Scalable Storage in Compressed Representation for Terabyte Data Management

by

Mohammad Abdur Rouf

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
April, 2006

The thesis **"Scalable Storage in Compressed Representation for Terabyte Data Management"**, submitted by Mohammad Abdur Rouf, Roll No. 100105021P, Session: October 2001, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, 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 April 19, 2006.

Board of Examiners

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.

Signature

[illegible]

(Mohammad Abdur Rouf)

Table of Content

List of Tables

List of Figures

Acknowledgement

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

I would like to acknowledge Dr. Douglas R. McGregor, Department of Computer and Information Science, University of Strathclyde, Glasgow, UK for providing his paper on HIBASE compression model.

I also express my gratitude to Professor Dr. M. Bashir Uddin, Head of the Department of Computer Science and Engineering, DUET, Gazipur for providing me enough lab facilities to make necessary experiments of my research. I am also grateful to my colleagues and relatives for encouraging me to continue my research.

Abstract

The emergence of e-application has been creating extremely high volume of data that reaches to terabyte threshold. Many organizations are producing data that are doubling every year. The conventional data management system is costlier in terms of storage space and processing speed, and sometimes it is unable to handle such huge amount of data. New algorithms and techniques need to develop to store and manipulate these data. The database compression can be used for scalable storage and faster data access.

We propose compression based data management system architecture that can be used to handle terabyte level of relational data. The existing compression schemes e.g. HIBASE or Three Layer Database Compression Architecture 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.

We have developed a disk based columnar multi-block vector structure (CMBVS) 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 lowest layer of the model is the block structure to store the compressed representation of data. The next higher level is the vector-structure that relates the block structure to an attribute of the relational data model. The structures are capable of carrying out query directly on the compressed form of data. This reduces query time drastically. We have compared our system with conventional relational DBMS. The experimental results show that our system is about twenty five times efficient in storage cost and twenty-seven to seventy-seven times faster in retrieval time performance than that of the conventional systems.

Chapter 1 Introduction

Storage requirement for database system is a problem for many years. Though the storage capacity has been increasing every year, in many cases, enterprise and service provider data storage needs double every six to twelve months.

It is a great challenge to store and manage this ever-increasing data in an efficient way. The amount of data reaches to terabyte threshold in many organizations. The conventional data management system for these huge data sets is a costlier one in terms of storage space and retrieval performance.

1.1 Motivation to Terabyte Data Management

Though the terabyte storage space is available, the price is high enough for common use. The backup of the terabyte database instance is another troublesome issue and time consuming matter. Accessing the terabyte database in highly concurrent environment is a great problem in conventional database system. The reason is that the number of disk accesses increase as database size grows, thus the response time of any query increases as well. The transactional throughputs create a bottleneck due to I/O intensive operations. Complexity of join operations become more complicated as dataset size grows.

Most of the large databases are often in tabular form. The operational database is not routinely terabyte in size whereas the typical size of fact tables in a data warehouse is generally terabyte in size. These data are write-once and read-many times category for further analysis. The recent developments in storage technology make it easy to store large volume of data in large disk space e.g. Network Attached Storage (NAS), Storage Area Network (SAN) and Redundant Array of Independent Disk (RAID). Still problem remains 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 reducing storage cost. High performance compression algorithm, necessary retrieval and data transfer technique can be candidate solution for terabyte data management system.

Reducing the volume of data without losing any information is known as loss-less data compression. This is potentially attractive in database system for two reasons:

- Reducing of storage cost without loss of information.
- Performance improvements.

It is difficult to combine a good compression technique that reduces the storage cost while improving performance. The performance improvements arise because 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 for a particular operation.

1.2 Aim and Objective of the Thesis

The objective of the thesis is to design a compression based system that can be used to store and retrieve terabyte level relational data efficiently. The features of the architecture are as follows:

- Data elements will be directly addressable in compressed representation irrespective of the location of compressed block in memory or disk. At the same time the compressed representation will be same for data being memory resident or in disk.
- Disk transfer time will be reduced due to compression.
- Decompression will be necessary only to get the final result. So decompression overhead will be minimized.

We use columnar format rather than the conventional row-wise format. Our aim is to develop disk based columnar multi-block vector structure for storage and querying of relational data in compressed form. The scalability can be achieved by distributing the structures among multiple servers in a shared nothing parallel architecture. The structure is such that the unnecessary columns need not to access for a particular query processing.

Every database system requires to support dynamic update. This requires a restructuring operation to the vector. The restructuring cost can be reduced by using columnar multi block vector structure (CMBVS).

Another aim is to design a multipart dictionary structure that can be easily used in terabyte data management system. Each attribute is associated to a domain dictionary. Some similar attributes may share the same dictionary.

1.3 Overview of the Compression Method

An overview of the compression and decompression method is given in Figure 1.1. Input goes through a compression engine that splits the data into dictionary and compressed representation of relational tables.

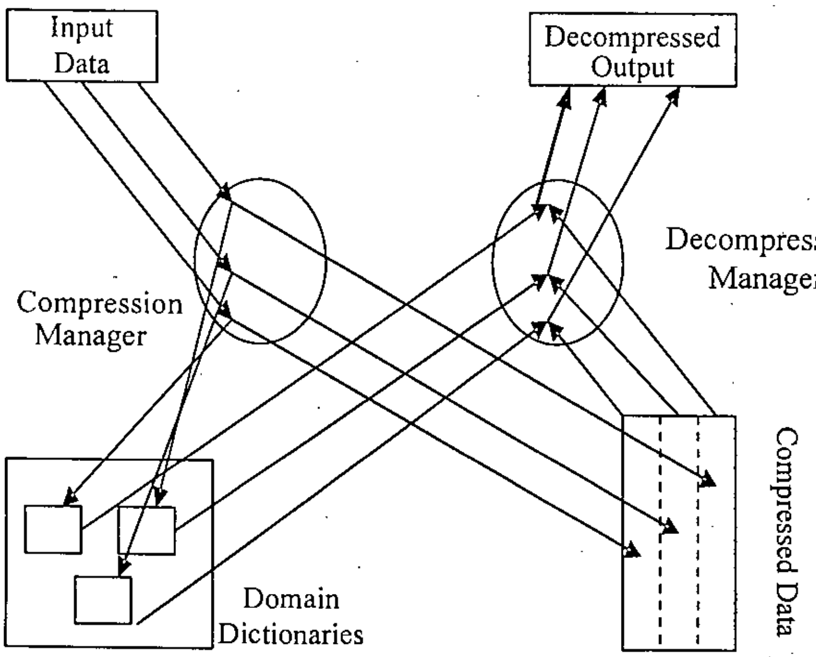


Figure 1.1: Overview of the Compression Technique

For input data, a dictionary is created per domain. Using the dictionary, a compressed representation respective to input data is created. We have developed a compressed representation that comprises of multiple blocks, which can be directly accessible. After accessing the block very easily, we can access any element within the block.

Direct addressability to the element in compressed form is characterized by the ability to retrieve an element without decompressing the whole structure. This allows the query processor to make query on the compressed representation. The decompression is a reverse process of the compression operation. Any element of the compressed representation can be decompressed by using the corresponding dictionary. All processing will be done on compressed form and decompression will be performed only when the result is necessary. This reduces the decompression overhead.

1.4 Columnar Multi-block Vector Structure (CMBVS) and Management of Compressed Data

The conventional database stores the relation in row wise format, which accesses the whole tuple for any particular query. The multi-block vector structure stores the relation in column wise format. Each attribute is associated to a domain dictionary and a dynamic vector. The vector is partitioned into multiple blocks. The vector accommodates a certain number of blocks in memory. When the number of blocks are increased, some blocks are swapped to disk. The high cardinality of domain dictionaries are also partitioned into blocks if necessary. The original tuple value is first searched in the dictionary, if the value is found in dictionary corresponding token is returned. If the value is not is not found in the dictionary, it will be inserted into the dictionary and corresponding token will be returned. It will then be inserted into the vector.

The query can be carried out directly on the compressed form of data. The query may be optimized if there are multiple attributes are accessed in the query. The query in the CMBVS only accesses the associated vector and dictionary, so that the I/O access is minimized and hence increased the query throughput.

1.5 Terabyte Data Management System Using CMBVS

The CMBVS is scalable to very large number of tuples in compressed form. Even a microcomputer (with 120 GB hard disk) can host huge data sets in compressed form like terabyte in size. However, the performance degrades as the database size increases.

The CMBVS provides the infrastructure that can be used in distributed system to manage multi-terabyte database. As the relations are stored in columnar format, the multi-block vectors can be distributed over column severs. The domain dictionaries are distributed over multiple domain servers. The system is scalable to virtually unlimited number of tuples, by increasing the number of disks in disk array of both column servers and domain servers.

We have developed the CMBVS structure and implemented it with Borland C++ 5.02. It is compared with widely used Microsoft SQL Server. We generate the data for voter relation of Bangladesh Electoral Database using our own data generator. We produce synthetic data for 85 million voters. The storage space occupied by the CMBVS is about 25 times smaller than that of SQL Server. The access speed is on an average 27 to 77 times faster than the conventional uncompressed counterpart in SQL Sever.

1.6 Thesis Outlines

In chapter 2, a review of the research in terabyte storage technology, terabyte data management and compression methods in database systems is presented. We have considered the HIBASE as the basis of our compression method. The dictionary organization and compression structure of HIBASE is also discussed.

Chapter 3 describes the CMBVS structure which is the basic building block of the terabyte data management system architecture. The details analytical description of the components of the system is also given.

Chapter 4 contains details of the experimental work that has been carried out and the discussions on the experiment. The experimental evaluation has been performed using various datasets. The experiment shows the performance of the terabyte data management system.

Chapter 5 presents the comments on the compression model and terabyte data management system. It also gives the suggestions for the future research.

Chapter 2
Literature Review

The terabyte problem emerges out very recently, as the information of some company is growing so fast that the size of information becomes double within six to twelve months [1]. This vast information is not easy to handle. So the database researchers need to think about how to solve this problem. This tends to make research on terabyte data management.

It is possible that some will neglect that a terabyte database with an unnecessary huge solution will have no practical use in normal life. The opposite is also true. In this age of electronic business, banking and electronic life in general, data is being formed "on the wire", which is producing a mammoth amount of data.

2.1 Terabyte Data Management

High performance data managements are emerging as critical technologies. Although it is becoming more common today to get a terabyte of disk, it is still an open problem how to manage, mine and analyze a terabyte of information. The terabyte challenge is an evolving, open testbed that can be used to test new algorithms and software for high performance and wide area management, mining and analysis of data. The first phase of Terabyte Challenge culminated at a demonstration during the Supercomputing Conference in San Diego, 1995 in which the Terabyte Challenge team members illustrated management, mining and analyzing a variety of large data sets totaling over 100 Gigabytes [2].

As information storage requirements continue to double and triple, companies of all sizes are facing a dilemma in managing corporate information. As multi-terabytes of stored information become the norm, the issue of backing up or restoring very large amounts of data in the shortest possible time has become increasingly urgent. Although backup times are critical, restore times are even more critical when the time comes to recover corporate data. Information recovery times are critical to the overall financial bottom-line of the corporation in today's competitive marketplace [3].

2.1.1 Terabyte Databases

The European organization for nuclear and particle physics research center is constructing a particle accelerator that will begin operating in 2006. The IT managers at the laboratory are expecting to collect up to 20 petabytes of data from the accelerator every year [4].

The SBC Communications Inc. in San Antonio runs a 20-terabyte NCR Teradata database, up from 10.5 terabytes in 2004 [4]. SBC's data warehouse has nearly doubled in size every year since it was built in 1994. Sears Roebuck & Co. is combining its customer and inventory data warehouse to create 70 terabyte system, the retailer will hit the 1 petabyte threshold within 4 years. It is expanding its database to make it more useful. It wants 35-terabyte data warehouse of product-inventory and store-sale information for retailer's restructuring as it tries to rebound from a year of falling sales and profits.

Comscore, stores 9 terabytes of click-stream data on 27 terabytes of disk space that's partitioned into two sections. One section holds aggregated data and the other stores detailed data. Both are regularly backed up on tape, a process that takes several days, but only the aggregated data is backed up within the database. That's because the aggregated data has undergone more processing and would take more work to reconstruct if lost [4].

ACNielsen's data warehouse requires multiple servers with a single database image [4]. That challenge prompted Sybase and Sun Microsystems last year to create the iForce data warehouse reference architecture to handle more than 25 terabytes of raw data. IForce is a blueprint that companies can use to assemble their own large data warehouses. General Motors North America in Detroit operates a number of large databases, including its engineering and product-development database, with 8 terabytes of raw data and 22 terabytes of disk space.

Florida International University went live with a 20-terabyte database of high-resolution aerial and satellite images of the entire United States provided by the U.S. Geological Survey [4]. The system is believed to be the largest publicly accessible database on the Internet. But data for individual images is distributed to avoid overloading any single server in the event of a spike in demand for a particular image--say, as the result of a natural disaster. To maintain the database's performance, the school is creating a hierarchy of data caches to hold more frequently accessed images. The caches should reduce the primary database's workload. The typical American consumer now generates some 100 Gigabytes of data during his or her lifetime, including medical, educational, insurance, and credit-history data. If we multiply that by 100 million consumers and we get a whopping 10,000 petabytes of data. A petabyte of data may seem like a lot to swallow today, but businesses' appetite for information shows no signs of diminishing [4].

2.1.2 Terabyte Storage

Dataquest, another leading industry analyst, reports the average desktop consumption of storage space has grown from 1.4 GB in 1997 to 3.5 GB in 1999 and is projected to reach 14 GB in 2003 [3]. Even though cost-per-megabyte for storage is declining at the rate of 35% to 40% per year, the increase in complexity of storage systems, such as mirroring, 24x7 backup, high-availability configurations, leads to increases in storage management costs [3]. As the need for more efficient information management grows, Internet, ecommerce, and digitization of data increase the requirement for more effective communications bandwidth. With this ever-increasing information storage need, comes an equally new level of complexity in the deployment and management of storage devices. This complexity can be significantly reduced by employing SAN/NAS (Storage Area Network/ Network Attached Storage) technologies into existing LAN/WAN environment, is shown in Figure 2.1 and Figure 2.2.

NAS signifies a Network Attached Storage device that provides high availability disk storage. It is attached to the network directly and not through a server. However SAN is a separate high-speed network with fiber channel backbone for shared storage. Each server on the LAN is also connected to the SAN, and these servers can be running any one wide variety of operating systems. These servers are directly connected to the storage devices. These storage devices can be of any tape or RAID arrays [5].

Now-a-days even terabyte of storage is gradually coming to the market. Nano-terabyte technology invent DVD up to 4.7GB and HD-DVD up to 20~25GB and enabling to reach 100GB~1000GB [6].

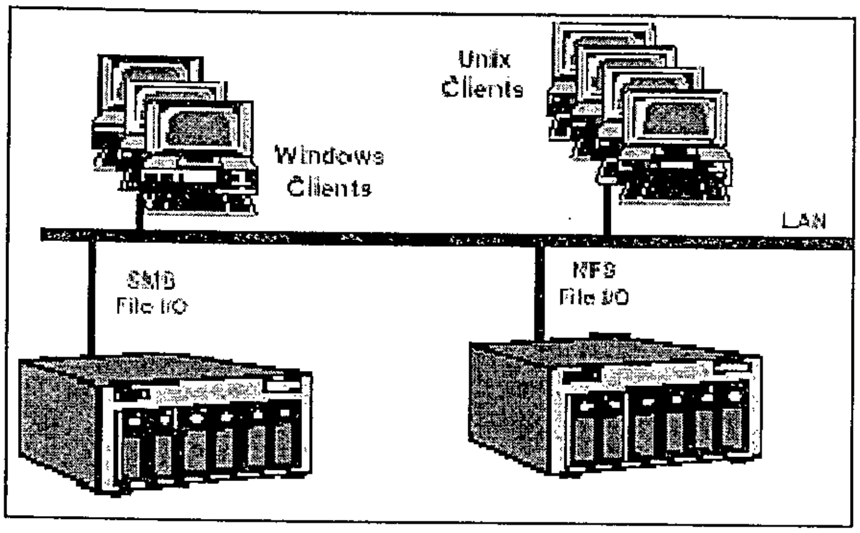


Figure 2.1: Simple Network-Attached Storage [3].

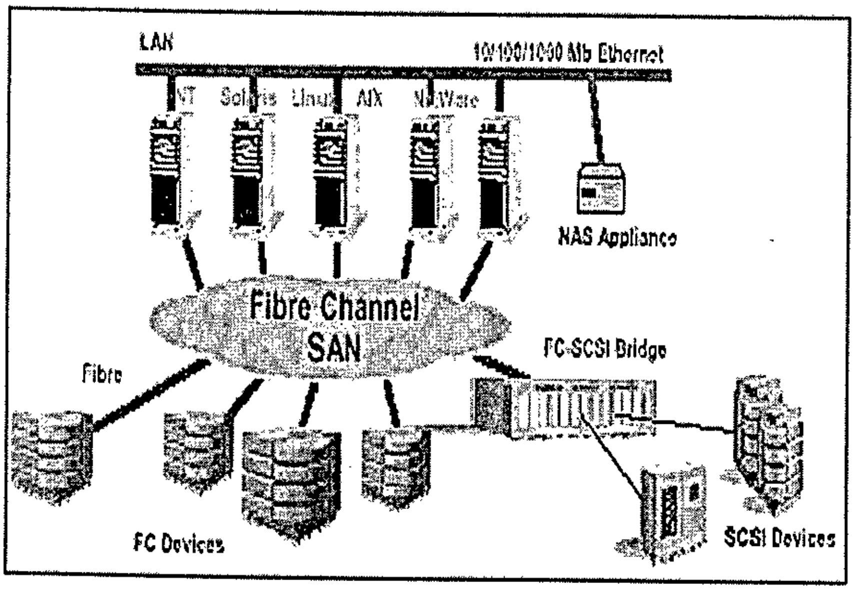


Figure 2.2: Fiber Channel Storage Area Network [3].

Maxtor's new external drive comes to market on the October, 2005, places a terabyte of storage space on the desktop - that's 1000 gigabytes of digital real estate ready to house business data, MP3s, digital images, video and just about any other electronic file that might have. It is designed to plug in to one PC or Mac [8].

ATP projects (Advanced Technology Program) develop technologies to increase the data storage density of existing magnetic tape data systems by a factor of 250, which lay a foundation to store greater densities in future system for terabyte data storage [9].

2.1.3 Terabyte Database Solutions

SGI Systems, Inc. in partnership with IBM Corporation, Legato Systems and Computer Associates Corporation performed backup of a one-terabyte Oracle7TM database in less than one hour [10].

A joint project of Microsoft, Dataware and HP has built up a terabyte data warehouse on Microsoft SQL Server 2000 platform with access via a thin client using a web browser with mobile access from a PDA computer or WAP telephone [7].

SQL Server 2005 is ready for the enterprise, offering exceptional data availability and manageability, hardened security, and the ability to scale from handheld mobile devices to the most demanding online transaction processing (OLTP) systems and multi-terabyte data warehouses [11].

Oracle today announced that Oracle powers the world's largest commercial database, according to the Winter Corporation 2005 TopTen (TM) Program [12]. The customer database is a 100 terabyte (TB) data warehouse, and is more than triple the size of the world's largest database (29.2 TB) in the previous TopTen Program (November 2003), which was also powered by Oracle.

2.1.4 Terabyte Data Warehouse

Data warehousing and on-line analytical processing (OLAP) are essential elements of decision support, which has increasingly become a focus of the database industry. Many commercial products and services are now available, and all of the principal database management system vendors now have offerings in these areas. Decision support places some rather different requirements on database technology compared to traditional on-line transaction processing applications. An overview of data warehousing and OLAP technologies is given in [13], with an emphasis on the new requirements. This paper describes the back end tools for extracting, cleaning and loading data into a data warehouse; multidimensional data models typical of OLAP; front end client tools for querying and data analysis; server extensions for efficient query processing; and tools for metadata management and for managing the warehouse. Typically, the data warehouse is maintained separately from the organization's operational databases. There are many reasons for doing this. The data warehouse supports on-line analytical processing (OLAP), the functional and performance requirements of which are quite different from those of the on-line transaction processing (OLTP) applications traditionally supported by the operational databases. OLTP applications typically automate clerical data processing tasks such as order entry and banking transactions that are the bread-and-butter day-to-day operations of an organization. These tasks are structured and repetitive, and consist of short, atomic, isolated transactions. The transactions require detailed, up-to-date data, and read or update a few (tens of) records accessed typically on their primary keys. Operational databases tend to be hundreds of megabytes to gigabytes in size.

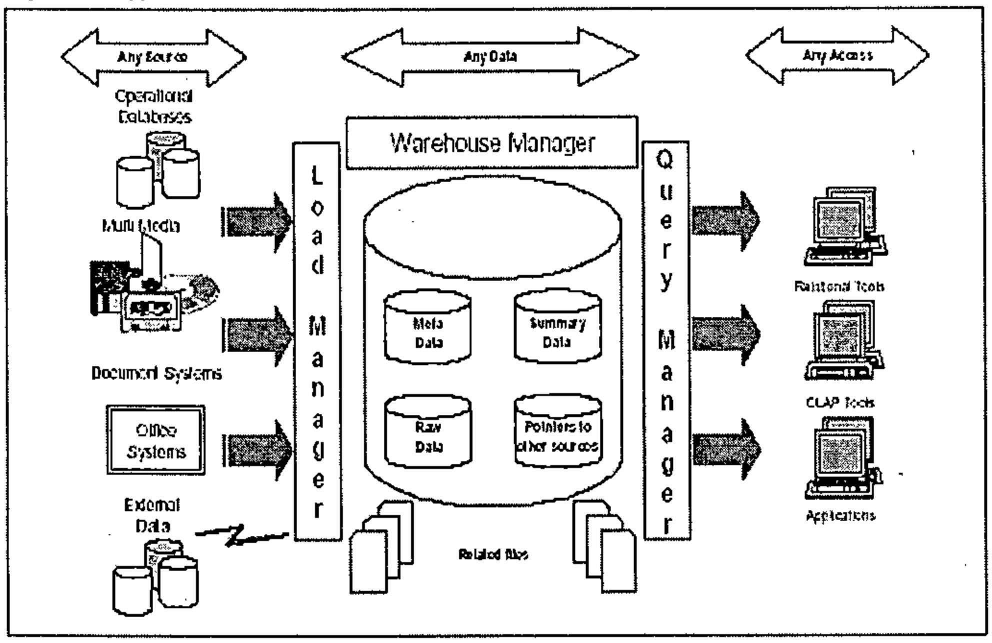


Figure 2.3: A Typical Data Warehouse [15].

Many organizations around the world have come to rely totally upon the essential information asset stored in their corporate databases and data warehouses. A typical data warehouse is shown in Figure 2.3. If the information is not available for any reason the business may grind to a halt, and if it stays unavailable for a protracted period, serious financial consequences may result. Making a data warehouse available is not easy. Corporate data warehouses range from one terabyte to ten terabytes or more, with the intention of giving users global access on a 24x365 basis. Data warehouses can take months to set up, yet can fail in seconds. And to react to changing business requirements the data warehouse will need to change in design, content and physical characteristics on a timely basis. The characteristics of the data warehouses and how to make it safe, available, perform well, and be manageable is described in [15].

2.1.5 Terabyte Data Management Using Conventional DBMS

The Czech researchers manage 1.1 TB of data in the SQL Server 2000 platform [7]. The researchers use 2 servers, the first of them, HP Net-Server LT6000r with 6 Xeon-700 MHz processors, 4 GB RAM and the Microsoft Windows 2000 Advanced Server operating system was used for the relational data warehouse. The second, HP Net-Server LXr8500DC with 8 Xeon-700 MHz processors, 4 GB RAM and the Windows 2000 Datacenter Server operating system had the task of creating multidimensional analytical data cubes and interacting with users. The HP SureStore XP512 disk array are used to connect the two servers by a pair of fiber channels. The disk array consisted of 32 disks with a capacity of 18 GB for applications demanding access to data at a maximum speed and another 32 disks with a capacity of 73 GB for applications demanding maximum capacity. The total gross capacity was therefore 2.8 TB. To protect against malfunction, RAID-5 arrays have been created from the discs, making available disc capacity 2 TB. For still more demanding applications, HP SureStore can be configured to a gross capacity of 92.6 TB when using disks sized to 181 GB. The system does not have any compression mechanism so that the huge storage space is required to manage the terabyte data.

The problems and critical issues of migrating a multi-terabyte archive from object to conventional relational database is discussed in [28]. This does not provide any general solution of terabyte data management system.

A terabyte scale test is performed at IBM's Teraplex center that is described in [30]. This test is done to make research on large scale data warehouse using conventional DBMS. The test examined only the role of OLAP technology in large scale data warehousing architecture, but the architecture does not apply any compression technique.

2.1.6 Tera-scale Architecture

An architecture for archiving and analyzing real-time scientific data [14] is given in Figure 2.4. It isolates researchers from the complexities of data storage and retrieval. Data access transparency is achieved by using a database to store metadata on the raw data, and retrieving data subsets of interest using SQL queries on metadata. The second component is a distributed web platform that transparently distributes data across web servers.

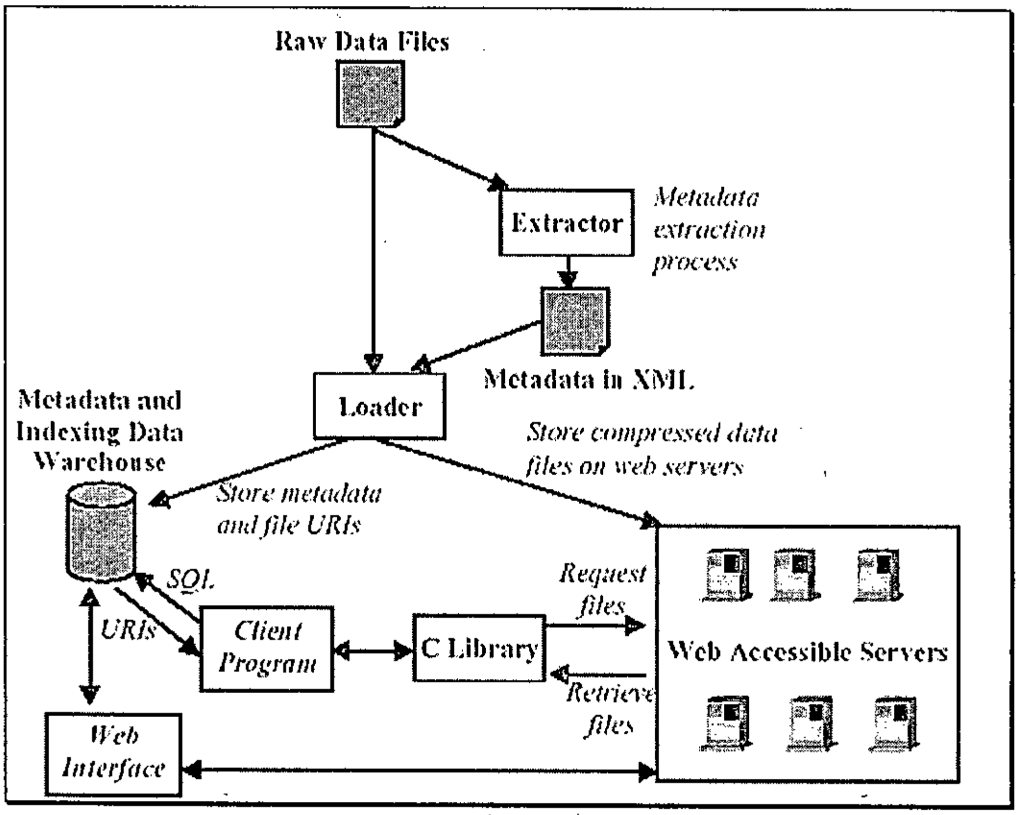


Figure 2.4: Data Archival Architecture for Real-Time Warehousing of Scientific Data [14].

The amount of scientific data is growing rapidly that is a challenge to store it and make query efficiently. The size becomes terabyte; and the archiving of such large-scale data is a great problem in terms of access speed and storage space. The architecture [14] provides a framework for archiving and retrieving the large-scale scientific data using web server and warehousing technology. It uses a compression technique to minimize the storage space, but the compression model does not allow query support directly on the compressed form. The loader stores the data files in compressed form to the web servers. The retrieval process retrieves the decompressed data and then applies any SQL query on the decompressed form. As it cannot carry out query directly on the compressed form, the query processing time is not the optimal one.

This architecture achieves the four design goals of scalability, extensibility, cost-effectiveness, and usability. The architecture is *scalable* as new servers can be added to store data files as required, and the archive can efficiently process multiple terabytes of data. The architecture is *extensible* as the metadata extraction process can be configured to calculate different types of metadata, which allows the architecture to be applied to different scientific domains.

The architecture provides the archiving mechanism only for the scientific domain. This is not applicable to business oriented relational data, whereas most of the business data are relational. Therefore, the architecture is failed to fulfill the market demands.

2.2 Data Compression for Management of Terabyte Database

The Tera-scale Scientific Data Management [14] uses the compression to archive the data files in web servers, the query cannot be processed directly in the compressed files. The decompression is required to make any searching on the data.

Oracle uses block level dictionary based compression technique. The compression model incurs some redundancy but a block itself contains all information that is required to decompress. The compression of Oracle database is to manage large data sets. The details of the compression model are discussed in section 2.3.

The Kx Database Technology uses columnar format [26] to store large database but does not apply any compression mechanism. Though the columnar format really exhibits high performance in data access, it is not as fast as the compressed columnar format shows.

The architecture for multi terabyte, hierarchical data warehouse for continuous, high-rate object stream archiving relational data in a hierarchical storage system composed of serialized objects that have been binned and indexed is described in [16]. The architecture does not apply any compression technique, so obviously the I/O access rate is high and thus reduces the query performance.

A column oriented compression method C-Store [27] improves the performance of the DBMS. It is designed mainly to support high performance query processing but has no guideline about terabyte data management.

A general purpose compression scheme for image, voice and text data is given in [31]. This is not applicable to conventional relational DBMS. A high-performance data structure is developed in [29] for mobile information. But this does not guide for terabyte data management.

2.2.1 Compression Methods

It is better to apply a compression technique to manage a terabyte level database. This will occupy small amount of storage space but performance is attractive one. It is difficult to achieve both storage reduction and performance improvement simultaneously in a database compression. However, a few compression techniques have the trade-off [18, 21].

According to Shannon [17], it has been known that the amount of information is not synonymous with the volume of data. Reducing the volume of data without losing any information is known as loss-less data compression. Loss-less data compression can be attractive for two reasons: data storage reduction and performance improvements. Storage reduction is a direct and obvious benefit but performance improvement arise as follows:

- Main memory access time is in the range of nanoseconds while disk access is in the range milliseconds. Only a smaller amount of compressed data needs to be transferred and/or processed to effect any particular operation. Thus, it is improving I/O processing and hence improving the overall performance.
- A further significant factor is now arising in distributed application using mobile and wireless communication. Low bandwidth is a performance bottleneck and data transfers may be costly. Both of these factors make data compression well worthwhile.

Combining compression with data processing improves performance. Database systems require providing efficient addressability for data, and generally must provide dynamic update. It has proved difficult to combine these with good compression technologies. Much research work has been done in database systems to exploit the benefit of compression in storage reduction and performance improvement [18].

2.2.2 Dictionary Based Methods

The compression scheme for database permits operations directly on the data in its compressed representation. It should allow for the data representation of any specific tuple of a relation to be directly addressable. To translate the compressed form, it is necessary to go through a dictionary. In its simplest form, a dictionary can be a list of unique values that occur in the domain [18].

2.2.3 The Implication of Compression on Database Processing

Compression has become now an essential part of many large information systems where large amount of data need to be processed stored or transferred. The data may be of any type e.g. voice, video, text, XML, table, etc. No single compression technique is suitable for all types of data. Lossy compression is use for voice or video data whereas loss-less compression is for most other data types. A common feature of all these types of data is that the data redundancy occurs in columns instead of rows. The basics of compression model, i.e. a column oriented dictionary based architecture is described in [18].

2.2.4 Compression of Relational Structure

Relational structure can be benefited using compression as follows:

Significant improvement occurs in index structures such as B-tree and R-tree by reducing the number of leaf pages. Reduction in transaction turnaround time and user response time as a result of faster transfer between disk and main memory in I/O bound system. In addition, since this will reduce I/O channel loading, the CPU can process many more I/O systems and thus channel utilization is increased [20].

Improvement in the 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 [19].

Processing data in compressed form makes it possible for the whole, or the major part of a database to be memory resident. Main memory access time several orders of magnitudes faster than the secondary storage access time. Thus improves performance.

The horizontal organization has the advantage that a single disk access fetches all attribute values in a tuple. Even for non-compressed memory resident databases, where memory access speeds are much faster, the implementation simplicity is appreciable. A relation can also be represented as a sequence of column vectors in which corresponding attribute values in successive tuples is stored adjacently [21].

In a main memory database system (MMDB) the data resides permanently in main physical memory and memory resident data may have a back up copy on the disk. In conventional disk based database systems the primary copy of the data resides in the disk and a small portion of data is memory resident. The distinguishing features of MMDB described in [22] are the access time of main memory is several orders of magnitude less than for disk storage. Disk has a high fixed cost per access that does not depend on the amount of data that is retrieved during the access.

2.2.5 Table Data Compression

Effective exploratory analysis of massive, high-dimensional tables of alpha-numeric data is a ubiquitous requirement for a variety of application environments including corporate data warehouses, network traffic monitoring or large socio-economic or demographic surveys. Compression and decompression are not symmetric but are based on the concept of compress once decompress many times. An example of massive data tables is Call Detail Records (CDR) of large telecommunication systems. A typical CDR is a fixed length record structure comprising several hundred bytes of data that capture information on various attributes of each call. These include network level information (e.g., end point exchanges), time-stamp information and billing information. These CDRs are stored in tables that can grow to truly massive sizes, in the order of several terabytes per year. Database compression differs from table compression in many ways.

Database compression stresses the preservation of indexing-the ability to retrieve an arbitrary record under compression. Table compression does not require indexing to be preserved.

Database records are often dynamic but the table data are write-once discipline. Databases consist of heterogeneous data whereas table data are more homogeneous, with fixed field lengths. Unlike table data, databases are not routinely terabytes in size.

The approach to database compression requires either lightweight techniques such as compressing each tuple by simple encoding or compression of the entire table. These approaches are not appropriate for table compression: the former is too wasteful and the latter is too expensive.

Table compression was introduced by Buchsbaum et.al. [23], as a unique application of compression based on several distinguishing characteristics. They have introduced a system called pzip. The basis of the method is to construct a compression plan studying a very small training set off-line. The plan is based on data dependency. They have defined two types of data dependency: combinational and differential. If two data intervals have the compressed size in separate intervals larger than the compressed size in a combined single interval, the intervals are combinational dependent. Differential dependency is an explicit dependency between columns. Though pzip outperforms gzip by a factor of two, partitioning the data determining the dependency is a problem. Optimum partitioning is not possible using the training data set. The primary focus is to optimize the compression ratio within a user defined error bound. However none of these methods can be applied to databases where no loss of information is permissible.

Sparsely-populated table data arises when sometimes data is represented using a single horizontal table. An example is the sparse bit-map for a digital library, the electronic marketplace in and the news-portal system in IBM-Almaden [24]. Sparsely populated data can be compressed using run-length encoding method given in [18].

2.3 Compression in Oracle Database

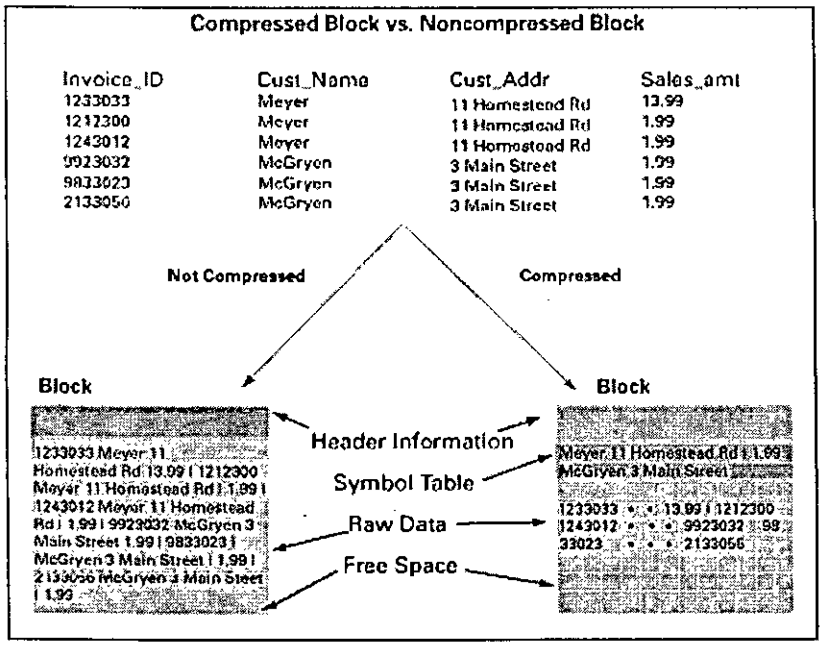
The Oracle RDBMS recently introduced an innovative compression technique for reducing the size of relational tables [25]. By using a compression algorithm specifically designed for relational data, Oracle is able to compress data much more effectively than standard compression techniques. More significantly, unlike other compression techniques, 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.

In past commercial database systems have not heavily utilized compression techniques on data stored in relational tables. A standard compression technique may offer space savings, but only at a cost of much increased query elapsed time. Hence, this trade-off has made compression not always attractive for relational databases.

Meikel Poess and Dmitry Patapov et.al. [25] recently describe how to compress Oracle table data in [25], that is an attractive solution for large relational data warehouses. It can be used to compress tables, table partitions and materialized view.

The compression algorithm used in Oracle for large data warehouse tables compresses data by eliminating duplicate values in database block (or page). The algorithm is lossless dictionary-based compression technique. The compression window for which a dictionary (symbol table) is created consists of one database block. Therefore, compressed data stored in a database block is self contained. That is, all the information needed to recreate the uncompressed data in a block is available within that block.

Figure 2.5 illustrates the differences between compressed versus non-compressed block. The top part of the Figure 2.5 shows a typical data warehouse like fact table with row ID, invoice Id, customer first name, customer last name and sales amount. There are entries for five customers showing six purchases. For data warehouse fact table it is very common to have this highly denormalized structure.


Compressed Block vs. Noncompressed Block

Invoice_ID	Cust_Name	Cust_Addr	Sales_amt
1233033	Meyer	11 Homestead Rd	13.99
1212300	Meyer	11 Homestead Rd	1.99
1243012	Meyer	11 Homestead Rd	1.99
9923032	McGryen	3 Main Street	1.99
9833023	McGryen	3 Main Street	1.99
2133056	McGryen	3 Main Street	1.99

Figure 2.5: Oracle Database Compression [25].

The bottom right part shows how the same data is stored in a compressed block. Instead of storing all data, redundant information is replaced by links to a common reference in the symbol table, indicated by the black dots. For each column value in all columns, based on length and number of occurrences in one block, the algorithm decides whether to create an entry into the symbol table for this column value. If column values from different columns have the same values, they share the same symbol table entry. This is referred to as cross-column compression. Only entire column values or sequences are compressed. Sequences of columns are compressed as one entity if a sequence of column values occurs multiple times in many rows. This is referred to as multi-column compression. This optimization is particularly beneficial for OLAP type materialized views using grouping sets and cube operators. For instance a cube of a table often repeats the same values along dimensions creating many potential multi-column values. Multi-column compression can significantly increase the compression factor and query performance. In order to increase multi-column compression, columns might be reordered within one block. For short column values and those with few occurrences no symbol table entry is created limiting the overhead of the symbol table and ensuring that compressing a table never increases its size. However, this is transparent to any application. This method improves I/O performance but the query processing requires a decompression on page by page basis.

2.4 Dictionary based HIBASE Compression Approach

The HIBASE [21] approach is a more radical attempt to model the data representation that is supported by information theory. The architecture represents a relation table in storage as a set of columns, not a set of rows. Of course, the user is free to regard the table as a set of rows. However, the operation of the database can be made considerably more efficient when the storage allocation is by columns.

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

Table 2.1: An Example of Simple Voter Information

First Name	Last Name	Village	Sex	District
Abdur	Rahim	Rampur	M	Tangail
Nasir	Uddin	Fulpur	M	Mymansing
Based	Mia	Rampur	M	Tangail
Abdur	Rouf	Fulpur	M	Mymansing
Nasir	Mia	Vuralia	M	Gazipur
Hasina	Begum	Rupganz	F	Gazipur
Saleha	Begum	Rupganz	F	Narayngonj
Parvin	Begum	Rampur	F	Tangail

An example of a relation is given in Table 2.1. In the conventional database technology, we have to allocate enough space to fit the largest value of each field of the records. When the database designer does not know exactly how large the individual values are, he/she must err on the side of caution and make the field larger than is strictly necessary. In this instance, a designer should specify the width in bytes as shown in Table 2.2. Each tuple is occupying 104 bytes, so that 8 tuples occupy 832 bytes.

Table 2.2: Field Lengths and Tuple Size for the Voter Relation.

Attribute	Attribute Name	Bytes
0	First Name	20
1	Last Name	20
2	Village	30
4	Sex	4
3	District	30
Total		**104**

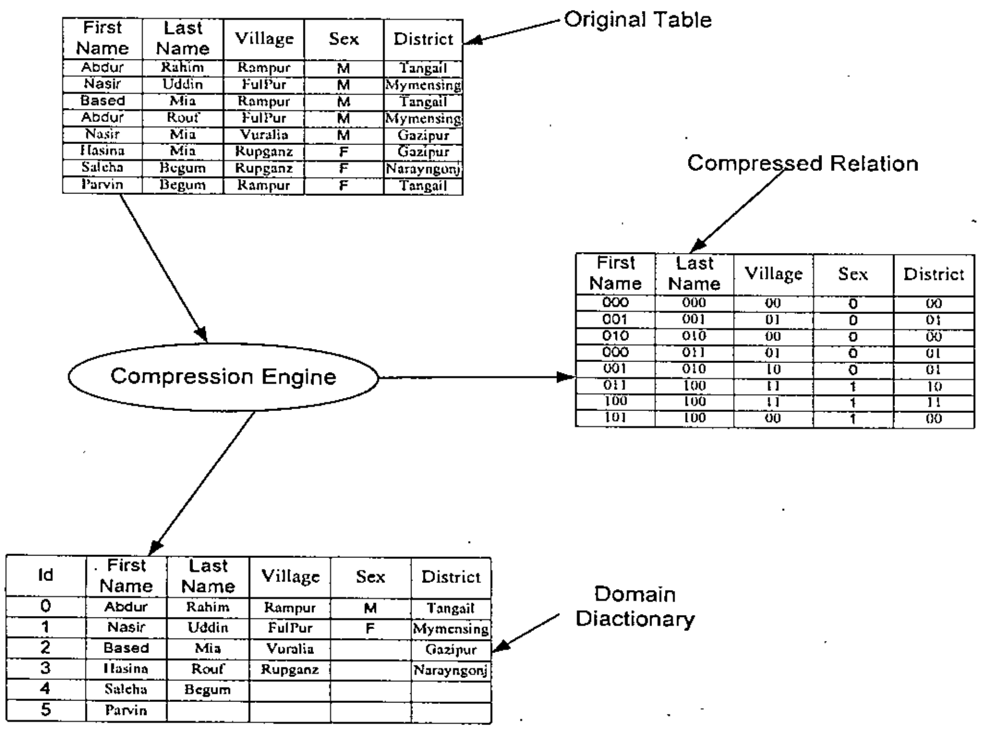
2.4.1 HIBASE Compression Architecture

The architecture given in Figure 2.6 is a compact representation that can be derived from a conventional record structure using the following steps:

- A dictionary per domain is employed to store the string values and to provide integer identifiers for them. This achieves a lower range of identifier, and hence a more compact representation than could be achieved if a single dictionary was provided for the whole database.

- Replace the original field value of the relation by identifiers. The range of the identifiers is sufficient to distinguish string of the domain dictionary. The example given in Figure 2.6 shows that there are only seven distinct first name, so only a 3-bit can represent this attribute.

Hence in the compressed table each tuple resume only 3 bits for First Name, 3 bits for last Name, 2 bits for village, 2 bit for district and 1 bit for sex forming total of 11 bits. This is not the overall storage; however, we must take account of the space occupied by the domain dictionaries and indexes. Typically, a proportion of domain is present in several relations and this reduces the dictionary overhead by sharing it by different attributes.



First Name	Last Name	Village	Sex	District
Abdur	Rahim	Rampur	M	Tangail
Nasir	Uddin	FulPur	M	Mymensing
Based	Mia	Rampur	M	Tangail
Abdur	Rouf	FulPur	M	Mymensing
Nasir	Mia	Vuralia	M	Gazipur
Hasina	Mia	Rupganz	F	Gazipur
Salcha	Begum	Rupganz	F	Narayngonj
Parvin	Begum	Rampur	F	Tangail

First Name	Last Name	Village	Sex	District
000	000	00	0	00
001	001	01	0	01
010	010	00	0	00
000	011	01	0	01
001	010	10	0	01
011	100	11	1	10
100	100	11	1	11
101	100	00	1	00

Id	First Name	Last Name	Village	Sex	District
0	Abdur	Rahim	Rampur	M	Tangail
1	Nasir	Uddin	FulPur	F	Mymensing
2	Based	Mia	Vuralia		Gazipur
3	Hasina	Rouf	Rupganz		Narayngonj
4	Salcha	Begum			
5	Parvin				

Figure 2.6: Compression of a Relation Using Domain Dictionaries.

2.4.2 Dictionary Structure in HIBASE

HIBASE used two alternative representation of the values stored in attribute: token and lexemes. A token is a sub-range of integers represented in its minimal binary encoding.

Lexemes are a sequence of 0 or more 8-bit characters. The translation of string value to token is optimized using the minimal number of stored bits. String of decimal digit can be directly converted into binary at the database designer's discretion. Other data, which can not be represented directly as an integer, such as character string or real, are translated using dictionaries although in principle, real can be encoded as primitive data type.

A Dictionary must have Three Characteristics:

- ❑ It should map attribute value to their encoded representation during the compression operation: encode (lexeme)→token.
- ❑ It should perform the reverse mapping from codes to literal values when parts of the relation are printed out.
- ❑ Decode (token)→lexeme.
- ❑ The mapping must be cyclic such that x=encode (decode (x)).

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. The HIBASE compression does not support Unicode which is essential for current database applications.

2.5 Summary

This chapter described the different existing terabyte databases, database scenario in data warehouse and compression methods with an emphasis on dictionary techniques for database applications. We have discussed the relationship between compression and database representation. The HIBASE compression method has been described in detail as we have considered this approach as the basis of our architecture. But there are fundamental differences with our methods. As for example, the dictionary structure, the compressed representation and the query processing.

Chapter 3
Terabyte Data Management System

This chapter describes the detail analytical model of Columnar Multi-Block Vector Structure (CMBVS) for storage of compressed representation of relational data. CMBVS is the basic component of Terabyte Data management System. We develop the system for single processor and extend it to a parallel architecture.

3.1 Basic Compression Technique

We extend the basic HIBASE compression model to facilitate multi-block vector structure with disk support. The vector is column oriented rather than the conventional row-wise format. Each attribute is associated to a domain dictionary and a column vector. Using the HIBASE Compression method described in section 2.5, we can represent a sample relation given in the Table 2.1 as a column structure as shown in Figure 3.1.

The column structure represents a table in storage as a set of columns, not a set of rows. This makes certain operation on the compressed database considerably more efficient. The columns are organized as a linked list, each of which points to the dictionary that is used to compress the column. The vertical slices through the tuples are then stored in compressed, bit-aligned vectors.

Representation by column can be processed more efficiently than representation by row. 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.

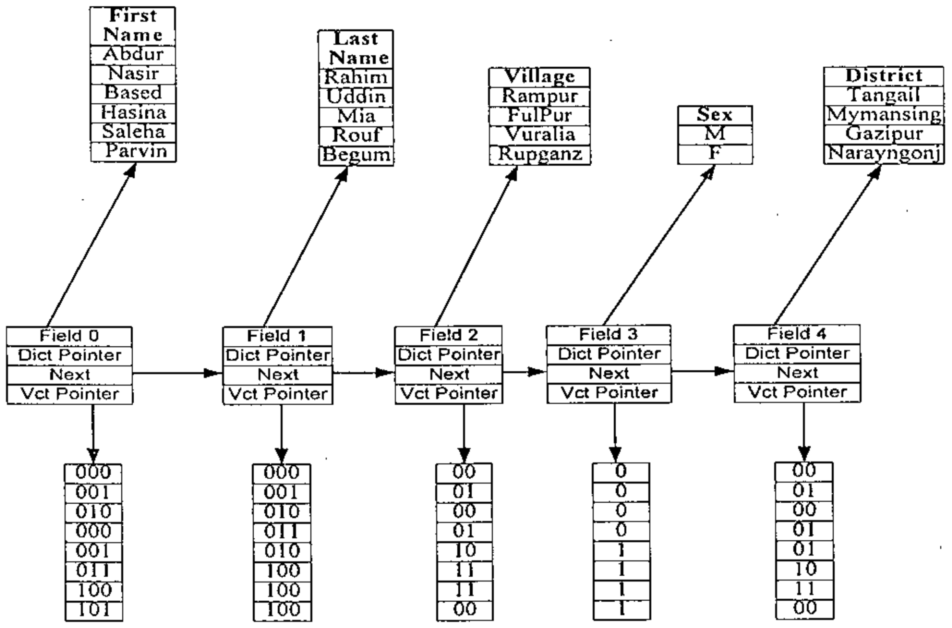


Figure 3.1: Column Organization of Relation Rather than Row Wise Format.

The processing speed of a query is enhanced because, a typical query specifies operation on only a subset of domains. In a column wise database, only those specified values need to be transferred, stored and processed.

The HIBASE model uses single block column vector, the whole vector has the same size of compressed element. It is costlier to restructure the whole vector in single block structure. We use the multi-block vector structure that partitions the vector into multiple blocks. Each block has a fixed number of elements and the block can be restructured independently. So the restructuring affects only the blocks that are involved.

The blocks are independently addressable so the structure is suitable for large database. In the architecture we use the multi-block vector structure. We call the structure CMBVS (for Columnar Multi Block Vector Structure) through out the chapter. The HIBASE structure was designed for memory resident data but we have designed CMBVS for disk support. So the ever-increasing data can be easily stored in the structure. The Three Layer Model by Hoque A.S.M. et. al. [18] uses also multi-block structure but does not have disk support.

3.2 Column Oriented Multi Block Vector Structure

The CMBVS is the container of compressed tuple value of an attribute. The CMBVS is divided into multiple blocks. A block is the lowest layer of the system. The CMBVS is the next higher level. First of all we discuss the block structure.

3.2.1 Block Structure

The block structure is the lowest layer of the system. This is a logical block of compressed elements. A block is a collection of multiple machine words that accommodate a fixed number of compressed elements of same size in compact form. It maximizes the utilization of storage space and minimizes retrieval time. A typical structure of a block is given in Figure 3.2.

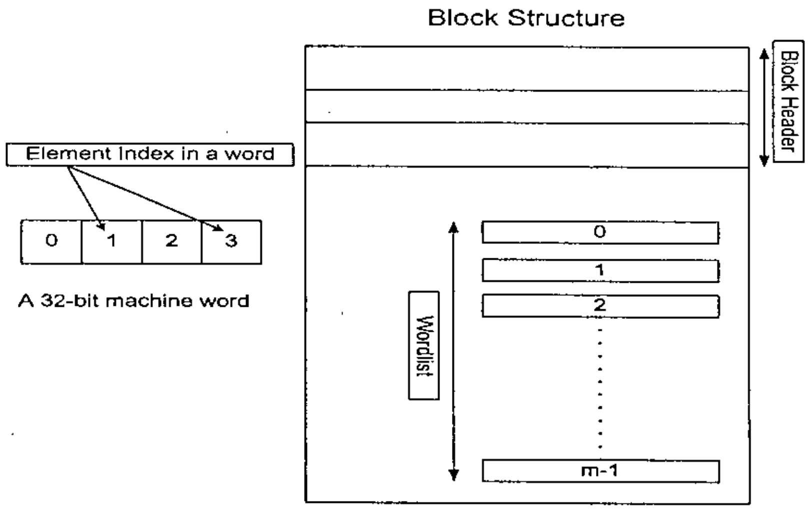


Figure 3.2: A Typical Block Structure

Each block has a header that represents the properties and some important aspects of the block like:

(i) **Block number:** It is the identification number of the block.

(ii) **Pointer to word list :** This pointer points to a wordlist that contains the compressed elements.

(iii) **Current word index:** This identify the current word from the wordlist.

(iv) **Number of elements in the block:** Specify the number element per block.

(v) **Number of elements per word:** Specify the number of elements per word.

(vi) **Element size:** Indicate the element size of the block.

(vii) **Element index in word:** This specify the element in a word.

Block Operations:

The class block performs the following main operations:

i) **insert_token(token, element_size):** insert a new token to the block. The insertion always occurs at the end of the block. The element size of the inserting element should be same as the element size of the block. If the element size of the token is larger than the element size of the block then widening operation is issued.

ii) **Get_token(i):** This operation returns the ith token from the block and i must be smaller than or equal to total number of elements in the block.

iii) **Widening (old_element_size, new_element_size):** Widening operation is issued when the element-size of inserting token is larger than that of existing element-size. During the widening operation, all the elements in the block are reorganized and element size is set to new element size. To perform this operation, all the existing tokens are inserted to a temporary wordlist with new element size then the new element is added to the temporary wordlist. The old wordlist is deleted and the properties of the block are set with new element size.

iv) **Get_Tuple_Id (token):** This operation returns a set of record-ids from the block that is matched to the token. If no matching is found in the block then it returns minus one. This function returns an enumerated list of record-ids.

3.2.2 Multi-Block Vector

The vector consists of multiple blocks. This is the next higher level than the block structure. Any operation in the vector ultimately propagated to the block. The following sections describe the details about the vector structure and operation. Multi-block structure reduces reorganization cost of vector and also reduces the wastage of space.

A general database system must support dynamic incremental update, while maintaining efficiency of access. We have considered a database as a set of columns instead of rows. In order to accommodate additional tuples, and additional lexemes in domains, each column requires being dynamically adjustable in width as well as length. To do this efficiently: it must allow easy adjustment of the number of tuples in relations.

It must also allow alteration in the length of the identifiers in use. When a compressed representation is in use, from time-to-time as updating proceeds, the range of element identifiers and hence their size must be altered. Incremental update of a column-oriented organization has a lower cost than for incrementally altering elements in a row-oriented representation. In the row-oriented case, all attribute values must be copied, not just the value being updated. This does not occur with a column-oriented representation.

It must be possible to address efficiently and directly each attribute value without requiring accessing preceding or succeeding tuples or values. This is called direct addressability.

In the single block organization, the worst-case temporary additional storage needed for the widening operation is the size of the widest vector in the database. There are two more problems: copying and wastage of space.

The followings are some significant features of multi-block structure:

Blocks can be added or removed to accommodate addition, or restructuring of the vector. During evolution of the database, elements of some attributes may need 'broadening'. Each block can be adjusted dynamically and incrementally to accommodate changes in the size of element being stored in it. This block-at-a-time reorganization is the key to enabling the DBMS designer to meet deterministic maximum update time constraints, and hence to achieve scalable update performance.

Without this independence of update for individual blocks, the storing of a longer identifier would require adjustments of the vector width (and hence copying) of the entire storage of the vector, a task proportionate to the size of the vector. This is one of the most important aspects in the use of this structure in database systems.

The independence of each block permits data compression to be applied within the block. A base-and-offset representation can be used; each element being stored as its offset from a base element for the block. This could be the minimum element stored in the block. The situations in which the elements are in fact ordered, or partially ordered this may be particularly effective in reducing the size of each stored element. The base element may be a minimum, average, or maximum value.

The maximum and minimum values are also particularly useful in rapid searching of a vector for an element with a particular value, elements within a range, elements belonging to a set of elements specified by a query.

The vectors are used to provide basic storage units from which relations, dictionaries etc. are constructed.

There are two options for organization of block: fixed number of elements per block or fixed size. According to [18], the fixed number of elements offer better performance than the fixed size.

In a fixed number of element option, each block contains a fixed number of elements and hence the memory requirement for different block is different and depends on the size of elements in that block. This may be fixed when the system designer fixes the number of elements per block. In practice it does not appear to be a parameter to which the system's performance is sensitive. To achieve the full space utilization the number of elements in each block is to be a multiple of word size. This decision also reduces the computational complexity. The size of element stored in the vector is uniform within any block, but may vary between blocks. This arrangement though slightly less convenient to manage in terms of memory allocation, it provides direct addressability to the ith element by means of few simple operations.

A multi-block vector structure is shown in Figure 3.3. It contains **m** blocks in memory and the remaining others are in disk. If we consider that the vector contains **k×m** blocks in total and **n** elements per block, the total number of elements in memory is **m×n**.

3.2.2.1 Providing Disk Support

As the main memory is limited, the entire vector of large database could not be accommodated in memory. A certain number of blocks reside in memory, the others are kept in backing store. Figure 3.3 shows that **m** blocks can reside in memory. Let ith block is mapped to jth position of vector and there are total **k×m** blocks in vector. If ith block is referred, it is first searched in j^{th} vector position. If the database block number of j^{th} block in vector is same as i, then j^{th} block in vector is the target block other wise issue a disk access to load the i^{th} block form disk.

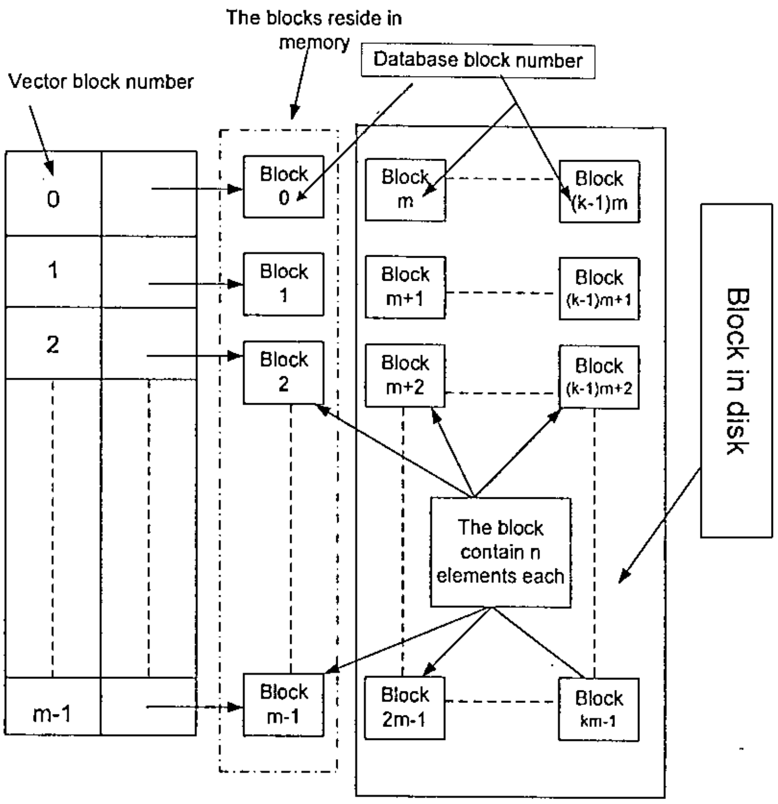


Figure 3.3: Multi-Block Vector Structure

3.2.2.2 Vector Operations

A vector represents a column of relational database. The main attributes of the vector class is :

i) **total number of blocks :** represents the total number of blocks that can reside in memory

ii) **total number of elements:** total number of elements that are in a vector

iii) **current block pointer:** a pointer to current block

iv) **database block number:** the last block index number of database.

Main Vector Operations:

i) **insert token(token, token size):** The vector insertion is always propagated to the insertion to the current block. If the block is full new block is added (with *yy* block number) to vector. The token is inserted to new block and the current block is stored to disk with *yy* identification number. The flow chart of the insertion operation is given in Figure 3.4.

```
If (current block is full)
        add new block to vector;
        disk_write(current block) with yy identification number;
endif;
```

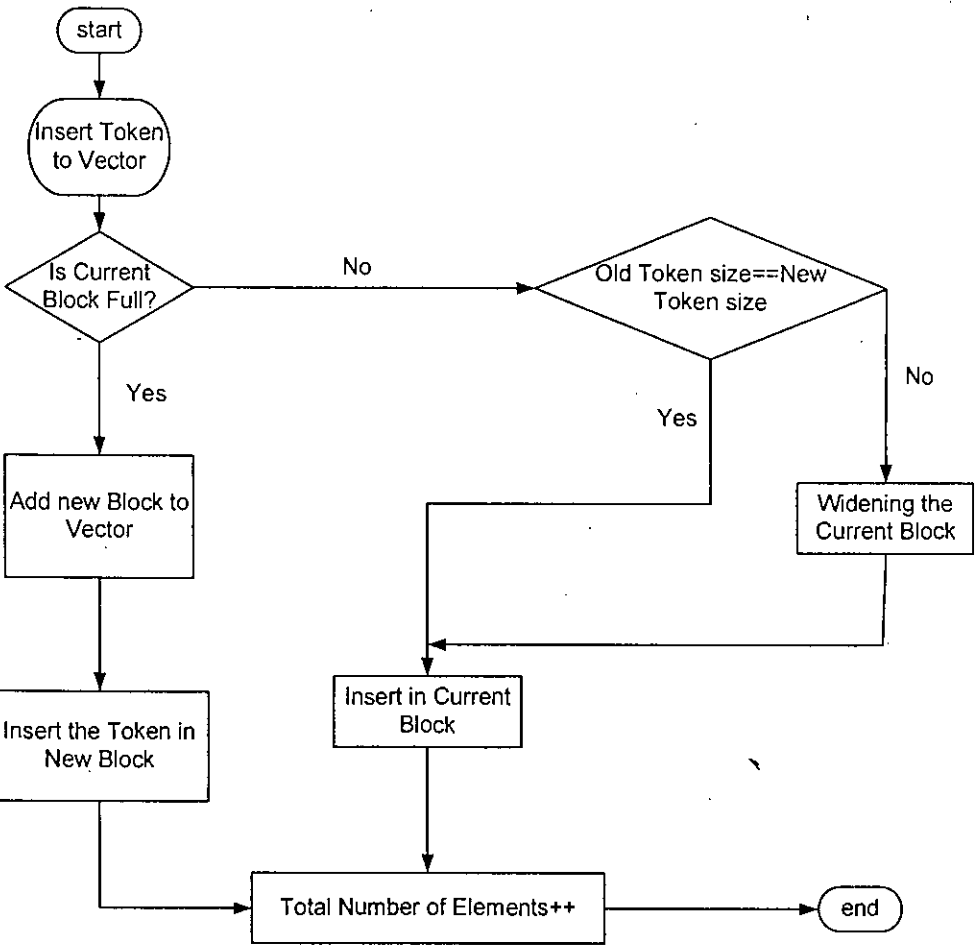

start
Insert Token to Vector
Is Current Block Full?
No
Old Token size==New Token size
Yes
No
Yes
Add new Block to Vector
Widening the Current Block
Insert in Current Block
Insert the Token in New Block
Total Number of Elements++
end

Figure 3.4: Flow Chart of the Insertion Operation in Dynamic Vector

ii) **get token(i):** Find the ith element in the vector. At first identify the block where the ith element is. Then just issue get token(block_tuple_id) operation to that block in following way:

```
get_token(i)
begin
        block_to_be_search ← i DIV number_of_element_per_block;
                    // Identify the block number
        block_tuple_id ←i MOD  number_of_element_per_block;
             // Identify the tuple id in the specified block
        vector_block_location ← block_to_be_search  MOD
        total_number_of_block_in_vector;
             //Block location of vector in memory
        if (block_to_be_search<> database_block_number)
             vector_block_location ← disk_read   (block_to_be_search)
        else
             search in vector;
        token← block[vector_block_location].get_token(block_tuple_id);
        return(token);
        endif;
end get_token;
```

iii) **get tuple id(token):** This function takes a token as an argument and search all the blocks one after another to find the matching token in the blocks and return the tuple ids.

3.3 Terabyte Data Management System Using Parallel CMBVS

This section describes an overview of the terabyte data management system using a parallel implementation of CMBVS structure. The conventional uncompressed data management system is slower in processing and in some cases unable to handle massive data sets.

There are two main assumptions about the architecture:

- Database applications where mainly relational data is stored.
- Shared nothing parallel database architecture.

3.3.1 An Overview of the Architecture

The main components of the architecture are as follows:

- Input Manager
- Terabyte Data Management System
- Compression Manager
- CMBVS Server Manager
- Column Servers
- Domain Dictionary Server Manager
- Domain Servers
- Query Compressor and Optimizer
- Decompression Manager
- Query Manager

The architecture is given in Figure 3.5. The details of each of the components of the architecture are described in the following sections.

3.3.2 Detailed Architecture of the Terabyte Data Management System

The Architecture provides an infrastructure to manage terabyte level of data in compressed form. The following sections describe the components of the architecture.

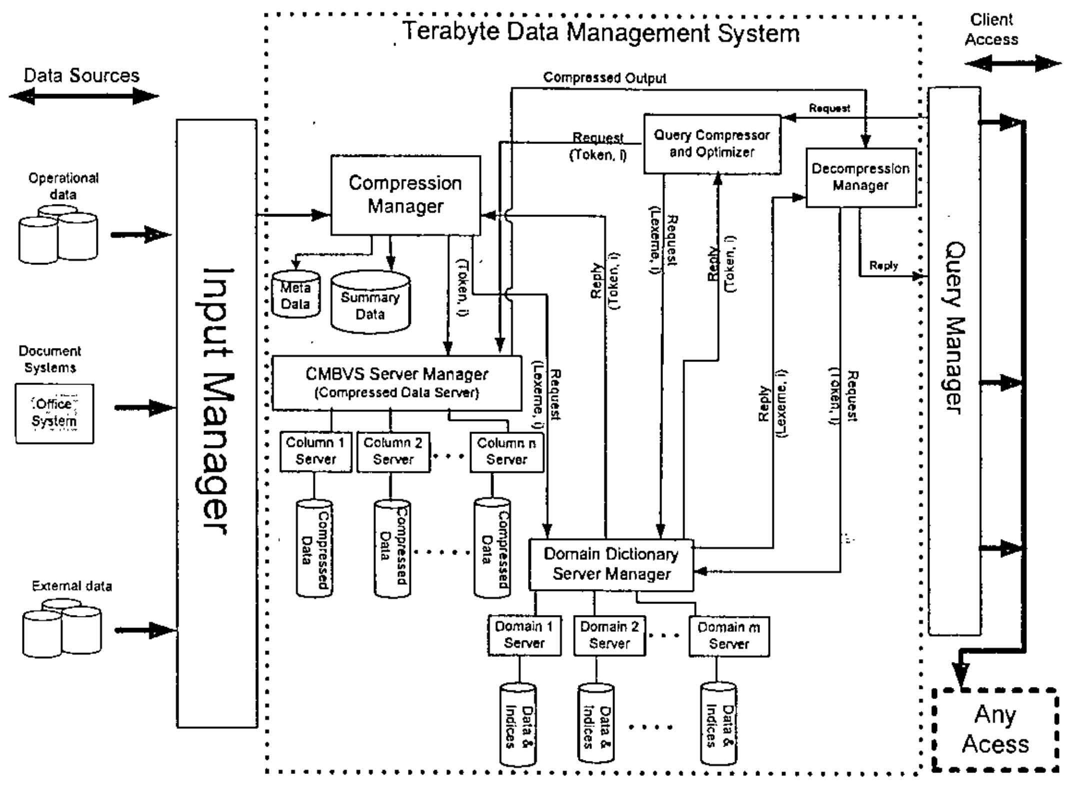


Figure 3.5: The Architecture of Terabyte Data Management System

3.3.2.1 Input Manager

This component is responsible to take input from various sources for storing data in compressed form. The architecture stores data in most homogenous and fixed format. It is responsible to process the data come from various sources and send to the compression manager for storing in compressed form. All the data must go through the input manager to marshal in a format that supported by the compression manager.

3.3.2.2 Compression Manager

The compression manager takes input from Input Manager. Taking the original data as input (lexeme) it issues a dictionary search and return the corresponding token. The Compression Manager then sends the token to CMBVS Manager to store in compressed form.

3.3.2.3 CMBVS Server Manager and Data Storage into Column Server

The CMBVS Manager is the middle level decision making unit (Figure 3.6). It distributes the column vectors over multiple column servers. It receives input from compression manager and route the compressed token to a specified Column Server.

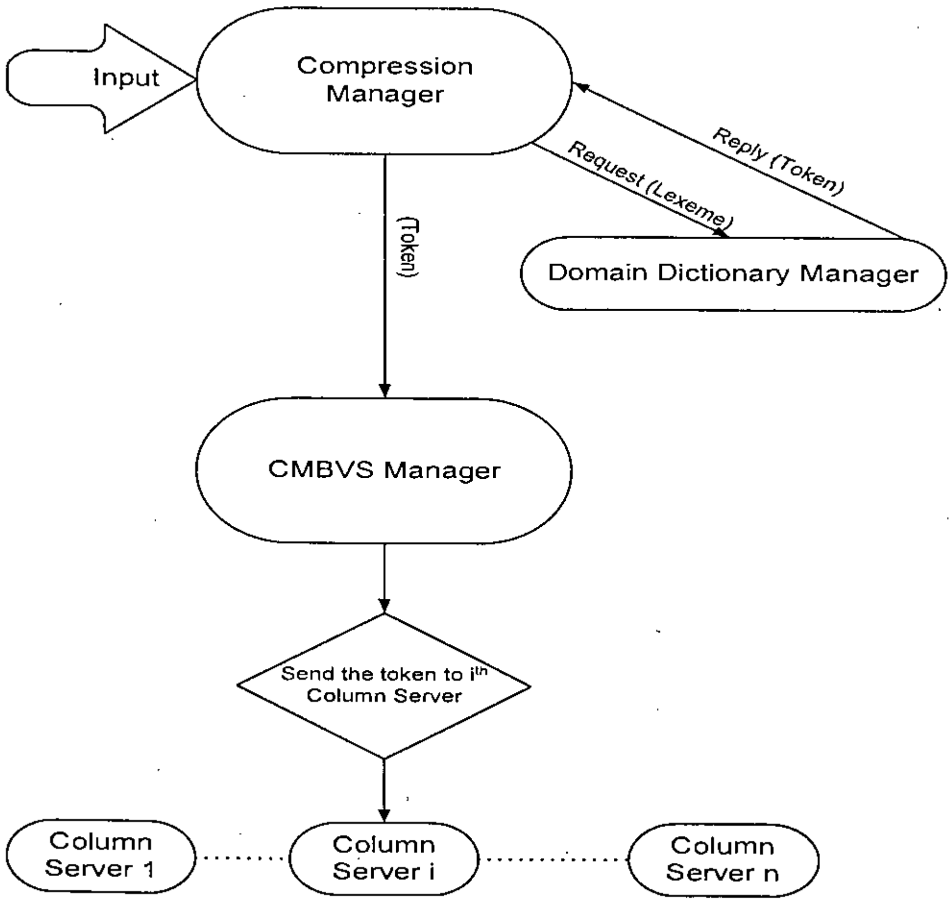


Figure 3.6: Data Insertion into Column Server

CMBVS Manager is also responsible for replying any query come from query processor. CMBVS Manager accumulates the compressed query results from Column Severs and reply the compressed output to Decompression Manager.

3.3.2.4 Column Servers

Column servers are the storage area of the compressed form of data. It takes the compressed token of each attribute from CMBVS manager. Each Column Server can store and manage one or more column vectors. Column vector is the lowest layer of the compressed data phase. Each column vector is partitioned into multiple blocks; each block contains a fixed number of elements. Detail of the block structure is described in section 3.2.1.

3.3.2.5 Decompression Manager

Decompression manager shown in Figure 3.7 does the following works:

It gets the compressed form of query results from the CMBVS Manager. This compressed data then decoded with help of Domain Dictionary Manager, the original decompressed results are finally returned to Query Manager.

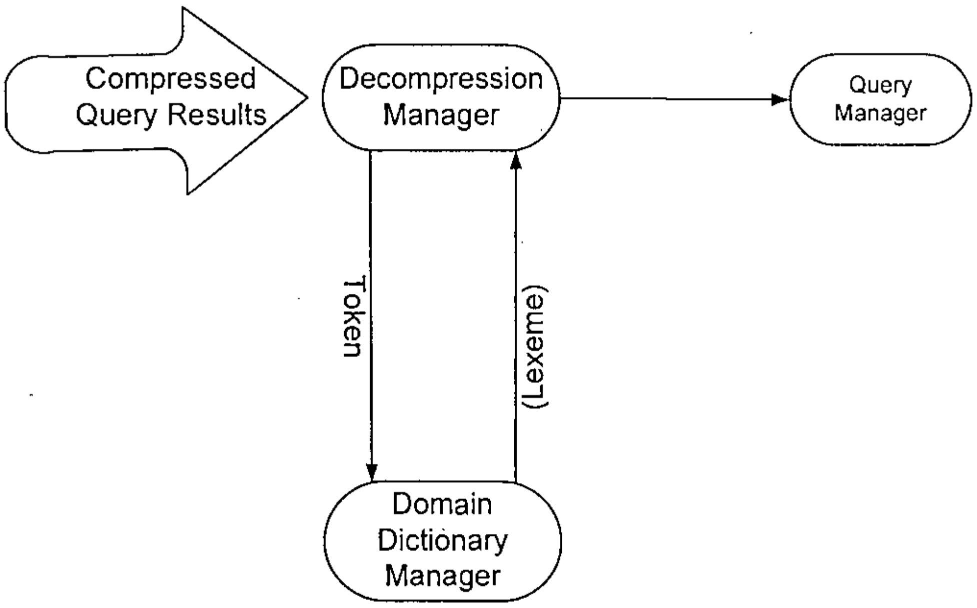


Figure 3.7: Decompression Manager

3.3.2.6 Domain Dictionary Server Manager

Domain Dictionary Server Manager distributes the domain dictionary over multiple domain servers. The main operations of a dictionary are as follows:

i) **insert (lexeme):** This operation insert the "lexeme' to a domain server. The dictionary manager decides the domain server where the domain is located.

ii) **search_token (lexeme):** This operation is propagated to the domain to find out the corresponding token associated to "lexeme".

iii) **search_lexeme (token):** This operation is propagated to ith domain to find out the corresponding "lexeme" associated to the "token". This is an inverse operation of *search_token* operation.

iv) **delete (lexeme):** When a user wants to delete a tuple value completely from the database then at first the corresponding token of the lexeme is deleted. Then the dictionary issue a *delete(lexeme)* operation if no other token exists corresponding to the lexeme.

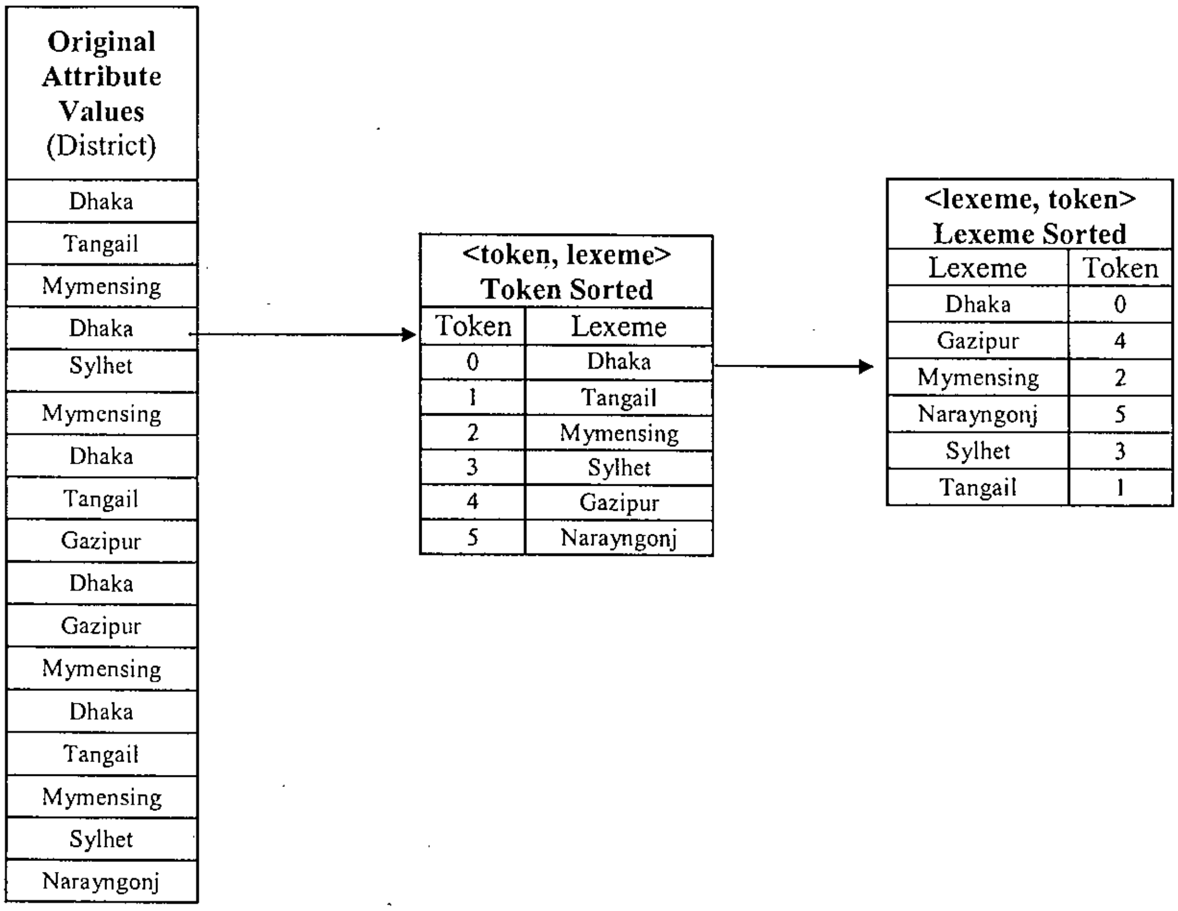
3.3.2.7 Domain Server

Each domain server stores one or more domain dictionaries. Domain server also stores the index file if the attribute is a search key. We store {lexeme, token} pair in a domain server. Though there is some storage wastage we store two instances of {lexeme, token} pair (Figure 3.8):

- **<lexeme, token>:** This instance store the lexeme in sorted order. It performs better during compression process.
- **<token, lexeme>:** This instance is sorted against token. It performs better during decompression.

Multipart Concept in Domain Dictionary:

During evolution of database the size of domain dictionary may be large enough to handle as a whole. So that we make partition to the domain dictionary and accommodate some partitions to memory and remaining others are in disk. They will be fetched on demand basis.



Original Attribute Values (District)
Dhaka
Tangail
Mymensing
Dhaka
Sylhet
Mymensing
Dhaka
Tangail
Gazipur
Dhaka
Gazipur
Mymensing
Dhaka
Tangail
Mymensing
Sylhet
Narayngonj

<token, lexeme> Token Sorted

Token	Lexeme
0	Dhaka
1	Tangail
2	Mymensing
3	Sylhet
4	Gazipur
5	Narayngonj

<lexeme, token> Lexeme Sorted

Lexeme	Token
Dhaka	0
Gazipur	4
Mymensing	2
Narayngonj	5
Sylhet	3
Tangail	1

Figure 3.8: Two Instance of Dictionary Storage.

3.3.2.8 Index Structure

An index file is created if the attribute is a search key. The index structure is associated to <lexeme, token> instance. We use the strategy < Search-Key, Row-Id List> for indexing. If Row-Id list is very large in size, we can apply multi-block vector structure algorithm to store the list.

3.4 Analysis of CMBVS

Element to Word Mapping:

There are two possibilities of element to word mapping:

- ❑ Map the bit string contiguously to the underlying word units (overlapping elements).

- Map the elements of the vector to words in such a way that no vector element ever overlaps a word boundary of underlying hardware (non-overlapping elements).

The second option appears to first sight to have a lower level of complexity since element will be sought only within one hardware defined word. It does, however, suffer from wastage of space at the word boundary. Depending on the number of elements in the block, there is another space overhead at the boundary of the block. With overlapping word boundaries, no space is wasted at the word boundaries. But there is still wastage of space at the block boundaries. This wastage ranges from 0 to a maximum of (wordSize-1).

The Three Layer Model evaluates that non-overlapping mapping technique is more efficient than overlapping in context of retrieval performance [18]. So we use the non-overlapping technique for our thesis.

Space Efficiency of Non-overlapping Element Mapping:

Let

n = total number of elements in the vector

m = total number of elements in a block

v_i = width of each element in the ith block (bits)

w = word size (bits)

Total number of blocks in the vector = $\lceil n/m \rceil$

For block i where, $1 \leq i \leq \lceil n/m \rceil$

The number of elements per word $= \lfloor w/v_i \rfloor$

The number of words per block $= \lceil (m/\lfloor w/v_i \rfloor) \rceil$

The size of the block,

$$S_i = w\,(\lceil (m/\lfloor w/v_i \rfloor)\rceil) \tag{3.1}$$

Hence the size of the vector,

$$S = \sum_{i=1}^{\lceil n/m \rceil} w\,(\lceil (m/\lfloor w/v_i \rfloor)\rceil) \tag{3.2}$$

When word size is a multiple of the element size, the percentage of space overhead is zero. The maximum wastage in the case of 64-bit word size is for a 33-bit element with almost half the space wasted.

Equation 3.2 shows the space needed for a vector. This represents the logical space. Physically the vector storage also includes the block overhead. The disk space occupied is higher than the physical storage because there are some fragmentations between disk blocks.

Insertion Time Complexity:

Insertion time includes the time of widening operation of a block, the time to store the block into secondary storage (disk) and time dictionary search time.

Time for widening operation

$$O\left(\left\lceil \frac{n}{m} \right\rceil \times U\right)$$

Where

n=Number of elements in the vector

m=Number of elements in a block

U=The number of widening operation in a block

In worst case $U = \log_2 m$

Time to store the blocks into disk

$$= O\left(\left\lceil \frac{n}{m} \right\rceil \times T_s\right) \tag{3.3}$$

Where

T_s=The average time to write a block in disk

Time to insert the lexemes into dictionary

$$O(C_i \times L_i)$$

Where

C_i =Cardinality of ith domain

L_i=Average length of lexemes in the dictionary

Insertion time complexity

$$= O\left(\left\lceil \frac{n}{m} \right\rceil \times \log_2^m\right) + O\left(\left\lceil \frac{n}{m} \right\rceil \times T_S\right) + O(C_i \times L_i) \tag{3.4}$$

Widening Operation:

It requires an additional temporary storage because the existing elements need to add to temporary block with new element size. After the completion of widening the storage will be released.

Temporary storage needed for widening operation:

$$S_w = O(w(\lceil m/\lfloor w/v_i \rfloor \rceil)) \quad (3.5)$$

Where

w=Size of word (bits)

v_i=Element size of the ith block (bits)

Searching in Vector:

There are two types of searching in vector:

Finding the token value of specified tuple_id done by the function get_token(i). This operation is a faster operation, because the vector can easily find out the token of a given tuple id. The function finds out the token after three steps:

- ❑ Finding the desired block where the tuple is located.
- ❑ Find the desired word.
- ❑ Find the desired element in the word using few basic CPU operations (SHIFT, AND, OR).

The time complexity depends on the element search in word. First two steps takes 2 CPU cycles. Complexity of third step depends on the number of elements per word.

So time complexity of get_token:

$$= O\left(\left\lfloor \frac{w}{v_i} \right\rfloor\right) \quad (3.6)$$

Finding the all tuple_ids in the vector whose value is matched to a specified token. This operation is complex operation and takes more time than the above. The given token is compared to every element of each block of whole vector. It will return a list of tuple ids. To perform the operation vector executes Get_tuple_id of each block. It may return all the tuple ids if the whole block contains similar value. So to hold the tuple id the operation needs a storage of $O(m)$. The get_tuple_id actually issue get_token(i) where i=0....m, i.e. get_token is performed m times.

Time complexity of get_tuple_id:

$$= O\left(m \times \left\lfloor \frac{w}{v_i} \right\rfloor\right) + O\left(\left\lceil \frac{n}{m} \right\rceil \times T_r\right) \qquad (3.7)$$

Where

T_r=Average time to read a block from disk

Analysis of Domain Dictionary:

Space Complexity:

Let

C_i = Cardinality of ith domain

L_i = Average length of lexeme of ith domain

Total storage of lexeme and token of ith domain

$= C_i \times L_i + 4C_i$; if token is stored as 4 byte integer

As we store two instance of domain dictionary. So total Domain Dictionary Storage(S_{DD}):

$$S_{DD} = 2 \times \sum_{i=1}^{p} \left(C_i \times L_i + 4C_i \right) \text{ ; for p attributes}$$

If the size of index file storage is= I

The Dictionary and Index Storage (S_{DI}):

$$S_{DI} = 2 \times \sum_{i=1}^{p} \left(C_i \times L_i + 4C_i \right) + I \text{ bits} \qquad (3.8)$$

Time Complexity of Dictionary Search:

Searching time in dictionary:
To search in dictionary we can easily apply hash searching technique.
Searching time in Dictionary

$$T_D = O(1) + O\left(\frac{l}{2}\right) \text{ ; where } l\text{= average length of bucket chaining} \qquad (3.9)$$

3.4.1 Analysis of the CMBVS Storage Capacity

This section is analytical study between conventional data warehouse and the proposed architecture:

Let a table T with p attributes:

$p =$ Number of attributes = Number of columns

$d =$ Distinct Domain Dictionary

As some similar attributes share a dictionary,

$d \leq p$

Total Size of Compressed Relation:

S_{CR} = number of attributes × Size of vector

$= p \times S$

$$= p \times \sum_{i=1}^{\lceil n/m \rceil} w \left(\left\lceil \left(m / \lfloor w/v_i \rfloor \right) \right\rceil \right) \text{ bits} \tag{3.10}$$

Since, $S = \sum_{i=1}^{\lceil n/m \rceil} w \left(\left\lceil \left(m / \lfloor w/v_i \rfloor \right) \right\rceil \right)$; Using equation 3.2

Total Size of Compressed Information (S_{CI})

$$S_{CI} = (p \times S) + 2 \times \sum_{i=1}^{p} \left(C_i \times L_i + 4C_i \right) + I \tag{3.11}$$

Storage for Row-wise Format:

Let n= number of tuple

x=size of each tuple

Total Storage of Uncompressed Format (S_{UF}):

$S_{UF} = n \times x$ bytes

Compression Factor (CF):

Using the Equation 3.8, 3.10 and 3.11

$$CF = \frac{S_{UF}}{S_{CI}} = \frac{S_{UF}^{'}}{S_{CR} + S_{DI}} = \frac{n \times x \times 8}{\left((p \times S) + 2 \times \sum_{i=1}^{p} \left(C_i \times L_i + 4C_i \right) + I \right)} \qquad (3.12)$$

If we estimate that the dictionary and index file is about 30% of total storage, then

$S_{DI}=0.3 \times S_{CI}$

$S_{CR}=0.7 \times S_{CI}$

Hence $\qquad S_{DI}=0.42 \times S_{CR}$

$$CF = \frac{S_{UF}}{S_{CI}} = \frac{S_{UF}}{S_{CR} + S_{DI}} = \frac{S_{UF}}{S_{CR} + 0.42\, S_{CR}} = \frac{S_{UF}}{1.42\, S_{CR}} \qquad (3.13)$$

$S_{UF} = CF \times 1.42 \times S_{CR}$

If the compressed relation is 40 GB and CF=20 then

Total uncompressed information is

$S_{UF} = 20 \times 1.42 \times 40GB = 1136GB = 1.109TB$

i.e. A simple house hold PC can handle a terabyte of information in compressed form.

3.4.2 Mathematical Derivation of Storage of Terabyte Data Management System

If the column vectors are distributed to **q** column servers and domain dictionaries are distributed to **d** domain servers

Total Storage of Column Server (S_{CS})

$S_{CS}=q \times S_{CR}$

We assume that dictionary and indices are approximately 30% of total data

So total size of the database (S_{DB}):

$$S_{DB}=CF \times (q \times S_{CR} + 0.42\, (q \times S_{CR})) =1.42 \times CF \times (q \times S_{CR}) \qquad (3.14)$$

If each column server has an array of 10 disks each 80GB, total disk capacity is 800GB for each server.

If consider there are 5% storage overhead in each server,

total usable storage capacity= 760GB/column server.

Hence using Equation 3.14

$$S_{DB} = 1.42 \times CF \times q \times 760GB = 1079.2 \times CF \times q$$

if CF=10~25 and q=10 then

$$S_{DB} = 105.39TB \sim 263.48TB$$

This amount is sufficient for any current terabyte data management application.

As we assume that the size of dictionary and indices are 42% of the size of compressed vector (and 30% of total information). So to manage such amount of information we need 0.42×q Domain Dictionary Servers with same configuration of column server. That is 5 domain servers and each has an array disk of total size is 800GB.

3.4.3 Analysis of Query Time

Query time for conventional row-wise format (SQL Server, Oracle 8 and other Conventional Database System):

Consider the following query:

SELECT A_1, ….. , A_z FROM Table[0] WHERE A_2=”XXX”;

The query is accessing **z** attributes among p attributes.

Number of disk access:

$$DA_1 = \frac{n \times x}{8K} \text{ ; If a disk page is 8KB in size.} \qquad (3.15)$$

Number of CPU comparisons:

Most of the time in uncompressed format use string comparison, which is slower than integer comparison, because a string comparison is multiple integer comparisons.

Let average speed-up for Integer Comparison =i_c

Query time for Compressed Column-Wise Format (T_{CR}):

Query time includes:

- ❑ Time to find the token value of search key from dictionary (T_D)
- ❑ Time of vector search in compressed form (T_V)
- ❑ Time of decompression (T_{DECOMP})

Dictionary Access Time (T_D):

Time taken for Coding and Decoding from Domain Dictionary depends on cardinality of the Domain Dictionary. If we apply hash technique search on domain dictionaries then $T_{D(i)}$ is the time of ith domain (using equation 3.9) :

$$T_{D(i)} = O(1) + O\left(\frac{l}{2}\right)$$, where l is the average length of bucket chain

Total Dictionary Search Time (T_D):

$$T_D = \sum_{i=1}^{p} T_{D(i)} \tag{3.16}$$

Vector search time (T_v) is summation of individual vector search time ($T_{V(i)}$):

$$T_V = \sum_{i=1}^{p} T_{V(i)} \tag{3.17}$$

Decompression time (T_{DECOMP}) includes the time of decompression of individual domain ($T_{DECOMP(i)}$):

$$T_{DECOMP} = \sum_{i=1}^{p} T_{DECOMP(i)} \tag{3.18}$$

Total Query Time in Single Processor System:

Combining the equations 3.16, 3.17 1 and 3.18 we find the following time equation for single processor system :

$$T_{Server(1)} = T_D + T_V + T_{DECOMP} \tag{3.19}$$

Total Query Time in Multi Processor System:

$$T_{SERVER(n)} = Max(T_{D(i)}) + Max\left(Max(T_{V(i)}), Max(T_{DECOMP(i)})\right) \qquad (3.20)$$

Comparison of Query time between Uncompressed and Compressed format:

Number of disk access for accessing vectors:

$$DA_2 = \frac{\frac{z}{p} \times (S)}{8K} = \frac{z \times S}{p \times 8K} \qquad (3.21)$$

Where

z=Number of attributes are accessed in the query

S=Vector size

p=Total attributes

The disk block size= 8K

Total Speed-Up:

=Speed-up for Integer Comparison+ Speed-Up in Disk-Access

$$= \frac{n \times i_c}{n} + \frac{DA_1}{DA_2 + T_D}$$

$$\approx i_c + \frac{DA_1}{DA_2}$$; Ignoring dictionary access time due to hash search

$$\approx i_c + \frac{n \times x \times p}{z \times S} \qquad (3.22)$$

Now let

i_c=5, p=11, z=4, n×x=200MB, S=10MB

Total speed-up using the equation 3.22:

$$SU = 5 + 20 \times \tfrac{11}{4} = 60$$

3.4.4 Complexity of Joining Operations

The term join refers to the form $r \times_{r.A=s.B} s$, where A and B are attributes or set of attributes of relations **r** and s respectively. The relation r is called outer relation and the relation s is called outer relation. Let the number of tuples in the outer relation is $\mathbf{n_r}$ and the number of tuples in the inner relation is $\mathbf{n_s}$

i) **Block nested-loop join in CMBVS:** Every block of the inner relation is paired with every block of outer relation. Within each pair of blocks, every tuple in one block is paired with every tuple in the other blocks. To generate all pairs of tuples that satisfy the join condition are added to the result. The time complexity of block access in CMBVS is $O\left(\left\lceil \frac{n_r}{m} \right\rceil \times \left\lceil \frac{n_s}{m} \right\rceil + \left\lceil \frac{n_r}{m} \right\rceil\right)$ whereas in uncompressed format requires at least $O\left(CF_1 \times \left(\left\lceil \frac{n_r}{m} \right\rceil \times \left(CF_2 \times \left\lceil \frac{n_s}{m} \right\rceil\right) + \left\lceil \frac{n_r}{m} \right\rceil\right)\right)$, where $\mathrm{CF_1}$ is compression factor of relation **r** and $\mathrm{CF_2}$ is compression factor relation s.

ii) **Merge join in CMBVS:** In merge join both the relations must be sorted. Each sorting requires $O(n\log n)$ time complexity. If both the relations are sorted on join attributes, the time require to access the blocks in CMBVS is $O\left(\left\lceil \frac{n_r}{m} \right\rceil + \left\lceil \frac{n_s}{m} \right\rceil\right)$, whereas uncompressed format requires the time to access the blocks is $O\left(CF_1 \times \left\lceil \frac{n_r}{m} \right\rceil + CF_2 \times \left\lceil \frac{n_s}{m} \right\rceil\right)$. To make merge join, it requires the relation s to fit in memory. For large database it is not always possible. We may use hybrid merge join. It uses indexing on join attributes.

iii) **Indexed nested-loop join in CMBVS:** If an index available on join attributes of relation s, index lookups can be replace file scans. For each tuple $\mathbf{t_r}$ in **r** the index of s is used to look up tuples in s that satisfy the join condition with tuple $\mathbf{t_r}$. The cost of the join operation in CMBVS is $O\left(\left\lceil \frac{n_r}{m} \right\rceil + n_r \times c\right)$, where c is a constant time cost of selection on s using join condition. The uncompressed form need $O\left(CF_1 \times \left\lceil \frac{n_r}{m} \right\rceil + n_r \times c\right)$ time complexity if $\mathrm{CF_1}$ is the compression factor of relation **r**.

In CMBVS each of result need to decompress, so the time of decompression is added to each type of join operations. However CMBVS provides cheaper join operation due to compression. As it needs fewer number of I/O access than that of uncompressed format.

3.5 Comparison with other Systems

Comparison with Tera-Scale Scientific Data: The tera-scale data warehouse in [14] designed for the scientific domain. This architecture also uses compression in file level. The compressed data file is stored in web server but the query can not be processed directly on compressed form. The SQL command is executed in the decompressed form. The query time obviously will be larger than our compressed data warehouse because our architecture can process the query in compressed form. Our architecture is applicable to wide variety of business or scientific relational data.

Comparison with DBMS on the Microsoft SQL Server 2000 Platform: The platform is housing terabyte of data in conventional manner. The hardware is multiprocessing environment with array of disk. Nevertheless, our system can handle terabyte of compressed data simply in a household PC, attached with 80 to 100GB of disk space. If we use array of disk in our proposed architecture, it can handle several hundreds of terabyte of data.

Comparison with Oracle Table Compression:

Let total storage in uncompressed format:

$$S_{UF} = n \times x$$

Average compression factor in Oracle is ≈3.11

Total storage in Oracle Compressed Data (S_{OD}):

$$S_{OD} = \frac{n \times x}{3.11} = \frac{S_{UF}}{3.11} = \frac{1.42 \times CF \times S_{CR}}{3.11}$$

$$\frac{S_{OD}}{S_{CR}} = 0.46 \times CF \text{ hence } S_{OD} = 0.46 \times CF \times S_{CR}$$

if Compression Factor (CF) is 20 then $S_{OD}=9.2 \times S_{CR}$

Since Oracle applies compression in block level and maintain a single dictionary per block, which incurs some redundancy. But our system use individual dictionary for each attribute.

3.6 Summary

The terabyte data management system is really an attractive one in comparison with any existing data management system powered by SQL Server or Oracle or other conventional database system. The compression model in our architecture is better than compression in Oracle, because Oracle applies block level compression that has some redundancy [25]. The Tera-scale architecture [14] also compresses data file but can not applies query in compressed form, so query time obviously is larger than our architecture. The query throughput in our proposed architecture may be affected by the communication between components of the Terabyte DBMS but if we use high speed, reliable optic fiber network this degradation will be reduced.

Chapter 4
Results and Discussions

The objective of the experimental work is to verify the feasibility and scalability of the design of multi-block vector, the dictionary and data model of the architecture. The experimental evaluation has been performed with large synthetic data. The storage and retrieval (query) time is compared with widely used Microsoft SQL Server 2000. Our target was to handle a large table and justify the storage trends and query time in comparison with SQL Server. Throughout the chapter we call our system "CMBVS" for Columnar Multi-Block Vector Structure.

4.1 Experimental Setup

The structure of the CMBVS model and the domain dictionaries have been implemented over Borland C++ 5.02 compiler (a 32 bit compiler) running on 1.6GHz Pentium IV processor with 256MB memory. The operating system is Microsoft XP. The results are compared with Microsoft SQL Server 2000 on the same hardware configuration. The Operating System is Microsoft Windows 2000 Professional.

4.2 The Data Sets

We have chosen the Electoral Database of Bangladesh. The voter information is the largest relation in the database. We consider approximately 85 million voters in Bangladesh. We assume eleven attributes in the relation that are given in the Table 4.1. Each tuple is estimated as 255 bytes. The estimated size of the relation is $8.5 \times 10^6 \times 255$ bytes i.e. 20.19GB.

Table 4.1: The Attributes and Length in Bytes

Attribute Name	Data Type	Length in bytes
Voter Id	Integer	4
Dist	Varchar	30
Thana	Varchar	30
First Name	Varchar	30
Last Name	Varchar	30
Father First Name	Varchar	30
Father Last Name	Varchar	30
Sex	Char	1
Marital Status	Varchar	10
Union Parishad	Varchar	30
Village	Varchar	30
Total		**255 bytes**

4.2.1 Synthetic Data Generator

It is quite a hard task to gather the real data sets for the voter relation. We, instead deign a data generator that generates the data that resembles the real data sets for voter relation. We assume an approximate cardinality of the domain values of each attribute is given in the Table 4.2.

To create synthetic data, we generate a random number within a value equal to the domain cardinality of an attribute and write the number in a text file with a fixed width. Each line of the text file contains a tuple of a voter relation.

The input manager load the text file to extract raw data and read each attribute value from the file and send the value to compression manager.

Table 4.2: The Approximate Cardinality of Attribute Domain Values of Voter Relation

Sl. No.	**Attribute Name**	**Approximate Cardinality of Domain**	**Bits Required to Represent**
0	Voter Id	Primary key	32
1	Dist.	64	6
2	Thana	507	9
3	First Name	5000	13
4	Last Name	5000	13
5	Father First Name	5000	13
6	Father Last Name	5000	13
7	Sex	2	1
8	Marital Status	4	2
9	Union Parishad	5007	13
10	Village	50000	16
Total			**131 bits=131÷ 8 bytes ≈16 bytes**

4.3 Storage Performance and Compression Factor (CF)

Our data generator generates 85 million voter records. Each compressed record occupies at most 16 bytes (last column of Table 4.2), whereas each uncompressed record occupies 255 bytes. The record level compression ratio is 15.93. The gross size of 85 million tuples in uncompressed raw data is 20.19GB. We generate various number of tuples and transform them in compressed form. Various tests are performed on the compressed data. The similar data is inputted to SQL Server 2000 using its import tool.

The storage space occupied by SQL Sever and by the CMBVS shows a significant factor of compression. The SQL Server occupies more than the estimated amount of storage due to overhead.

Table 4.3: Storage Space Comparison between SQL Server and CMBVS

Number of Tuples (million)	Estimated Storage for Uncompressed Record (GB)	Actual Storage for SQL Server (Ss) (GB)	Estimated Storage for CMBVS (GB)	Actual Storage for CMBVS (Sc) (GB)	Compression Factor (CF) (Ss/Sc)
1	0.24	0.61	0.02	0.02	30.50
2	0.47	1.19	0.05	0.05	23.80
3	0.71	1.80	0.07	0.08	22.50
5	1.19	3.10	0.13	0.13	23.85
10	2.37	6.41	0.24	0.26	24.65
15	3.56	9.73	0.35	0.39	24.95
20	4.75	13.06	0.46	0.53	24.64
25	5.94	16.38	0.56	0.66	24.82
30	7.12	19.71	0.70	0.79	24.95
40	9.50	26.38	0.92	1.06	24.89
50	11.87	33.04	1.12	1.33	24.84
60	14.25	39.70	1.33	1.60	24.81
70	16.62	46.35	1.51	1.87	24.79
75	17.81	49.68	1.62	2.00	24.84
80	19.00	53.68	1.72	2.16	24.85
85	20.19	56.34	1.80	2.27	24.82

We insert up to 85 million tuples in CMBVS. For SQL Server we insert few millions of tuples and then extrapolate the result up to 85 million tuples. The SQL Sever occupies 56.34GB whereas CMBVS occupies only 2.27GB for 85 million tuples. The storage space occupied by uncompressed form and compressed form for different numbers of tuple is given in Table 4.3. The storage comparison between SQL Sever and CMBVS is shown in Figure 4.1. The average compression ratio is 24.90.

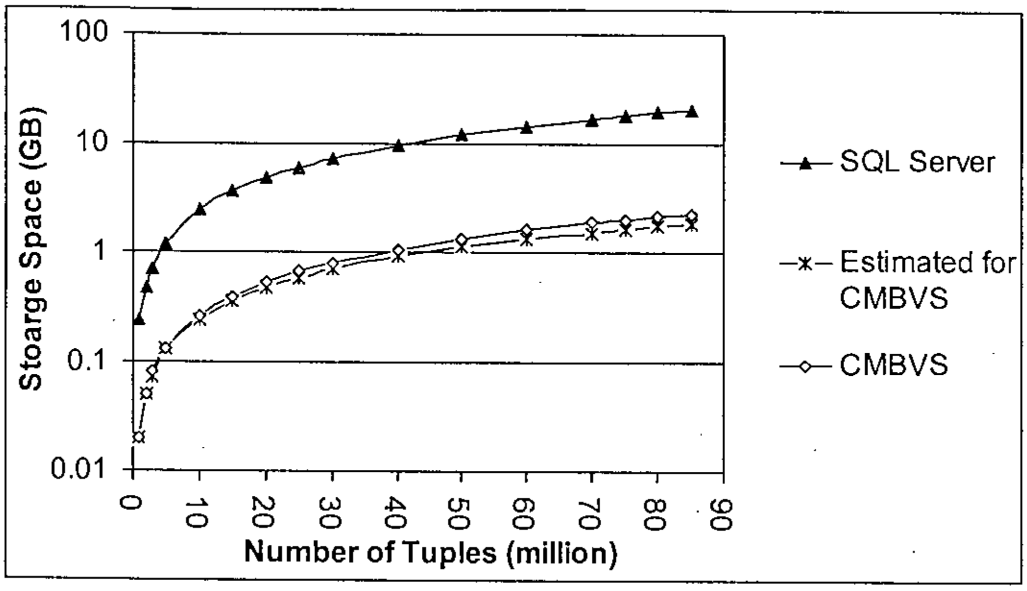


Figure 4.1: The Comparison of Storage Space between SQL Sever and CMBVS (logarithmic scale)

The Figure 4.1 shows that the storage space increases linearly as the number of tuples increase. The space occupied by CMBVS is very small in comparison with SQL Server.

The trends of storage space of dictionary and dynamic vector is depicted in Figure 4.2. The average storage space of vector is 66.98% of total storage in compressed form. On the other hand, the dictionary occupies 33.01% of total storage. For the 85 million tuples, the size of the vector is 1.52GB and the dictionary is 0.75GB. The storage estimated for dynamic vector is 1.27GB and for dictionaries is 0.5334GB. The actual storage space is more than the estimation. The extra storage for vector is needed for block overhead and dictionary partitions overhead. Extra space need for fragmentation in disk between blocks.

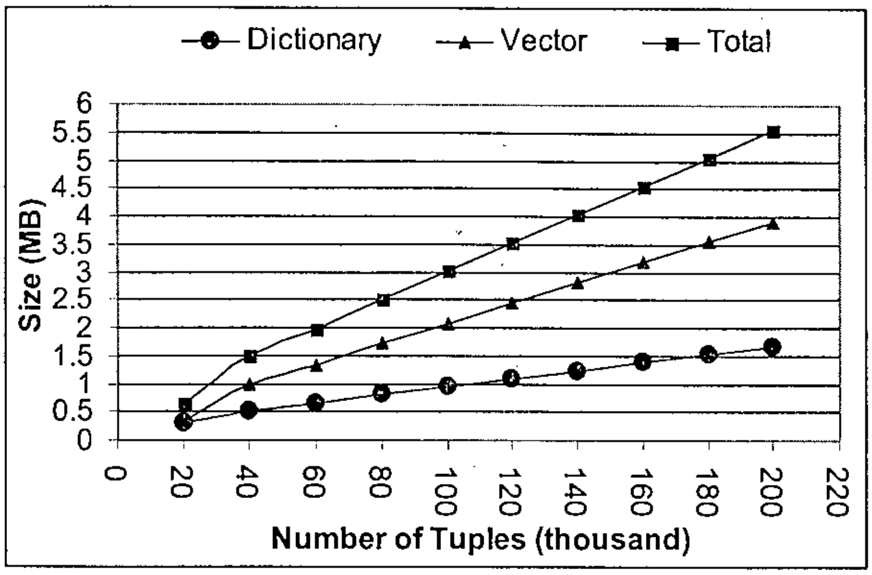


Figure 4.2: Storage Trends of Vector and Dictionary in Compressed Database

The CMBVS occupies storage space is only 4% of uncompressed counterpart in SQL Server. The CMBVS shows about 25 times better performance on storage space. This is a significant achievement of the architecture.

4.4 Insertion Time Trends

The time of insertion in CMBVS is longer than that in SQL Server. During the conversion of raw data to CMBVS includes the time of dictionary search, insertion time in dictionary, time taken for restructuring the block when the token size increases and the time needs to write the blocks in disk. The insertion process is slower than that of SQL Server. The trend of insertion time is shown in Figure 4.3.

From the Equation 3.3, the number of element per block (**m**) is fixed, average length of lexeme ($\mathbf{L_i}$) is also fixed. So **n**, the number of tuples and $\mathbf{C_i}$, the cardinality of domain dictionary is the main factor of time trends. In Figure 4.3, the number of tuple (**n**) is gradually increasing and the cardinality of dictionary ($\mathbf{C_i}$) may also increase, so the insertion time curve depends on only **n** and $\mathbf{C_i}$.

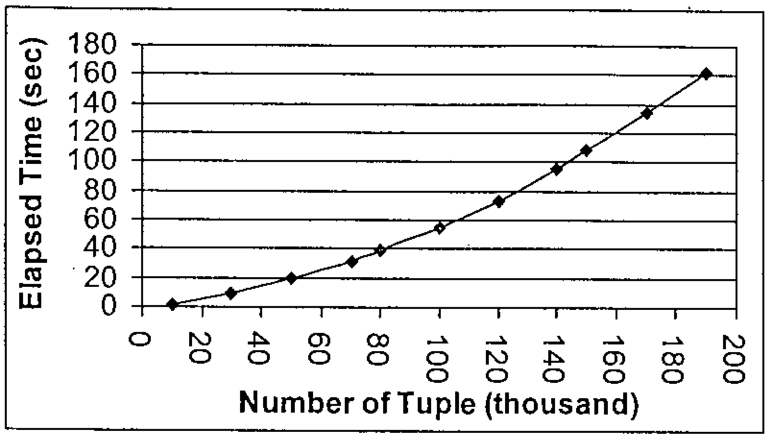


Figure 4.3: Insertion Time Trends in CMBVS for Varying Number of Tuples

4.5 Access Performance

We present here four types of access time performance:

- ❑ Time comparison of projection operation
- ❑ Time comparison of 50% selectivity
- ❑ Time comparison of single predicate select queries
- ❑ Time comparison of compound predicate select queries

4.5.1 Time Comparison of Projection Operations

Projection operation just accesses the specified attributes from database. To compare the access speed up we choose 4 queries shown in Table 4.4. Query Q1, Q2, Q3 and Q4 represents the projection operation of 1, 2, 3 and 4 attributes respectively. The average speed is 55 to 110. The Figure 4.4 shows the comparison of trends of elapsed time for projection operation. The difference of elapsed time between two trend lines is maximum at the beginning, and decreases as the number of attributes increase. The access speed decreases with number of attributes increase. Because the conventional database system accesses the entire tuple for any type of query, so the number of attributes has no significant effect in the query time. Whereas the CMBVS accesses only the information related to the attributes used in query, so that number of attributes increases and query time also increases.

Table 4.4: Comparison of Access Speed on Projection Queries

Sl. No.	SQL Command	Average Elapsed Time (sec)		Access Speed (T_s/T_c)
		SQL Server (T_s)	CMBVS (T_c)	

Q1	SELECT Ai
	FROM VoterInfoTable;
	(Single attribute projection and $1 \leq i \leq 11$)
		285	2.57	110.89

Q2	SELECT Ai , Aj
	FROM VoterInfoTable;
	(Two attribute projection, $1 \leq i, j \leq 11$ and $i \neq j$)
		319	4.04	78.96

Q3	SELECT Ai , Aj , Ak
	FROM VoterInfoTable;
	(Three attribute projection, $1 \leq i, j, k \leq 11$ and $i \neq j \neq k$)
		373	5.92	63.01

Q4	SELECT Ai , Aj , Ak, Al
	FROM VoterInfoTable;
	(Four attribute projection, $1 \leq i, j, k, l \leq 11$ and $i \neq j \neq k \neq l$)
		430	7.70	55.84

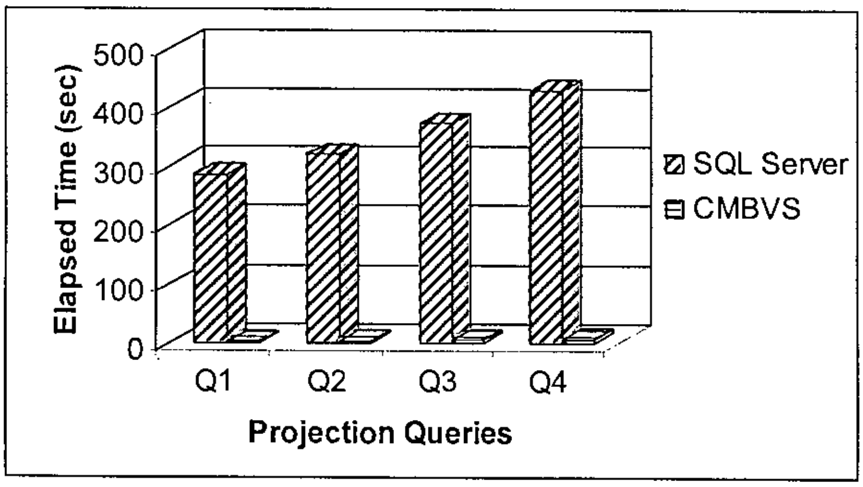


Figure 4.4: Access Speed Comparison on Project Queries

4.5.2 Time Comparison of 50% Selectivity

This query returns almost 50% of tuples from the relation. We choose the attribute "Sex". Sex has only two values (male or female). Among all the tuples about 50% is male and remaining 50% is female.

Q5: SELECT * FROM voterInfoTable WHERE Sex='X';

(Here X may be either "Male" or "Female")

The query Q5 is actually finding almost half of the tuples from the vector and decompresses them. So we can say the query Q5 as 50% selectivity. The elapsed time comparison between SQL Sever and CMBVS for the query Q5 is shown in Figure 4.5. The figure shows that the average speed ratio is 88.93. When the number of tuple is 1 million the query Q5 takes comparatively less time than for 2 or 3 million tuples. The reason is, during 1 million tuple the whole database can be accommodated in memory so the query time becomes lower.

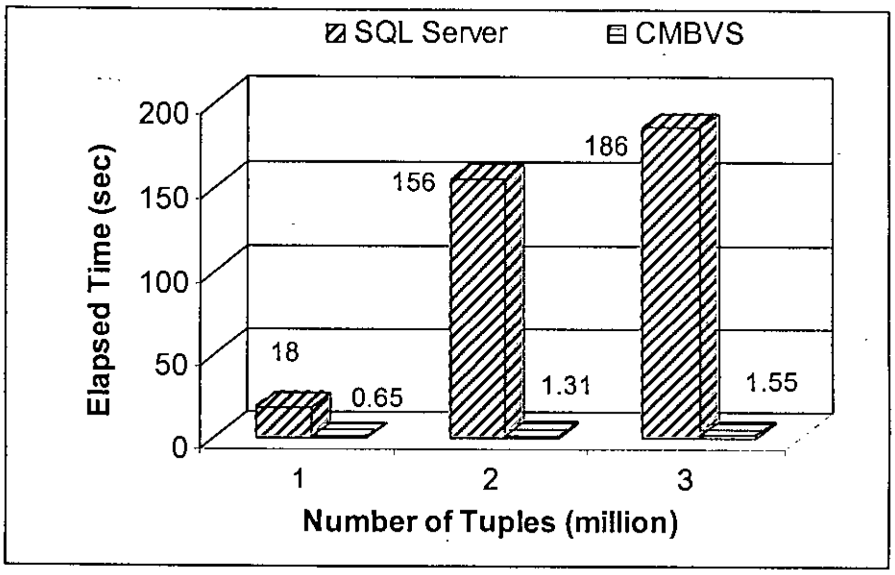


Figure 4.5: Elapsed Time Comparison for 50% Selectivity

4.5.3 Time Comparison of Single Predicate Select Query

The SQL commands that find out the complete tuple value when the query satisfy single predicate (associated to one attribute). A set of queries are listed up in the Table 4.5 that operated on 3 million tuples. Query Q6 has predicate on the primary key, the query Q7 is associated to the village in WHERE clause and returning 0.0015% of tuples, the query Q8 is associated to union and returning 0.019% of tuples, Q9 is associated to thana and shows 0.18% os selectivity and the Q10 is showing 1.54% selectivity. Though the queries in Table 4.5 are associated to one attribute in WHERE clause but access the complete tuple, the chosen queries are representative of query performance.

The Figure 4.6 shows the comparison between SQL Server and CMBVS for five queries associated to one attribute in WHERE clause. The result shows that the access performance in CMBVS is 36.23 times faster than that of SQL Server. The access time of predicated SELECT query takes more time in CMBVS than the project queries, because predicated query has to filter out the un-matching tuples, which takes significant time in large datasets. Whereas, in the projection queries do not need to filter out the result.

Table 4.5: A Set of Single Predicate Select Queries

Query No.	**SQL Command (Single attribute used in WHERE clause)**	**Average Number of Tuples Returned**	**% of Selectivity**
Q6	SELECT * FROM voterInfoTable WHERE VoterId="XXXXXX"	1	Single Tuple
Q7	SELECT * FROM voterInfoTable WHERE Village="YYYYY"	46	0.0015%
Q8	SELECT * FROM voterInfoTable WHERE Union="ZZZZZ"	599	0.019%
Q9	SELECT * FROM voterInfoTable WHERE Thana="WWWW"	5407	0.18%
Q10	SELECT * FROM voterInfoTable WHERE Dist="VVVVV"	46465	1.54%

The queries in Table 4.5 (Q6~Q10) actually accessing the value of all the attributes. The query like "SELECT * FROM voter_info_Table WHERE Ai='XXXX' " ultimately accessing all the attributes where the tuple satisfying the criteria. This is another cause of degradation of speed ratio than the project queries.

4.5.4 Time Comparison of Compound Predicate Select Queries

The compound predicate SELECT queries include multiple attributes in the WHERE clause. We take here four queries given in Table 4.6. Query Q11, Q12, Q13 and Q14 has one, two, three and four attributes respectively in the WHERE clause. We run the same query in SQL Server and CMBVS. Different ten experiments carried out and the average of all elapsed time is used in the Figure 4.7.

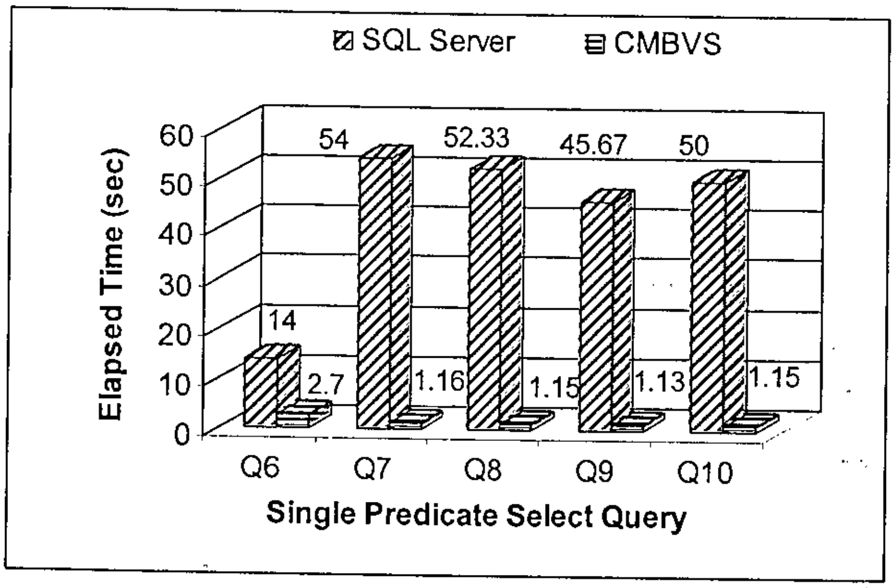


Figure 4.6: Comparison of Elapsed time for Single Predicate Select Queries

Table 4.6: List of the Compound Predicate SELECT Queries

Query No.	SQL Command
Q11	SELECT * FROM voterInfoTable WHERE Ai='XXX' ; (1≤ i ≤ 11)
Q12	SELECT * FROM voterInfoTable WHERE Ai='XXX' AND Aj='YYY'; (1≤ i, j ≤ 11 and i ≠ j)
Q13	SELECT * FROM voterInfoTable WHERE Ai='XXX' AND Aj='YYY' AND Ak='ZZZ' ; (1≤ i, j, k ≤ 11 and i ≠ j ≠ k)
Q14	SELECT * FROM voterInfoTable WHERE Ai='XXX' AND Aj='YYY' AND Ak='ZZZ' AND Al='WWW' ; (1≤ i, j, k, l ≤ 11 and i ≠ j ≠ k ≠ l)

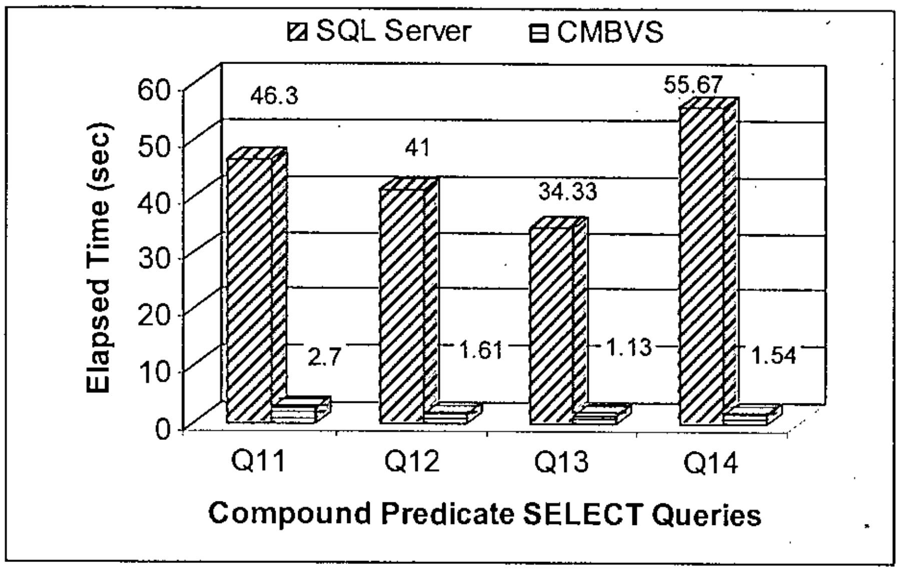


Figure 4.7: Elapsed Time Comparison of Compound Predicate SELECT Queries

The elapsed time comparison for the compound predicate select queries given in Table 4.6 is depicted in Figure 4.7. The result shows that the CMBVS is 27.29 times faster than that of SQL Server. The query optimizer optimizes compound predicate SELECT queries. The query optimizer reduces the access of number of column vectors and minimizes the retrieval latency. The secrets of the optimizer are very simple. The optimizer just decides the attributes which contains most information. Then retrieve the column vector associated to the attribute and continue searching until the last block of vector reach.

The Figure 4.8 shows the comparison of elapsed time for the query like SELECT * FROM voterInfoTable WHERE VoterId="XXXXXX" in both indexed and non-indexed form of the compressed database. The indexing reduces query time satisfactorily. The estimated time is also plotted in the figure. The estimated time trends show that it is less than experimental time. The reason is, the estimated time considers only the disk access time of dictionary and vector is taken. It also includes a fixed decompression time. We don't consider the processing time inside the block. As the database grows, the processing will take substantial amount of time.

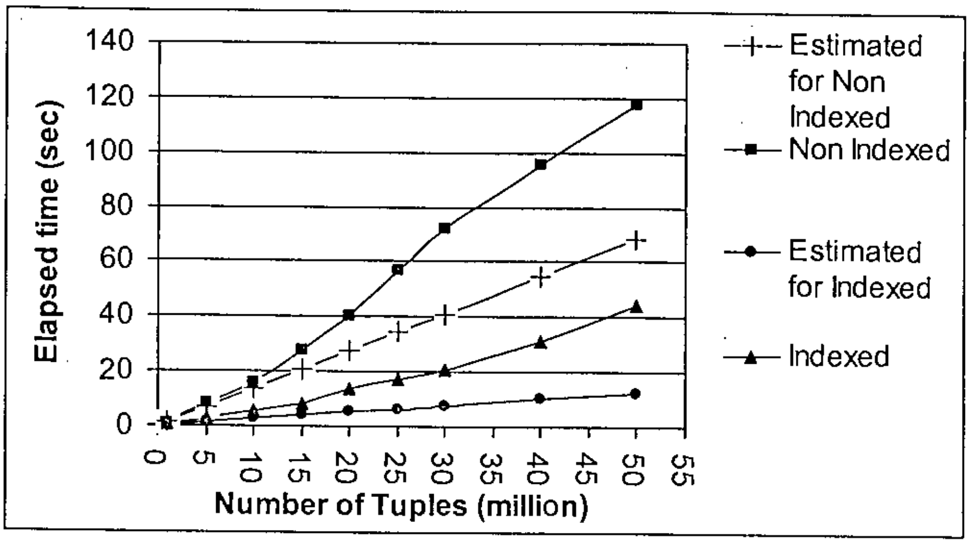


Figure 4.8: Trace of Elapsed Time on Indexed and Non-Indexed Form of Compressed Database

102870

4.6 Comparison of Terabyte Data Management System with other Existing Terabyte Architecture

The Tera-scale architecture [14] (discussed in section 2.2.4) is deployed on a dual PowerG5 processor attached with 1.5 TB RAID for management of scientific data. The average archiving rate per day is 3GB in compressed form whereas in uncompressed data stream is about 15 GB/day; that is compression ratio is about five. The system can not make query in compressed form, so the query processing is obviously slower than our architecture.

The Oracle table compression for managing large data warehouse has the compression ratio approximately 3.11. It applies compression to every block. The SQL Server 2000 platform is warehousing terabyte of data in uncompressed form [7]. The Table 4.7 shows the estimated storage space requirements in different systems to store different millions of tuples. The Figure 4.9 is the pictorial representation of the Table 4.7.

Table 4.7: The Estimated Storage Space Requirements for Different number of Tuples in Different Architecture.

Number of Tuples (million)	SQL Server 2000 Platform (TB)	Tera-Scale Architecture (TB)	Oracle Compressed Database (TB)	TDMS (TB)
10	0.0064	0.00047	0.00076	0.00026
100	0.0537	0.00474	0.00763	0.00263
300	0.1613	0.01422	0.02286	0.00648
500	0.2690	0.0237	0.03810	0.01081
1000	0.5383	0.0474	0.0762	0.02162
1500	0.8060	0.0711	0.1140	0.0324
2000	1.075	0.095	0.152	0.0432
2500	1.346	0.188	0.190	0.054
3000	1.615	0.142	0.228	0.065
4000	2.153	0.189	0.304	0.086
5000	2.962	0.237	0.381	0.1081

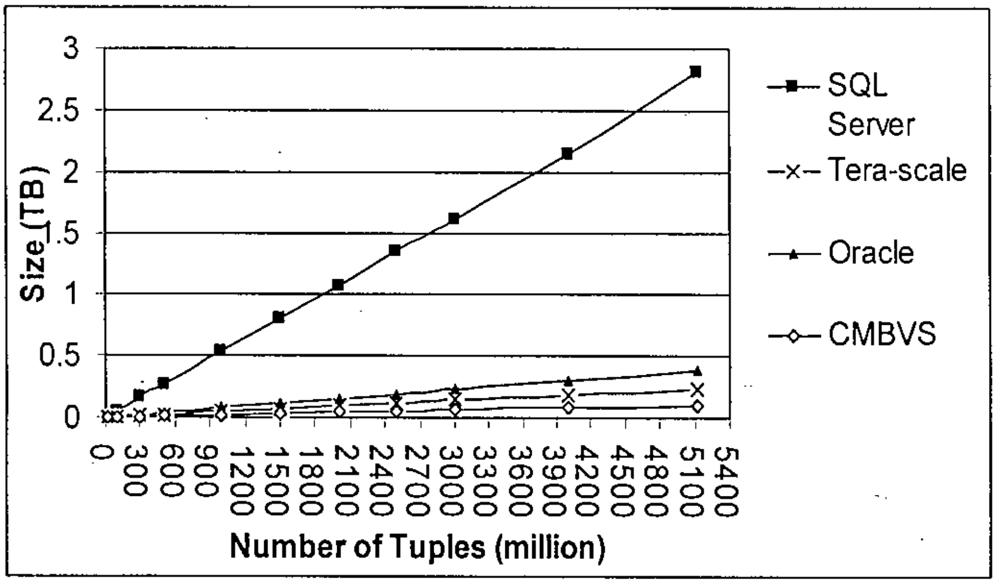


Figure 4.9: Estimated Storage Space Comparison in Different Architecture

Simulated query processing time using the CMBVS in single processor system is given in Table 4.8 and query time for parallel processor architecture is given in Table 4.9. In both the system we take the query like:

Q15: SELECT * FROM voterInfoTable WHERE DistId='ZZZZ' and Thana='YYYY'

The query Q15 finds out the voter list of a "Thana" of a particular "District". The parallel processor architecture can execute search in different dictionaries in concurrent manner, so that the total dictionary search time is the maximum time of individual domain search time. Similarly, search in compressed column vectors can be carried out simultaneously in different vectors, and total time will be the maximum time of individual column search.

The decompression of different domain can be done concurrently and are pipeline with block search in compressed vector. So the query time in parallel processor system follows the equation 3.20 and single processor system follows the equation 3.19. The Table 4.8 and 4.9 show the simulated time of query processing in single and parallel processor system respectively.

Table 4.8 : Query Time for Single Server Terabyte Data Management System Architecture

Number of tuples (million)	**Query Time in Single Processor Architecture (sec)**			
	Dictionary Search *(T_D)*	**Vector Search in Compressed Form** *(T_V)*	**Time of Decompression and Combining the Result** *(T_{DECOMP})*	**Total** *($T_{SERVER(I)}$)*
10	0.174×10^{-6}	7.52	24.82	32.34
20	0.233×10^{-6}	14.79	42.89	57.684
30	0.247×10^{-6}	27.01	86.63	113.648
40	0.254×10^{-6}	35.93	138.5	174.494

Table 4.9: Simulated Query Time for Parallel Processor System

Number of Tuples (million)	Query Time in Parallel Processor (sec)			
	Dictionary Search $Max(T_{D(i)})$	Vector Search in Compressed form $Max(T_{V(i)})$	Decompression and Combining the Result $Max(T_{DECOMP(i)})$	Total $(T_{SERVER(n)})$
10	0.090×10^{-6}	6.54	13.62	13.73
20	0.094×10^{-6}	12.92	25.33	25.93
30	0.105×10^{-6}	23.15	41.58	41.85
40	0.113×10^{-6}	30.48	56.61	56.94

The Table 4.10 shows simulated query time comparison between single processor and parallel processor system at a glance. The response time in parallel processor system is also high enough for any application. To solve this problem the compression in vertical column format is not sufficient. We need to maintain the indexing in horizontal manner and to distribute the data in horizontally fragmented way in different small TDMS systems. This will increase the parallelism of the architecture and hence the query throughput will be higher.

Table 4.10: Query Time Comparison between Single Sever and Parallel Server Architecture

Number of tuple (million)	Single Processor System (sec)	Parallel Processor System (sec)
10	32.34	13.73
20	57.684	25.93
30	113.648	41.85
40	174.494	56.94

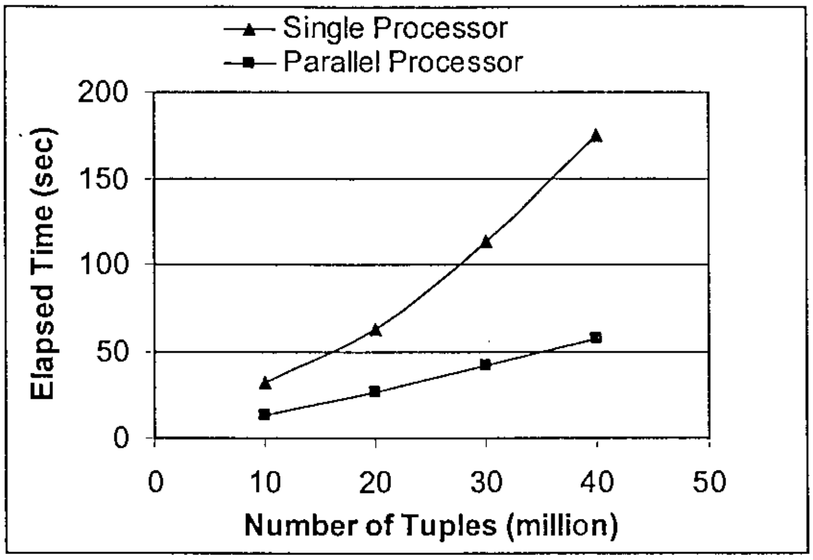


Figure 4.10: Simulated Query Time Comparison between Single and Parallel Processor System.

4.7 Summary

In this chapter we presented the experimental evaluation of the CMBVS, the main component of the Terabyte Data Management System. We evaluated mainly the storage performance which is achieved in compare with Microsoft SQL Server, a widely used database system. The storage performance that is achieved in CMBVS is 24.90 times cheaper than that of SQL Server The projection queries show great access speed compared with SQL Server. The investigation finds out that CMBVS is 77 times faster than SQL Server. The other types of select queries are 27.29~36.23 times faster than that of SQL Server. The indexing on CMBVS also affects the query time. The indexing reduces query time almost half of the non-indexed form of the compressed data. The scalability we tested up to 85 million tuples. In compressed form it occupies only 2.27GB whereas the SQL Server needs 56.34GB. The simulated storage space comparison for terabyte data is computed using the interpolation. The query time in parallel processor system is simulated using parallel algorithms.

Chapter 5
Conclusion and Further Research

Data compression in database system is attractive for two reasons; storage reduction and performance improvement. The both factor is essential for terabyte data management system.

Literature survey shows that compression techniques have been adopted in memory resident database systems that are not suitable for managing very large database ranges to terabytes. We have developed a disk-based multi-block vector structure for storage and querying of terabyte level of data.

We have improved the basic HIBASE compression model (by McGregor et. al. [21]) and Three Layer Model (by Latiful Hoque A. S. M. et.al. [18]) for disk support. Thus the architecture can be used in terabyte data management system.

5.1 Fundamental Contributions of the Thesis

- ❑ The main contribution of this research is the columnar multi block vector structure with very small restructuring overhead.
- ❑ Compressed data are stored using the CMBVS structure with disk support. This overcomes the limitations of the memory resident DBMS for scalability with high performance
- ❑ Multipart domain dictionaries are used to support high cardinality of domain values
- ❑ A great storage reduction is achieved using the CMBVS structure. The experimental results show that CMBVS structure is 22 ~ 25 times space efficient than that of SQL Server.
- ❑ We have shown that a PC-based system can handle a terabyte level of data. The performance for projection decreases linearly with the size of the database. The size of 1 TB of conventional database is 43 GB in CMBVS.

- We have achieved a very high order of performance improvement using CMBVS structure. Performance improvement for projection operation is in the order of 55 to 110 compared to conventional DBMS like SQL Server. The same is 27 to 36 for selection operation.
- We have proposed a terabyte data management system architecture that works on the CMBVS structure. The architecture can scale up to multi-terabytes of data by distributing the column vectors to the column severs and domain dictionaries to the domain servers.

5.2 Summary of the Thesis

The following sections summarize the thesis:

- The single block vector incurs huge restructuring cost, whereas the CMBVS needs very small restructuring overhead. Multi block structuring makes it easy to swap in memory and swap out to disk. Thus CMBVS can manage very large size of vector. We have used fixed number of elements per block for CMBVS, so that its size depends on element size in that block. The logical storage of block is imposed to physical disk block for permanent storage. As the size of the logical block is not fixed, it causes some fragmentation wastage of storage between two disk blocks.
- A domain dictionary is created per attribute. If the size of the domain is such that it can not be kept in memory, we use multi part domain dictionary. The dictionary is partitioned into fixed number of lexemes. One partition is swapped in memory. A single dictionary may be shared by multiple similar attributes if it is specified during designing the database.
- The experimental result shows that the storage cost of the CMBVS for 85 million tuples is only 2.27 GB whereas the SQL Server needs 56.34GB. The CMBVS is about 25 times efficient in storage space than that of SQL Server. The CMBVS considers the storage for vector structure, the dictionary and the index structure. CMBVS supports a compact index for both primary and secondary index. We could achieve higher compression factor in case of primary and secondary indices.

- Access performance is very high for a particular query where a few number of attributes are involved in the query. Because the architecture accesses only the related column vectors and dictionaries, whereas the conventional DBMS accesses the entire tuple for any query. Another reason behind the higher access performance is the reduced I/O access due to compression. The projection queries are 55 to 110 times faster than that of SQL Server. The access speed degrades as the number attributes increased in where clause. The selection (σ) query is 27 to 36 times faster than that of SQL Server.

- We have proposed a shared nothing parallel architecture for multi-terabyte data management system (section 3.2). The column vectors can be distributed over column servers on the supervision of CMBVS manager. The domain dictionaries can be distributed over domain servers on the supervision of domain dictionary manager. This provides a parallel computing environment to achieve high performance for multi-terabyte database. The architecture can store and manage virtually unlimited number of tuples with linear scale up of resources. The architecture may suffer some communication delay while interacting with different components. If we use gigabit range fiber optic communication interface, the delay will be negligible. Though the query time in this architecture is significantly improved than any uncompressed counterpart, the response time is still high for any typical terabyte database. To overcome this problem we should make hybrid fragmentation of the database and the use of parallel algorithm which was beyond our scope.

5.3 Future Plan

The CMBVS has been implemented in a single processor system and achieved significant performance improvement over conventional DBMS. We have proposed a shared-nothing architecture for further improvement in performance by parallel access to the vectors and dictionaries.

The CMBVS is a disk based compressed database architecture and extended to parallel architecture. This provides an easy way to use it in terabyte database system. The future work of this research is to explore on the following other areas:

- ❑ The architecture can be used for storage of fact table of data warehouse and parallel computation of datacube for OLAP system.
- ❑ The architecture can be used for parallel mining and analyzing of very large database.
- ❑ We have not considered any back-up and recovery mechanism to the CMBVS database. So the designing of commit protocol and redo-undo operations for compressed vectors and domain dictionaries need to be explored.
- ❑ To achieve concurrent access to CMBVS structure, a multi-threaded algorithm needs to be considered.

Bibliography

[1] Tashenberg, C. B. "Data Management Isn't What It Was", *Data Management Direct Newsletter*, May 24, 2002.

[2] Grossman, R. L. "The Terabyte Challenge: An Open, Distributed Testbed for Managing and Mining Massive Data Sets", *Proceedings of the Conference on Supercomputing,* 1996, Pittsburgh, November 1997.

[3] http://www.backbone.com/docs

[4] Whiting, R. "Data Management Tower of Power", *InformationWeek,* February 11, 2002.

[5] Goodman, I. "Emerging Network Attached Storage Strategies", *Technical Support Magazine,* September 1999, Vol. 7, No. 9.

[6] http://www.itri.org.tw/ eng/research/nano/

[7] http://www.systemonline.cz/site/data-warehousing/terabyte.htm

[8] Simonds, L. "A Terabyte for Your Desktop", *Maxtor Corporation Technical Report,* October 17, 2005.

[9] http://jazz.nist.gov/

[10] Cariapa, S., Clark, R. and Cox, B. "Origin 2000 One-Terabyte Per Hour Backup White Paper", *Silicon Graphics, Inc., Technical Report,* 2002, pp. 1-12.

[11] http://www.del.com/powersolutions/

[12] http://www.oracle.com/ corporate/press/2005/sep/

[13] Chaudhuri, S. and Dayal, U. "An Overview of Data Warehousing and OLAP Technologies", *SIGMOD Record,* Vol. 26, No. 1, 1997, pp. 65-74.

[14] Lawrence, R. and Kruger, A. "An Architecture for Real-Time Warehousing of Scientific Data", *International Conference on Scientific Computing*, Vegus, Nevada, June 2005, pp. 151-156.

[15] Barker, R. "Managing a Data Warehouse", *Veritas Software Corp., Technical Report,* Chertsey, UK, 2001.

[16] Heath, D. J., Chambers, R. W., Leggett, W. D., Tuggle, E. D., Warner, D. D., Wobschall, C. M., Fiedler, L. E. and Polluconi, M. A. "A Multi-Terabyte, Hierarchical Data Warehouse for Continuous, High-Rate Object Streams", *System and Information Review Journal,* Spring/Summer, 2000, pp. 71-88.

[17] Shannon, C. E. "A Mathematical Theory of Communications", *Bell System Technical Journal*, Vol. 27, 1948, pp. 379-423.

[18] Latiful Hoque, A. S. M., Accepted for the Degree of Ph. D. on "Compression of Structured and Semi-structured Information", *Department of Computer and Information Science, University of Strathclyde*, Glasgow, UK, 2003.

[19] Reghbati, H. K. "An Overview of the Data Compression Techniques", *IEEE Computer*, Vol. 14, No. 4, 1981, pp. 71-75.

[20] Steinmentz, R. and Nahrsteat, K. "Multimedia Computing Communication and Application", ISBN. No. 0-13-324435-0, July 1995, pp.115-125.

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

[22] Lee, I. and Yeom, H. Y. "A New Approach for Distributed Main Memory Database Systems: A Casual Commit Protocol", *IEICE Trans. Inf. & Syst.*, Vol. 87-D, No. 1, 2004, pp. 196-204.

[23] Church, K. W., Fowler, G. S., Buchsbaum, A. L., Caldwell, D. F. and Muthukrishnan, S. "Engineering the Compression of Massive Tables: An Experimental Approach", *Proceedings of the 11th Annual ACM-Siam SODA*, San Francisco, January 2000, pp. 175–184.

[24] http:/www.almaden.ibm.com/

[25] Poess, M. and Potapov, D. "Data Compression in Oracle", *Proceedings of the 29th VLDB Conference*, Berlin, Germany, September 2003, pp. 937-947.

[26] http:/www.kx.com/

[27] Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Charniack, M., Ferreira, M., Lau, E., Lin, A. and Madden, S. "C-Store: A Column-Oriented DBMS", *Proceedings of the 31st VLDB Conference*, Trondheim, Norway, August/September 2005, pp. 553-564.

[28] Thakar, A., Szalay, A. and Kunszt, P. "Migrating a Multi-Terabyte Archive from Object to Relational Database", *Computing in Science and Engineering*, Vol. 5, No. 5, September/October 2003, pp. 16-29.

[29] http://www.cis.strath.ac.uk/research/papers/ strath.cis.publication 227.pdf

[30] Codd, E. F. "Large-scale Data Warehousing Using Essbase OLAP Technology", *Hyperion White Paper*, 2000.

[31] Cannane, A. and Williams, H. E. "A General Purpose Compression Scheme for Large Database", *Proceedings of the Data Compression Conference,* Snowbird, Utah, 1999, pp. 519-525.