:

Algorithm IX: Set intersection

Read all disk blocks of $C1$ of $T1$ into $D1()$. Read all disk blocks of $C2$ of $T2$ into $D2()$

Output will be stored into $D()$

While both $D1()$ and $D2()$ have more elements {

$t =$ next value from $D1()$

 while $D1()$ has more values and $t =$ next value of $D1()$ {

 move to next value of $D1()$

 counter = 0

 While $D2()$ has more values and $t =$ next value of $D2()$ {

 move to next value of $D2()$ and increment counter

 If counter $\neq 0$ then insert t into $D()$

Write all blocks of $D()$ to disk

Complexities are same that shown in section 3.9.2.4 (Set Union).

3.9.2.6 Set Difference

The **except** operation, like **union** and **intersect** operations, also operates on relations. The operation corresponds to set difference operation $-$. The relations participating the **except** operation must the same set of attributes. The **except** operation automatically discards duplicates.

(Select $C1$ From $T1$) Except (Select $C2$ From $T2$). The algorithm is given below:

Algorithm X: Set difference

Read all disk blocks of $C1$ of $T1$ into $D1()$. Read all disk blocks of $C2$ of $T2$ into $D2()$

Output will be stored into $D()$

While both $D1()$ and $D2()$ have more elements {

$t =$ next value from $D1()$

 while $D1()$ has more values and $t =$ next value of $D1()$ {

 move to next value of $D1()$

 counter = 0

 While $D2()$ has more values and $t =$ next value of $D2()$ {

 move to next value of $D2()$ and increment counter

 If counter = 0 then insert t into $D()$

If $D1()$ is not finished then {

If M blocks are allocated to each of D(), D1() and D2() then no of block transfer is

$\lceil \frac{n \lg m}{8b} \rceil$ for D() and $\lceil \frac{(n/\bar{n}) \lg m}{8b} \rceil$ for each of D1() and D2(). No of seeks is 3 *

$\lceil \frac{n \lg m}{8Mb} \rceil$. Where \bar{n} is the average size of each group.

3.9.3.2 Max/ Min

select branch_name, max(balace) from account group by branch_name. The algorithm is given below:

Algorithm XII: Max/ Min aggregation function

Sort account by branch_name

Read all disk blocks of branch_name column into D()

Part-1 of algorithm given in section 3.9.3.1

Read all disk blocks of column balance into D()

base = 0

For each record i of D2() {

max = maximum of (base+1)th to (base+i)th element of D()

set D2(i) = max

Increment base by i

Write all blocks of D1() and D2() to disk

Complexities are double that shown in section 3.9.3.1 (Count).

3.9.3.3 Sum/ Avg

select branch_name, sum(balace) from account group by branch_name. The algorithm is given below:

Algorithm XIII: Sum/ Avg aggregation function

Sort account by branch_name

Read all disk blocks of column branch_name into D()

Part-1 of algorithm given in section 3.9.3.1

Read all disk blocks of column balance into D()

```

base = 0
For each record i of D2() {
    max = sum of (base+1)th to (base+i)th element of D()
    // in case of avg make average of these elements
    set D2(i) = max
    Increment base by i
}
Write all blocks of D1() and D2() to disk

```

Complexities are same that shown in section 3.9.3.2 (Max).

3.10 Summary

In this chapter we have presented an attractive compression-based architecture, called DHIBASE. Due to disk based compression DHIBASE 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. Moreover, we have designed a sorting algorithm of compressed relation stored in column wise format which is perhaps new. Algorithms of query operators given in this chapter have been thoroughly analyzed.

Chapter 4

Results and Discussion

The objective of the experimental work is to verify the applicability and feasibility of the proposed DHIBASE architecture. The experimental evaluation has been performed with large synthetic data. The experimental result is compared with widely used Microsoft SQL Server 2000. Our target was to handle large relations and justify the storage requirements and query time in comparison with SQL Server.

4.1 Experimental Environment

DHIBASE has been tested on a machine with 1.73 GHz Pentium IV processor and 256 MB of RAM, running on Microsoft Windows XP. We have created 5 different relations as given in Table 4.1 to Table 4.5. A random data generator has been used to generate synthetic data and large no of records have been inserted into each table. Each query has been executed 5 times and the average execution time has been taken.

Table 4.1: **Item** relation

Attribute Name	Cardinality	Uncompressed field length (byte)	Compressed field length (bit)
Item_id (Primary key)	1000	8	10
Type	7	6	3
Description	100	20	7
Total		34 bytes	20 bits (CF = 13.6)

Table 4.2: **Employee** relation

Attribute Name	Cardinality	Uncompressed field length (byte)	Compressed field length (bit)
Employee_id (Primary key)	2000	10	11
Name	400	20	9
Department	15	8	4
Total		38 bytes	24 bits (CF = 12.67)

Table 4.3: **Store** relation

Attribute Name	Cardinality	Uncompressed field length (byte)	Compressed field length (bit)
Store_id (Primary key)	100	6	7
Location	10	10	4
Type	6	6	3
Total		22 bytes	14 bits (CF = 12.57)

Table 4.4: **Customer** relation

Attribute Name	Cardinality	Uncompressed field length (byte)	Compressed field length (bit)
Customer_id (Primary key)	10000	10	14
Name	2000	20	11
City	30	12	5
District	14	12	4
Total		54 bytes	34 bits (CF = 12.71)

Table 4.5: **Sales** relation

Attribute Name	Cardinality	Uncompressed field length (byte)	Compressed field length (bit)
Sales_id (Primary key)	Upto 1 million (Numeric)	4	32 (Uncompressed)
Employee_id (Foreign key)	2000	10	11
Customer_id (Foreign key)	10000	10	14
Item_id (Foreign key)	1000	8	10
Store_id (Foreign key)	100	6	7
Quantity	Numeric	2	16 (Uncompressed)
Price	Numeric	4	32 (Uncompressed)
Total		44 bytes	122 bits (CF = 2.89)

4.2 Storage Requirement

Our data generator has generated 1000, 2000, 100, 10000 records for *Item*, *Employee*, *Store* and *Customer* relations respectively and these records remained fixed during experiments. 0.1, 0.4, 0.7 and 1 million records were stored in *Sales* relation for each experiment. Compression factor (CF) of each relation is given in table 4.1 to 4.5. Table 4.6 shows overall CF's for different number of records in *Sales* relation. CF's are calculated with respect to Microsoft SQL Server 2000. We observe that the proposed system outperforms SQL Server by a factor of 10 to 20. It was expected that with more records in *Sales* relation the CF would increase; but we found opposite result. The reason is that we have used some uncompressed attribute in the *Sales* relation (uncompressed 80 bits out of total 122 bits) to speed up query processing and due to these uncompressed attributes the overall CF decreases with the increase in the no of records in *Sales* relation. CF = 10 is still a great achievement. If someone wants higher compression factor then he has to make performance degradation in query processing.

Table 4.6: Compression Factor (CF) achieved

No of tuples in <i>Sales</i> relation	Overall CF
0.1 million	20.3
0.4 million	17.7
0.7 million	12.4
1.0 million	10.1

Fig. 4.1 shows a graphical comparison between DHIBASE and SQL Server storage requirements

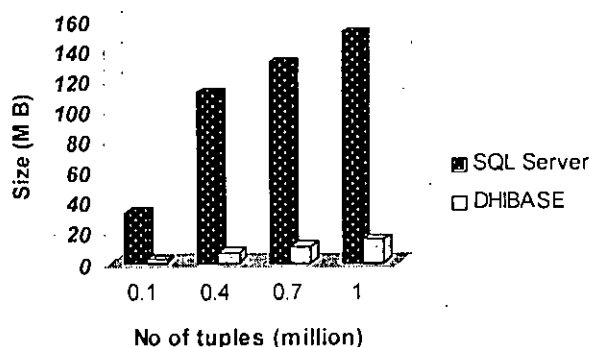


Fig. 4.1: Storage comparison between DHIBASE and SQL Server

4.3 Query Performance

To assess query performance, we carried out queries on both DHIBASE and SQL Server. The performed queries and obtained results are described in the following sub-sections. In all cases *Sales* relation contains 0.1, 0.4, 0.7 and 1 million records. *Item*, *Employee*, *Store* and *Customer* relations contain 1000, 2000, 100, 10000 records respectively. All queries executed in DHIBASE 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 4.2.

select Customer_id from Sales

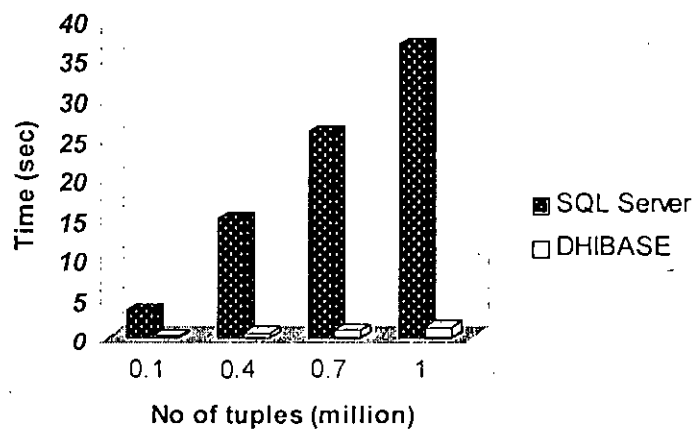


Fig. 4.2: Single column projection

Figure 4.2 shows that DHIBASE is much faster than that of SQL Server in case of projection operation. This is obvious because DHIBASE stores data in compressed form and in column wise format. Therefore, it needs 14 bits data to process one record. But SQL Server stores data in row wise format and it needs $(44 * 8)$ -bit data to process one record. So, in case of 0.1 million records, DHIBASE examines 43 disk blocks (block size = 4K) where SQL Server has to examine a minimum of 1075 disk blocks. This is the main reason of speed-gain in DHIBASE system.

4.3.2 Single Predicate Selection

We have executed the following query and the result is shown in figure 4.3.

```
select Customer_id from Sales where Store_id = "100075"
```

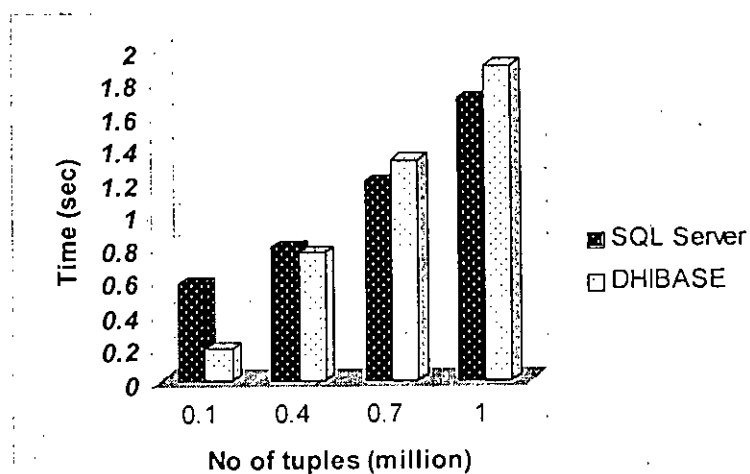


Fig. 4.3: Single predicate selection

Figure 4.3 shows that DHIBASE is faster than SQL Server in case 0.1 million to 0.4 million records but slower in case 0.7 million and 1.0 million records. DHIBASE does not use any indices. In case of 1 million records, it reads all 214 disk blocks to check the *Store_id* column for making a list of desired row no's. If 100 records satisfy the selection predicate and each of them resides in different disk block then another 100 disk blocks read is necessary to project the *Customer_id*'s. But SQL Server uses index on *Store_id* field. So it reads 1 disk block to find pointers to disk blocks containing each of 100 desired records, and needs another 100 disk blocks to read the desired records. So total no of disk block read in SQL Server is 101, which was 314 in DHIBASE.

4.3.3 Double Predicate Selection

We have executed the following query and the result is shown in figure 4.4.

```
select Customer_id from Sales where Store_id = '100075' and Item_id =  
"10000300"
```

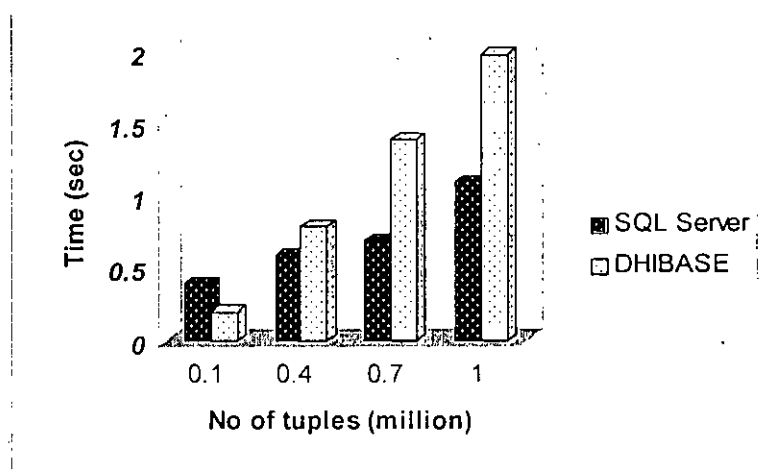


Fig. 4.4: Double predicate selection

Figure 4.4 shows result similar to the result of single predicate selection. DHIBASE does not use any indices. In case of 1 million records, it reads all 214 disk blocks to check the *Store_id* column for making an intermediate list of desired row no's. If 100 records satisfy the first predicate and each of them resides in different disk block then another 100 disk blocks read is necessary to check the *Item_id* column for making the final list of desired row no's. If 10 records satisfy the second predicate and each of them resides in different disk block then another 10 disk blocks read is necessary to project the *Customer_id*'s. But SQL Server uses indices on *Store_id* and *Item_id* fields. So it reads 1 disk block to find pointers to disk blocks containing each of 100 desired records containing *Store_id*, also reads 1 disk block to find pointers to disk blocks containing each of 10 desired records containing *Item_id* and needs another 10 disk blocks to read the desired records. So total no of disk blocks read in SQL Server is 12, which was 324 in DHIBASE. 12 disk blocks read in SQL Server is theoretically minimum. But the actual index structure is not known. And SQL Server uses storage optimization Therefore, the actual no is higher than 12.

4.3.4 Range Predicate Selection

We have executed the following query and the result is shown in figure 4.5.

```
select Customer_id from Sales where Store_id >= "100091" and Store_id <=
- "100100"
```

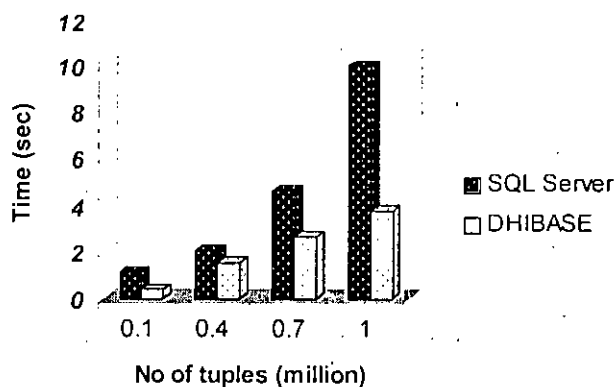



Fig. 4.5: Range predicate selection

Figure 4.5 shows that DHIBASE is much faster than SQL Server. DHIBASE does not use any indices. In case of 0.1 million records, it reads all 22 disk blocks to check the *Store_id* column for making a list of desired row no's. Range predicate checking involves access to *SortDic()*. As *SortDic()* is totally in memory, access to it does not require extra disk block transfer. The records satisfying the selection predicate may scatter across all disk blocks of *Customer_id* column. So another 43 disk blocks read is necessary to project the *Customer_id*'s. But SQL Server, in worst case, may require all 1075 disk blocks to process the scattered *Customer_id*'s.

4.3.5 Sorting of Relation

Figure 4.6 shows sorting time needed to sort *Sales* relation with 0.1, 0.4, 0.7 and 1.0 million records.

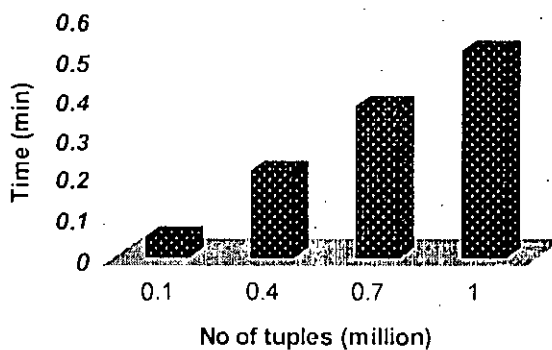


Fig. 4.6: Sorting of *Sales* selection

When we sort a relation based on dictionary code we do not need to refer to the dictionary during sorting. But when we sort a relation based on string order we have to refer to the dictionary. But we avoid this dictionary access by using *SortDic()*. The technique is described in section 3.7 in details. As the *SortDic()* is also compressed and requires small amount of memory, it can be kept into memory during sorting. So the time shown in Fig. 4 indicates that the time needed for sorting based on dictionary code and the time needed for sorting based on string order are same.

4.3.6 Aggregation: Count

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

```
select Store_id, count (Store_id) from Sales group by Store_id
```

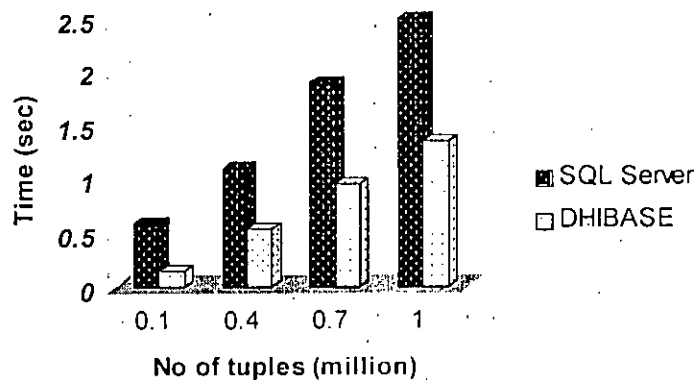


Fig. 4.7: Aggregation: Count

DHIBASE does not use any indices. We assume that the relation *Sales* is already sorted by *Store_id* field according to dictionary code. In case of 0.1 million records, DHIBASE reads all 22 disk blocks of *Store_id* column to calculate the result. SQL Server uses indices but still it has to read all 1075 disk blocks to process the entire *Sales* relation. Therefore, DHIBASE performs better than SQL Server.

4.3.7 Aggregation: Max/ Min/ Sum/ Avg

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

```
select Store_id, max (Quantity) from Sales group by Store_id
```

```
select Store_id, sum (Price) from Sales group by Store_id
```

```
select Store_id, avg (Price) from Sales group by Store_id
```

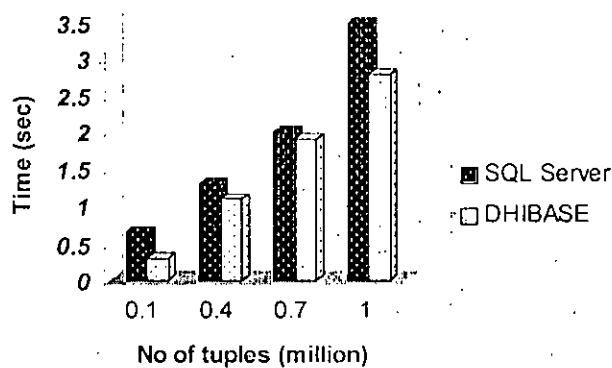


Fig. 4.8: Aggregation: Max/ Min/ Sum/ Avg

We assume that the relation *Sales* is already sorted by *Store_id* field according to dictionary code. In *Sales* relation, both *Quantity* and *Price* fields are uncompressed. So aggregation functions may directly be applied on them. In case of 0.1 million records, DHIBASE reads all 22 disk blocks to examine *Store_id* field for making group information. Using this group information it reads 49 disk blocks (*Quantity*) or 98 disk blocks (*Price*) to compute the result. Therefore, time needed for max, min, sum or avg is same. But SQL Server has to read 1075 disk blocks to process the entire *Sales* relation. Therefore, DHIBASE performs better than SQL Server.

4.3.8 Natural Join

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

```
select Sales_id from Sales, Customer where Sales.Customer_id =
Customer.Customer_id
```

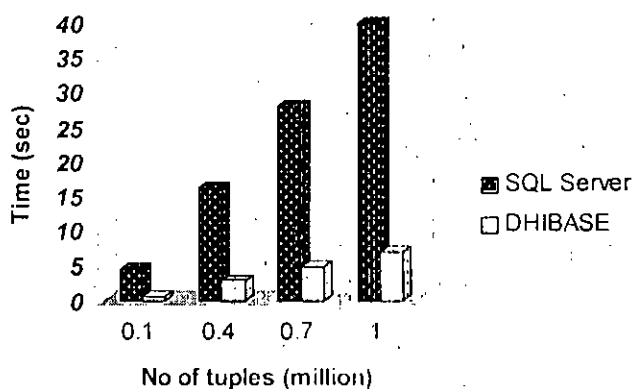


Fig. 4.9: Natural join

DHIBASE performs much better because it is possible to calculate the partial join and project the result. DHIBASE does not use any indices. We assume that the relations *Sales* and *Customer* are already sorted by *Customer_id* field according to dictionary code. We have to scan *Customer_id* columns of both relations to make *AC()*'s (section 3.8). In case of 0.1 million records, DHIBASE reads 5 disk blocks to check *Customer_id* field of *Customer* relation and 43 disk blocks to check *Customer_id* field of *Sales* relation. If the result is scattered across all disk blocks then another 98 disk blocks read is necessary to project *Sales_id* field of *Sales* relation. But SQL Server needs to read 134 disk blocks of *Customer* relation and 1075 disk blocks of *Sales* relation to compute the join. So SQL Server reads a total of 1209 disk blocks read while DHIBASE read only 141 disk blocks. This is why DHIBASE is much faster.

4.3.9 Set Union

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

(select Customer_id from Customer) union (select Customer_id from Sales)

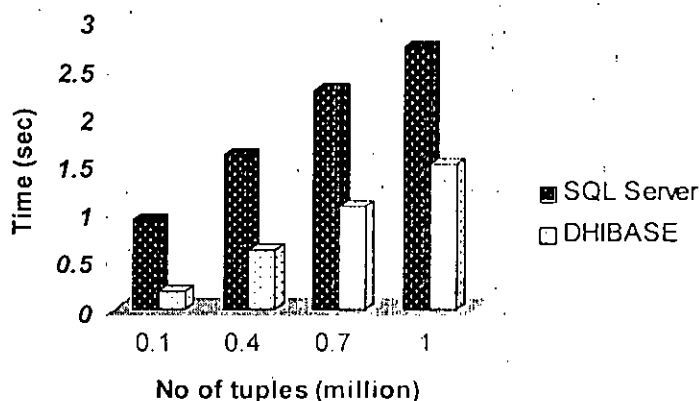


Fig. 4.10: Set union

DHIBASE performs better because it is possible to calculate the result only by scanning *Customer_id* fields of *Customer* and *Sales* relations. DHIBASE does not use any indices. We assume that the *Customer_id* fields of relations *Sales* and *Customer* are already sorted according to dictionary code. DHIBASE reads 5 disk blocks and 43 disk blocks for the relations. But SQL Server reads minimum 134 disk blocks and 1075 disk blocks

respectively. So SQL Server reads a minimum of 1209 disk blocks read while DHIBASE read only 48 disk blocks. This is why DHIBASE is much faster.

4.3.10 Set Intersection/ Set Difference

We have executed the following queries on DHIBASE:

```
(select Customer_id from Customer) intersect (select Customer_id from Sales)
(select Customer_id from Customer) except (select Customer_id from Sales)
```

But SQL Server does not support *intersect* and *except* operator directly. So we have executed the following queries on SQL Server.

```
select distinct Customer_id from Customer where Customer_id in (select
Customer_id from Sales)
select distinct Customer_id from Customer where Customer_id not in (select
Customer_id from Sales)
```

In the worst case, both DHIBASE and SQL Server have to examine the entire *Customer_id* columns of relations of *Customer* and *Sales* for both queries, and therefore, time requirements will be similar. Time requirements are shown in figure 4.11.

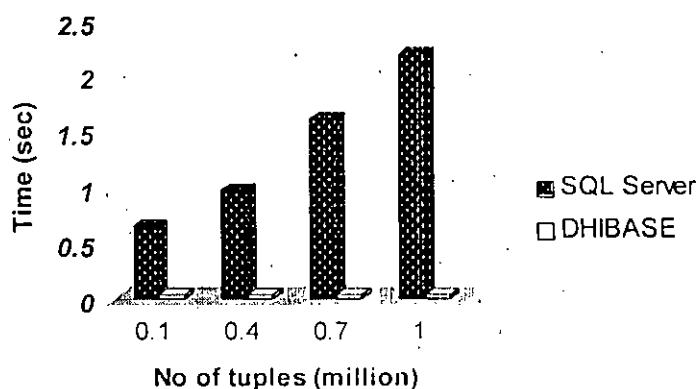


Fig. 4.11: Set intersection/ Set difference

DHIBASE performs much better because it is possible to calculate the result only by scanning *Customer_id* fields of *Customer* and *Sales* relations. DHIBASE does not use any indices. We assume that the *Customer_id* fields of relations *Sales* and *Customer* are already sorted according to dictionary code. DHIBASE reads 5 disk blocks and 43 disk

blocks for the relations. But SQL Server reads 134 disk blocks and 1075 disk blocks respectively. So SQL Server reads a total of 1209 disk blocks read while DHIBASE read only 48 disk blocks. This is why DHIBASE is much faster. It is mentionable that the result size of four different data sets is same. In case of intersection, DHIBASE reads the smaller relation first. As soon as, this smaller relation finished scanning DHIBASE stops and thus saves time necessary to scan the rest of the other relation.

4.3.11 Bulk Update

We have created a table with three fields. We have inserted data into first and third fields but retain nulls in the second field. Then we updated the nulls in the second field with values. The time needed for this bulk update of the second column is compared for DHIBASE and SQL Server in figure 4.12.

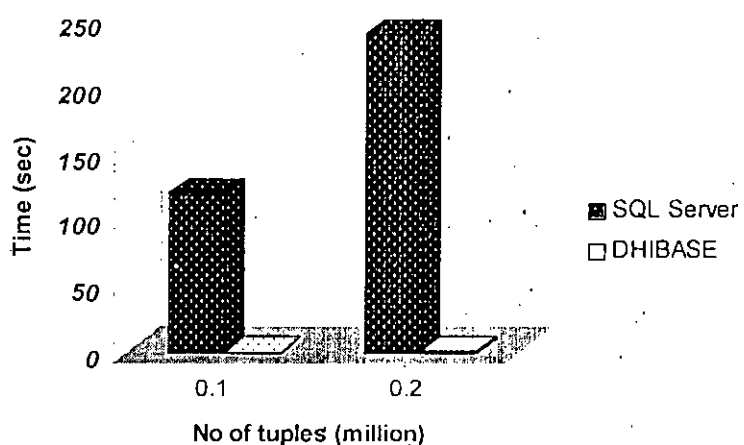


Fig. 4.12: Bulk update

DHIBASE performs exceptionally better than SQL Server. DHIBASE stores data in column wise format. Therefore, column reorganization does not incur extra overhead. But SQL Server stores data in row wise format and also it optimizes storage for faster query processing. SQL Server optimizes the nulls in the second column for fast indexing. When the update commands come into the second column, it has to reorganize all data previously stored in first and third columns, which needs huge time.

4.4 Summary

In this chapter we have presented the experimental evaluation of the DHIBASE architecture. We evaluated the storage performance in comparison with Microsoft SQL Server 2000, a widely used database system. The storage performance that is achieved in DHIBASE is 10 to 20 times better than that of SQL Server. The projection query shows a great speed-gain compared with SQL Server. The experimental result shows that DHIBASE is 22 to 30 times faster than SQL Server in single column projection. In case of selection queries DHIBASE is slightly slower than SQL Server. This is because of the absence of indices in DHIBASE. But all other queries can run significantly faster in DHIBASE than in SQL Server. Finally, in case of bulk update, DHIBASE exceptionally outperforms SQL Server.

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 [9] for disk support. We 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 disk based HIBASE (DHIBASE) architecture with increased query processing capability.
- ❖ Compressed data are stored using the DHIBASE architecture with disk support. This overcomes the scalability problems of the memory resident DBMS.
- ❖ Considerable storage reduction is achieved using the DHIBASE architecture. The experimental results show that DHIBASE architecture is 10 to 20 times space efficient than that of conventional DBMS like SQL Server.
- ❖ Each compressed column is kept in separate disk file. Consequently schema evolution and/ or bulk update is highly efficient.

- ❖ We developed techniques for sorting of compressed relation according to both code order and string order. We introduced a compressed auxiliary dictionary which facilitates sorting according to string order without accessing the relevant dictionary. Using same compressed dictionary we have implemented selection operation with range predicate without accessing the main dictionary.

- ❖ We have designed algorithms for all relational algebra operations that support most of the SQL:2003 standard. Experimental results show that the DHIBASE system has significantly faster query performance for projection on single relation, multiple relation join, set and aggregation operations compared to SQL Server. In case of selection operation, DHIBASE is slower. But introducing indices can easily eliminate this drawback.

5.2 Future Research

The DHIBASE architecture has been implemented in a single processor system and achieved significant performance improvement over conventional DBMS. DHIBASE is a 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 DHIBASE architecture. These features may be included. We did not use any index. Techniques should be considered for indexing.

- ❖ To achieve concurrent access to DHIBASE architecture, a multi-threaded algorithm can be considered to support multi-user DBMS.

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 Van Horn, S. J., "Database compression", SIGMOD Record, Vol-22, 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 29th 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] Latiful Hoque, A. S. M., "Compression of structured and semi-structured information", Ph. D. Thesis, Department of Computer and Information Science, University of Strathclyde, Glasgow, UK, 2003.
- [13] 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.
- [14] McGregor, D. R. and Hoque, A. S. M. L., "Improved compressed data representation for computational intelligence systems". In UKCI-01, Edinburgh, UK, September 2001.
- [15] Held, G. and Marshel, T. R. , "Data and Image Compression", Number 0-471-95247-8, John Wiley and Sons Ltd, West Sussex, England, 1996.
- [16] Huffman, D.A., "A method for the construction of minimum-redundancy code", In *Proceedings of IRE*, Vol-40, No. 9, pp 1098–1101, 1952.
- [17] Fano, R.M., "The transmission of information", In *Research Laboratory for Electronics, MIT Technical Report*, (65), 1949.
- [18] Shannon, C.E., "A mathematical theory of communications", In *Bell system Technical Journal*, Vol-27, pp 379–423 and 623–656, 1948.
- [19] Vitter, J.S., "Design and analysis of dynamic Huffman code", In *Journal of the ACM*, Vol-34, No. 4, pp 825–845, October 1987.
- [20] Gallager, R.G., "Variations on a theme by Huffman", In *IEEE Transaction on Information Theory*, Vol-24, No. 6, pp 668 -674, November 1978.
- [21] Lampel, A. and Ziv, J., "A universal algorithm for sequential data compression", In *IEEE Transaction on Information Theory*, Vol-23, pp 337–343, 1977.
- [22] 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.
- [23] Welch, T. A., "A technique for high-performance data compression", In *IEEE Computer*, Vol-17, No. 6, pp 8–19, 1984.
- [24] Larson, N. J. and Moffat, A., "Off-line dictionary-based compression", In *Proceedings of the IEEE Data Compression Conference*, Snowbird, Utah, March 2000.
- [25] Golomb, S. W., "Run-length encodings". In *IEEE Transaction on Information Theory*, Vol-12, No. 3, pp 399-401, 1966.

- [26] 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.
- [27] Oberhummer, M. F. X. J., "LZO: A real-time data compression library", 2002, <http://www.oberhumer.com/opensource/lzo/lzodoc.php>.
- [28] Moffat, A. and Zobel, J., "Parameterised compression for sparse bitmap", In Proceedings of the 15th Annual International SIGIR 92, pp 274–285, ACM, 1992.
- [29] 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.
- [30] 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.
- [31] 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.
- [32] Xu, Y., Agrawal, R. and Somani, A., "Storage and querying of e-commerce data", In Proceedings of the 27th VLDB Conference, pp 149–158, Roma, Italy, 2001.
- [33] 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.
- [34] Aho, A. V., Sethi, R. and Ullman, J. D., "Compilers: Principles, Techniques, and Tools", Addison Wesley, 1986.
- [35] Knuth, D. E., "The art of computer programming", Vol-3, Addison Wesley, Sorting and Searching, 1973.
- [36] 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.
- [37] Graefe, G. and McKenna, W., "The Volcano optimizer generator", Proc. of the International Conf. on Data Engineering, pp 209-218, 1993.
- [38] 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.
- [39] Oracle 8 concepts manual, Oracle Corporation, Redwood Shores, 1997.

- [40] 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.
- [41] Seshadri, P., Pirahesh, H. and Lueng, T. Y. C., "Complex query decorrelation", Proc. of the International Conf. on Data Engineering, pp 450-458, 1996.
- [42] Roy, P., Seshadri, S., Sudarshan, S. and Bhojhe, S., "Efficient and extensible algorithms for multi-query optimization", Proc. of the ACM SIGMOD Conf. on Management of Data, 2000.
- [43] 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.
- [44] 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.
- [45] 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.
- [46] 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.
- [47] 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.
- [48] Davison, D. L. and Graefe, G., "Memory-contention responsive hash joins", Proceedings of VLDB Conference, 1994.
- [49] Graefe, G., Bunker, R. and Cooper, S., "Hash joins and hash teams in Microsoft SQL Server", Proceedings of VLDB Conference, pp 86-97, 1998.
- [50] Klug, A., "Equivalence of relational algebra and relational calculus query languages having aggregate functions", ACM Press. Vol-29, No. 3, pp 699-717, 1982.
- [51] Chaudhuri, S. and Shim, K., "Including group-by in query optimization", In Proceedings of VLDB Conference, 1994.
- [52] Galindo-Legaria, C., "Outerjoins as disjunctions", Proc. of the ACM, SIGMOD Conf. on Management of Data, 1994.
- [53] Gupta, A. and Mumick, L. S., "Maintenance of materialized views: problems, techniques and applications", IEEE Data Engineering Bulletin, Vol-18, No. 2, 1995.

- [54] 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.
- [55] Dar, S., Jagadish, H. V., Levy, A. and Srivastava, D., "Answering queries with aggregation using views", Proceedings of VLDB Conference, 1996.

