DMBVA - A COMPRESSION-BASED DISTRIBUTED DATA WAREHOUSE MANAGEMENT IN PARALLEL ENVIRONMENT

***Fazlul Hasan Siddiqui*[1], *Abu Sayed Md. Latiful Hoque*[2]**
[1]Department of Computer Science and Engineering,
Dhaka University of Engineering and Technology, Gazipur 1700, Bangladesh.
Email: fhsani@yahoo.com
[2] Department of Computer Science and Engineering,
Bangladesh University of Engineering and Technology, Dhaka 1000, Bangladesh.
Email: asmlatifulhoque@cse.buet.ac.bd

ABSTRACT

Parallel and distributed data warehouse architectures have been evolved to support online queries on massive data in a short time. Unfortunately, the emergence of e-application has been creating extremely high volume of data that reaches to terabyte threshold. The conventional data warehouse management system is costlier in terms of storage space and processing speed and sometimes it is unable to handle such huge amount of data. As a result, there is a crucial need for the new algorithms and techniques to store and manipulate these data. In this paper, we have presented a compression-based distributed data warehouse architecture – 'DMBVA' for storage of warehouse data, and support online queries efficiently. We have achieved a factor of 25-30 compression compared to SQL server data warehouse. The main computational component of data warehouse is the generation and querying on the data cube. Our algorithm – 'PCVDC' generates data cube directly from the compressed form of data in parallel. The reduction in the size of data cube is a factor of 30-45 compared to existing methods. The response time has also been significantly improved. These improvements are achieved by eliminating the suffix and prefix redundancy, virtual nature of the data cube, direct addressability of compressed form of data and parallel computation. Experimental evaluation shows the improved performance over the existing systems.

Keywords: ***Data Warehouse, Compression, Parallel, Virtual Data Cube***

1.0 INTRODUCTION

Data Warehouse is a subject oriented, integrated, non-volatile and time-variant collection of massive data in support of management's decisions. Today's warehouses are approaching several terabytes of data with a broader mix of query types, which are not supported by the conventional archiving of data. As a result, this increased data put a number of strains on physical resources, IT aspects and management spectrum. Performance can be improved using compression based distributed warehouse. In contrast, several distributed warehouse architectures have been put forward without the application of compression methods. Integrating these two technologies, our work exhibits a complete solution to a compression based distributed data warehouse management in a parallel environment, which is the first part of our contribution, and we named it as '**D**istributed **M**ulti-**B**lock **V**ector **A**rchitecture (DMBVA)'.

Next, **Data cube** construction requires computation of aggregations on all possible combinations of each dimension attribute. Pre-computation of this data cube is very critical to improve the query performance of data warehouse. However, as the data warehouse size grows, this pre-computation becomes a significant performance bottleneck, because the required computation cost grows exponentially with the increase of dimensions. Hence, this paper demonstrates virtual data cube construction directly from the compressed and distributed data warehouse in parallel on a shared nothing multiprocessor system. We named it as '**P**arallel **C**onstruction of **V**irtual **D**ata **C**ube (PCVDC)'. Additionally optimization techniques are applied onto the data cube for better performance.

We have developed a three-tier data warehouse architecture as shown in Fig. 1. In tier one, data is collected from the external source of operational data, which are then extracted, transformed, refreshed, and loaded into the data warehouse server. Next with the intension of compressing the warehouse data, we have designed a Distributed Multi-Block Vector Architecture (DMBVA) in tier two to store the relational schema in a compressed and distributed format. In tier three, we have generated the data cube directly from the compressed vector in parallel. This procedure is named as PCVDC and it belongs to the main focus of our work. To achieve the optimal result in terms of storage space and construction time of data cube, prefix and suffix redundancy are eliminated. Finally the

clients do their queries from the virtual data cube with the help of the query manager, which is synchronized by the application server.

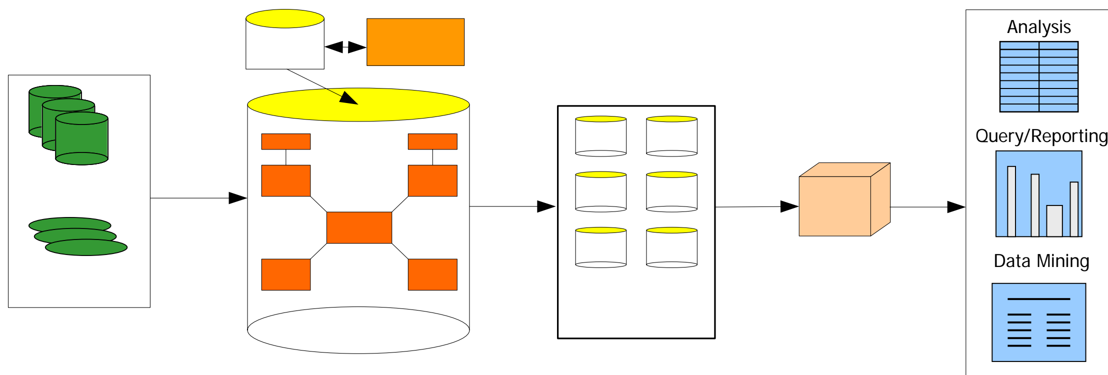

Fig. 1: Overall system architecture

The rest of this paper is organized as follows: Section 2 describes different related work, their utility and drawbacks. Section 3 depicts our compressed relational architecture – DMBVA, with detailed analysis of its components – domain dictionary, compressed vector and index structure. Then the PCVDC algorithm with the data cube file format, their detailed analysis and evaluation is exposed in Section 4. The optimization for data cube creation and storage is presented in Section 5. In Section 6, experimental outcome is discussed, where various comparisons are made on the basis of storage space, construction and query response time against other work. Finally conclusion and future work is delivered in Section 7.

2.0 RELATED WORK

Some of the aspects of creating, maintaining, and querying a distributed and parallel data warehouse are outlined in [1]. Teradata's parallel processing architecture [2] possesses distinct advantages for highly complex, large scale data warehouse. Cost optimization for distributed data warehouse is addressed in [3] which is based on a weighted sum of query and maintenance costs. WHIP architecture [4] performs identification, transformation and incremental integration of distributed heterogeneous data. SAND Technology [5] proposed new architecture of data warehouse that reduces storage requirements and therefore costs. Agent-Based Warehouse [6] architecture put forward a great contribution for seamless integration of new data. None of these works addresses the use of compression in storage and querying warehouse data.

Alternatively, research on database compression put a great role on performance improvement. A dictionary based compression technique HIBASE model [7] is further extended by the three-layer architecture [8] for sparsely populated and XML data. Migrating a multi terabyte archive from object to relational database is depicted in [9]. CMBVS [10] disk based compression technique handle terabyte level database.

Data cube computation algorithm mostly based on sorting and hashing techniques. PipeSort, Overlap and MemoryCube [11] are sort based where as Pipehash [11] is hashing based algorithm. Since sorting a terabyte sized data warehouse takes a huge computation time, our approach is based on hashing technique. Dwarf [12] and PrefixCube [13] separately present compact data cube structures and addresses the issues of prefix and suffix redundancy, but their structures store the lexemes of the dimension attributes which occupy massive space. On the contrary, in our approach, prefix and suffix redundancy is automatically abolished for the absence of lexemes. Recently, different fast parallel methods are developed for constructing data cubes [14, 15] based on partitioning and efficient parallel merge procedure with a cost of lexeme storage. This merge cost is much higher in case of very large scale data warehouse.

3.0 DISTRIBUTED MULTI-BLOCK VECTOR ARCHITECTURE

With the intension of compressing the warehouse data, we apply our Distributed Multi-Block Vector Architecture (DMBVA) to store the relational schema in a compressed and distributed format. DMVBA is an extended

modification of CMBVS [4]. Here for successful storage management, the relational warehouse is compressed and distributed over multiple column servers as shown in Fig. 2. Each column server is associated with a domain dictionary, a large compressed vector and an index structure. Application server stores the measure attributes and synchronizes among those column servers.

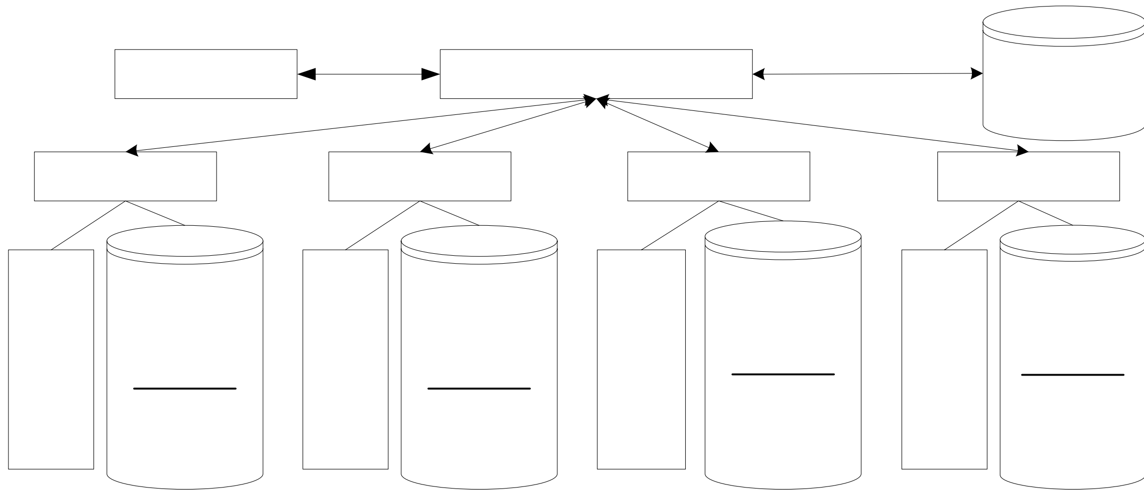

Fig. 2: General architecture of distributed compressed data warehouse containing a fact table of four dimensional attributes 'ABCD' and one measure attribute 'M'.

3.1 Domain Dictionary

The domain dictionary consists of set of lexemes with corresponding token values. It is updated dynamically whenever facing a new lexeme. We have created disk-based multiple part dictionaries to accommodate huge quantities of lexemes, where some parts resides in the memory and others are in the disk, which are fetched on demand basis. There are two main operations in the dictionary. *(i) getToken(lexeme):* this operation return the token of the given lexeme. If the lexeme is not present in dictionary, then add it there and return the corresponding new token. *(ii) getLexme(token):* it returns the corresponding lexeme of the token. The size of the resultant dictionary in the ith column server is: $C_i \times L_i$. Where, C_i = cardinality of domain attribute, L_i = average length of lexeme, I = index structure storage size. To search the dictionary hashing technique is applied, where the time cost is $O(1)$.

3.2 Compressed Vector

A compressed vector represents a single column of relational database. Here only the tokens of the corresponding lexemes are stored and hence compressed. As the main memory is limited, so each vector is partitioned into blocks, each block is partitioned into machine words, and a certain number of blocks reside in memory, the others are kept in the disk. We store fixed number of elements per block. The size of element stored in the vector is uniform within any block, but may vary between blocks. Any operation in the vector ultimately propagated to the block. Multi-block structure reduces reorganization cost of vector and also reduces the wastage of space.

The main operations of the block are, (i) isEmpty(): check whether the block is empty. (ii) addToken(token): 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. (iii) getToken(index): This operation returns the ith token from the block and i must be smaller than or equal total number of elements in the block. (iv) widen(numBit): 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. (v) getTupleId(token): This operation returns a set of record_IDs from the block that has the same token found in parameter. This record_ID represents the index of matching tuple within the block (not the

database record_ID). If no matching token found in the block then it returns –1. The overall size of the compressed vector is:

$$w\sum_{i=1}^{\lceil n/m \rceil}\left\lceil \frac{m \times v_i}{w} \right\rceil$$

Here, 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).

The overall creation time complexity is the sum of the,

(i) Overall widening operation (U=$O(\lceil n/m \rceil \times \log_2 m)$).

(ii) Time to store blocks in disk: $O(\lceil n/m \rceil \times T_s)$.

(iii) Dictionary creation time: *total dictionary search time + total block storage time* = $O(C_i) + O(B_d \times T_d)$.

Here, T_S = time to store a compressed block into the disk, T_d = time to store a dictionary block into the disk and B_d = size of the dictionary block. Here, the network delay will be zero, because each compressed vector and its corresponding domain dictionary resides in the same column server.

3.3 Index Structure

An **index file** is created if the lexeme is a search key. The index structure is associated to <lexeme, token> instance. We transform lexeme to token using dictionary and then token is used as a new search key. A hashing technique is used to extract the specific token from the multi-block vector structure.

4.0 PARALLEL CONSTRUCTION OF VIRTUAL DATA CUBE

A data warehouse is based on a multidimensional data model which views data in the form of a data cube. Data cube is the computation of aggregation on all possible combination of dimension attributes. For a base relation R with N dimension attributes $\{D_1, D_2 \ldots, D_N\}$ and one measure attribute M, the full data cube of R on all the dimensions is generated by CUBE BY on $D_1, D_2 \ldots, D_N$. It is equivalent to the union of 2^N distinct GROUP BY. Each group-by is called a cuboid. The CUBE BY is a very expensive operator, and the size of its result is extremely large especially when the number of CUBE BY attributes and the number of tuples in the source relation is large.

4.1 Data Cube File Format

Data cube construction format is depicted in Fig. 3. Here the smallest unit is the record_ID, which is a sequence of bit string. Every single record_ID and its corresponding measure_Value melded into a cube_Record. Multiple cube_Record form a cube_Word, where multiple cube_Word fused into a cube_Block. Every cube_Block is independently addressable so the structure is suitable for large data cube. Here each cube_Block contains a fixed number of cube_Records and hence the memory requirement for different blocks of different cuboids may be varied. However the cube_Records of a particular cuboid are mapped contiguously to the underlying cube_Words. So no space is wasted at the word boundaries except at the block boundaries. Then, multiple cube_Block form a cube_Vector, alternatively which is called a cuboid. Finally all the cuboids collectively form the resultant data cube.

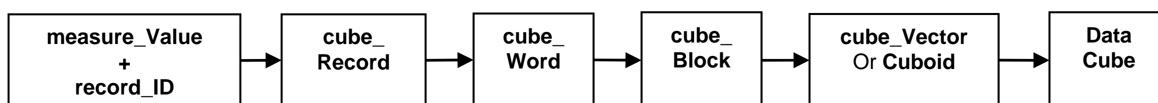

Fig. 3: Data Cube construction procedure.

We have generated the data cube directly from the compressed vector in parallel. A data cube is constructed from a schedule tree, whereas a schedule tree is generated from a cuboid lattice (Fig. 4(a)) by solving a minimum spanning tree problem. This is illustrated in Fig. 4 for a four dimensional attribute 'ABCD'. Here we assume that $|A| \geq |B| \geq |C| \geq |D|$. There are two types of data cube file: root cuboid and child cuboid. Fig. 4(b) depicts the flow of data cube generation.

4.2 PCVDC Algorithm

The general idea of our Parallel Construction of Virtual Data Cube (PCVDC) [16] is: starting from the first token, each column server extracts a chunk of tokens from their individual compressed vector in parallel and sends it to the application server. Application server then generate and broadcast {record_ID, measure_val} to all the column servers. These are stored into the blocks of the cube_vectors. Repeating this process the root cuboid is generated. Each column server then locally computes the child cuboids from its smallest parent cuboid in parallel as assigned in the schedule tree. Eventually the complete data cube is constructed. The detailed algorithms are depicted in Algorithm 1, 2 and 3.

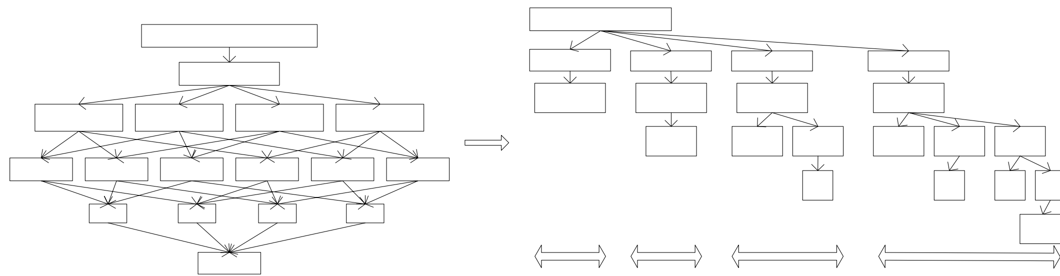

Fig. 4: (a) A cuboid lattice of four dimensional attributes 'ABCD'.
(b) A Schedule tree produced from the aforementioned lattice

Algorithm 1: PCVDC in the Application Server

Input: tokens from column server
Output: record_IDs and measure_Values

1. create a server socket
2. create as many as child threads for each column server
3. pass socket acceptance to each thread constructor
4. for each column server thread pardo
5. while(true) do
6. wait to receive a chunk of tokens from each column server
7. send this token to the root thread
8. root thread generates the record_IDs using the collected tokens and extract the corresponding measure values
9. send these record_IDs and measure_Values to the column server
10. if the last token had received then break
11. end while
12. end for

Algorithm 2: PCVDC in the Column Server

Input: schedule tree, compressed vector file
Output: complete data cube

1. create a thread for this column server
2. create a socket and connect to the server port
3. while EOF of all compressed vector file is not reached
4. extract a chunk of tokens from the vector
5. send these tokens to the application server
6. wait to receive the record_IDs and the measure_Values from the application server
7. store them to the rootCuboid file
8. end while

Algorithm 3: constructChildCuboid(rootCuboid)

1. r ← rootCuboid
2. generate all childCuboids of r
3. for each children c of r from left to right do
4. if c has no children then
5. return to parent cuboid
6. else constructChildCuboid(c)
7. end if
8. end for

4.3 PCVDC Analysis & Evaluation

Fact Tab

Let, m = average cardinality of each dimension attributes. n = Average number of tuples (elements) in compressed vector. d = Number of dimension attributes. b = Total number of blocks. For a ***single server system***, time complexity to construct a data cube:

ABCD

ABC ABD

$$
\begin{aligned}
& O\left(m^{d} \left\{ n + \frac{n}{m}(d-1) \right\} \right) + \\
& O\left({}^{d}C_{d-1} m^{d-1+1} + {}^{d}C_{d-2} m^{d-2+1} + \cdots + {}^{d}C_{1} m^{1+1} \right) \\
& = O\left(m^{d} n\right) + O\left(dm^{d}\right) = O\left(m^{d} n\right)
\end{aligned}
$$

In the case of root cuboid generation: total number of tuples in the data cube is m^d, for each tuple in the data cube, we have to search sequentially all the n tuples of the first dimensional attribute (vector) to match with the first element of that data cube tuple. If a match is found then we have to extract the same tuple of the next dimensional attribute (vector) in order to find a match with the next element of that particular tuple in the data cube. On an average, there are n/m replicas of each tuple of the domain dictionary in the vector. So for each tuple in the data cube, total searching for the rest of the d-1 dimension attributes (vectors) is: $(n/m)(d-1)$. In the case of child cuboid generation: since there are total ${}^{d}C_{d-x}$ '*d-x*' dimensional child cuboids of a *d* dimensional cuboid, for each of this child cuboid, a total of m^{d-x+1} comparison is made. Now, For a ***multiple server system***, time complexity to construct a data cube is $O(n) + O(m^d) = O(m^d)$. Here to generate the root cuboid, only *n* number of searching is made in parallel through the compressed vectors. Whereas, for child cuboid generation, it is same as single server system, but the generation is distributed among d number of servers. So a maximum of m^d comparison is made. If the number of dimensions in the source relation is N, i.e. total number of cuboids is 2^N, number of dimension in the cth cuboid is d_c, cardinality of the jth domain of the cth cuboid is C_{cj}, size of the block-ID is B_c and measure value is M in c^{th} cuboid. Therefore the maximum size of the data cube is:

$$
\sum_{c=1}^{2^N} \left\{ \left(\prod_{j=1}^{d_c} \left(C_{cj} + 1\right) \right) \times \left(B_c + M\right) \right\}
$$

4.4 Query from the Data Cube

For centralized systems, the primary criterion for measuring the cost of a particular strategy is the number of disk accesses. In a distributed system, other issues must be taken into account:

- The cost of data transmission over the network.
- The potential gain in performance from having several sites process parts of the query in parallel.

The relative cost of the data transfer over the network and data transfer to and from disk varies widely depending on the type of network and on the speed of the disk. Thus, we cannot focus solely on disks cost or on the network cost. Rather, we must find a good tradeoff between the two. We speed up the processing of a query by executing the different operations in parallel of a query expression**.** Here the resultant data cube is independently distributed and the different query expressions which are imposed onto the same cube_Record are managed by pipelining. Therefore is no necessity of locking and logging in the query operations and cache-coherency problem is eliminated as well. A detailed query operation is shown in Fig. 5 and the corresponding algorithm is depicted in Algorithm 4.

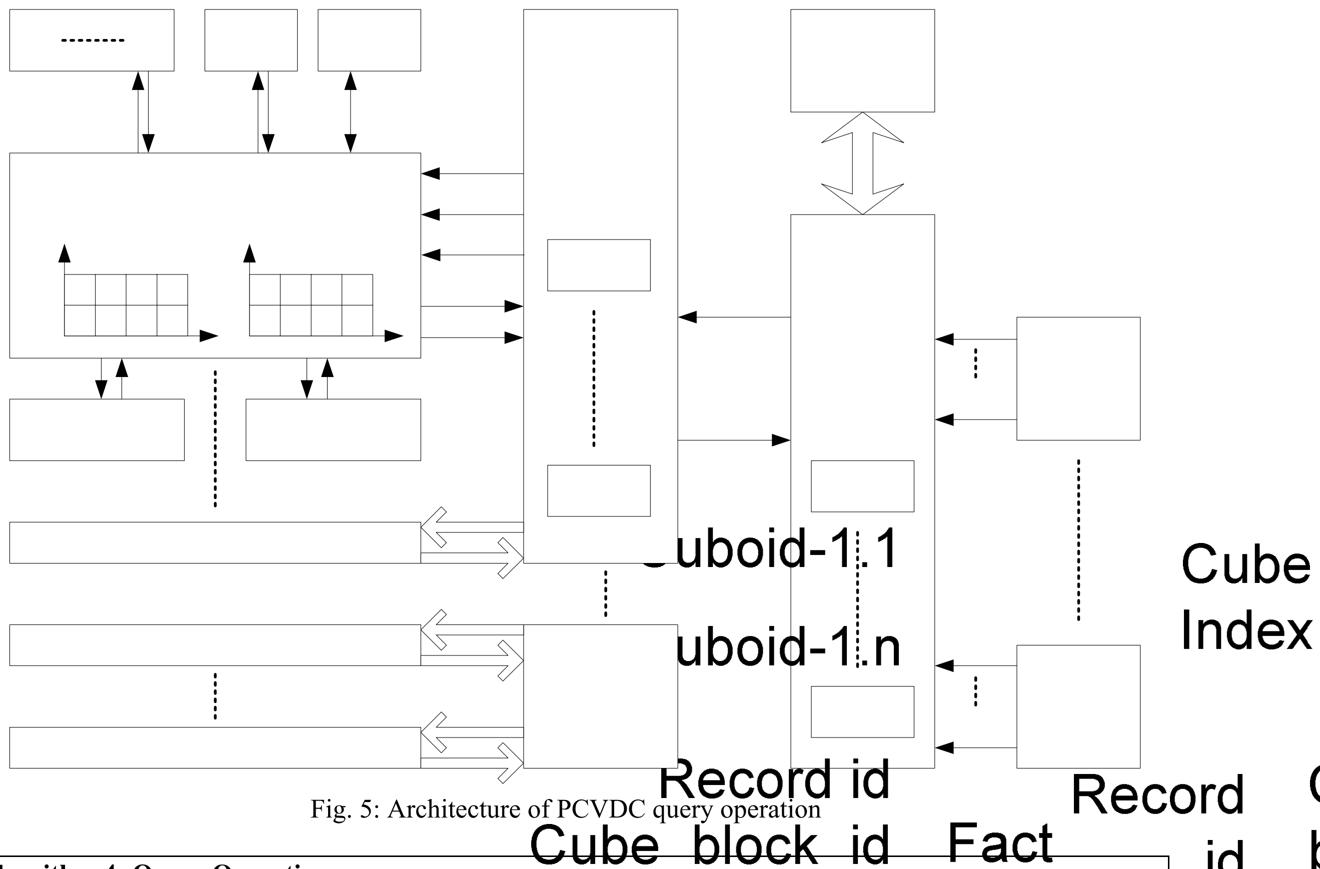

Fig. 5: Architecture of PCVDC query operation

Algorithm 4: Query Operation

Input: query

Output: resultant dataset

1. Application server creates a client session for each client when client connected to application server. The client session is created by multithreading.
2. Application server talk over the query manager for each query and the cheapest execution plan is found out among the many possible execution plans that give the same answer.
3. Translate each query into the SQL form.
4. Select the appropriate Middleware server from its metadata, which is associated with the column servers under which the query results reside.
5. Middleware server decomposes the tokenized query into individual projection, sort, where condition, join operation and determine the dependency between them. Then it forwards the request for projection and condition operation to the pertinent column servers.
6. Column servers extract the result of the condition and projection operation from the appropriate cuboids with the help of the cube index.
   - 6.1. For condition it creates two stages Pipeline for Domain Dictionary (DD) in the 1st stage and Data Cube (DC) in the 2nd stage. Here C_x implies condition number x.
   - 6.2. For projection it also creates two stages pipeline for Data Cube (DC) in the 1st stage and Domain Dictionary (DD) in the 2nd stage.
   - 6.3. When different pipelining stages try to invoke the same domain dictionary or data cube block, it is handled thread synchronization.
7. Finally, the results containing facts and lexemes are sent back to the Middleware server, which are packed with the query ID and eventually passed back to the Application server.

4.4.1 An Example Query Operation

A simple query example is shown in Fig. 6, which is constructed from the schedule tree (Fig. 4 (b)) of the PCVDC architecture. The example query is made based on customer relation entropy. Given five dimensional attributes, which are ID (X), Name (A), Street (B), City (C), Status (D), and one measure attribute named Salary (M). Let the query is "Select Name, Max (Salary) From Customer Relation Where Street = 'bijoy' AND Status = 'Unmarried'."

This query is imposed to the query manager which is forwarded to application server in a different manageable format. Then it is partitioned into sub-query by the application server and send on to the appropriate column server for tokenizing the query attributes. After that, as the result of the query lies in the cuboid ‘ABD’, whereas the cuboid ‘ABD’ resides in the column server-B, so the query tokens are passed to column server-B. Then the result, Name = ‘011..0’ and Salary = ‘40000’ is passed back to application server. Token ‘011..0’ is further sent to column server ‘C’ for decompression, therefore Name = ‘Obayed’ is returned, which is passed back to the query manager.

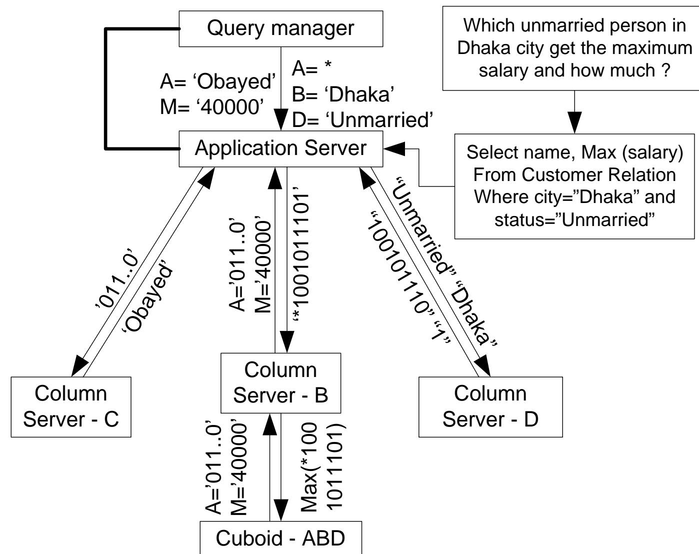

Fig. 6: A simple query example

4.4.2 Time Complexity to Query from a Data Cube

Let, t = Query tokenization time, m = Average no. of cube records per cube_block. d = Number of dimension attributes, and net_d = overall data transmission delay through the network. Now, the overall query response time = Query tokenization time + Cube_block search time + Cube_record searching time + Network delay. Since a hash indexing is used to locate the cube_block and binary search is applied to locate the cube_record, therefore cube_block searching time is $O(1)$ and cube_record searching time is $O(\log_2 m)$. So, the overall query response time we achieved is: $O(t + \log_2 m + net_d)$.

5.0 OPTIMALITY ISSUE OF PCVDC

Prefix and suffix redundancy are identified in our approach and eliminated as well. Prefix redundancy is understood by the fact that there is a cube with dimensions A, B and C. Each value of dimension A appears in 4 group-bys (A, AB, AC, ABC) and possibly many times in each group-by. This prefix redundancy can be further classified into inter-cuboid and intra-cuboid prefix redundancy. Since there is no direct storage of dimension attribute values, the prefix redundancy is automatically wiped out. Suffix redundancy occurs when two or more group-bys share a common suffix (like ABC and BC). For example, consider a value B_j of dimension B that appears in the fact table only with a single value A_i of dimension A. Then, the group-bys <A_i, B_j, x> and <*, B_j, x> always have the same value for any value x of dimension C. More specifically taking an extreme case as an example, let a fact table R have only a single tuple r = <A_2, B_1, C_3, M>. Then, the data cube of R will have $2^3 = 8$ tuples, <A_2, B_1, C_3, V_1>, < A_2, B_1, *, V_2>, <A_2, *, C_3, V_3>, <A_2, *, *, V_4>, <*, B_1, C_3, V_5>, <*, B_1, *, V_6>, <*, *, C_3, V_7>, <*, *, *, V_8>. Since there is only one tuple in relation R, we have $V_1 = V_2 = \ldots = V_8$ = aggr(r). Therefore, we only need to physically store one tuple <A_2, B_1, C_3, V>, where V= aggr(r). So the cube for R with 8 tuples can be condensed into one tuple. Our optimization feature detects and purges this type of redundancy. Algorithm 4 depicts the optimal cuboid construction procedure where the suffix redundancy is eliminated.

Algorithm 4: Constructing Optimal Cuboid
Input: name: cuboid name = ‘X1X2…Xn tokens from column server
metadata: name.dat = ‘X1X2…Xn’.dat
Output: optimal cuboid
1. Load ‘X1X2…Xn’.dat into the memory.
2. t ← 1st tuple going to be added into the cuboid ‘X1X2…Xn’
3. while any tuple remains to be added do
4. bID ← blockID of t
5. mVal ← measureValue of t
6. check the existence of t into the metadata
7. if not found
8. add the bID and mVal into the cuboid
9. end if
10. if $Xj_{valj} \ldots Xk_{valk}$ is a constant suffix of $Xi_{vali} \ldots Xj\text{-}1_{valj-1}$, where i <= j <= k, then
11. broadcast (fileName=Xj…Xk, suffixToken=valj…valk, prefixName = Xi…Xj-1, prefixToken= vali…valj-1) targeted to the column servers which contain any cuboid, whose name starts with Xj…Xk.
12. end if
13. t ← next tuple going to be added into this cuboid
14. end while

Algorithm 5: Update Metadata
Input: broadcastInfo
Output: updated metadata file
(Running under an infinite thread on each column server)
1. if the file addressed in the broadcast_Info do not exist then
2. create this file.
3. else if the broadcastInfo is already exist then return
4. else
add ‘suffixToken, prefixName, prefixToken’ into the file referred as fileName.dat

6.0 EXPERIMENTAL SETUP

In our experimental setup there were one application server and several column servers (one for every compressed vector) connected via LAN. Each server having 3GHz Pentium IV processor, 40GB hard disk and 512MB RAM. The platform was windows XP and was implemented by java development kit 1.5.

We used TCP (Transmission Control Protocol) for the communication between two of our applications residing in different computers. During the root cuboid of the data cube construction process we used multicasting which can be thought of as broadcasting data over a network connection to many connected agents, as opposed to unicasting packets between two agents on a normal connection. To support concurrent access onto the Application server by multiple Column servers for the root cuboid generation we used multithreading. Also multiple tokenized query requests which are imposed onto the Column servers are handled by multithreading. When the multiple query threads in our system tried to access the same data cube or dictionary block asynchronously, it was handled by Thread Synchronization. Collection of threads is managed by Thread Pool to improve the performance when executing large numbers of tasks as a result of reduced per-task invocation.

We chose a classical retail data warehouse of an enterprise which is collected from the oracle data warehouse corporation. A synthetic data generator was implemented to generate the data that resembles the real data sets for merchandising relation. We assumed six attributes (Table 1) and considered approximately 4 million customers of an enterprise. Based on this, the approximate cardinality of each dimension attribute is shown subsequently. Each

record is estimated as 104 bytes in the uncompressed base relation, whereas each compressed record occupies at most 7 bytes.

Table 1: The sales relation structure

Attribute Name	Data Type	Cardinality	Length (bytes)	Bits required
Sales_Id	Integer	Primary key	4	22
Product Name	Varchar	507	30	9
Customer Name	Varchar	5000	30	13
Customer Status	Varchar	2	10	1
Sales City	Varchar	64	30	6
Price Charged	Double	Measure value		
Total			104 bytes	51bits ≈7bytes

Table 2: Storage requirement of DMBVA versus SQL Server

Number of records (million)	Storage space in SQL Server (MB)	Storage space in DMBVA (MB)	Compression Factor against SQL Server
1	251.92	9.525	26.45
2	501.86	18.312	27.40
3	755.76	26.32	28.71
4	1011.65	33.56	30.14
5	1289.36	41.20	31.30

6.1 DMBVA Storage

The storage space occupied by DMBVA demonstrates a significant compression factor against SQL Sever, as shown in Table 2. The average compression ratio is 28.87 with respect to SQL Server. This is because SQL Server needs to store the original source relation in an uncompressed format with some additional information which contrasts with our work.

6.2 Data Cube Storage

The compressed vector act as the source relation for the generation of the data cube, and the data cube was generated directly from this compressed format in a parallel environment. Since the data cube was generated directly from this compressed vector and its contents are virtual, a great deal of reduction in the ultimate storage space was achieved. We compared our system with binary storage footprint (BSF) and Dwarf [12]. In PCVDC the fact table i.e. the compressed vector contained 100000 tuples with a uniform distribution of the dimensional values over a cardinality of 1000. In Table 3, the storage occupied by different methods for different dimensions is demonstrated. This experimental result is consistent with the analytical evaluation which is stated in section 4.3.

BSF comparatively took much more space for all the dimensions, because all the cuboids are stored here in non-indexed binary summary table without eliminating the redundancy. Excluding Prefix but Including Suffix redundancy (EPIS) both Dwarf and PCVDC perform better. However, in this case, PCVDC dominates Dwarf in the data cube storage reduction due to the absence of lexemes. After eliminating the suffix redundancy (noted as ES) the storage cost reduced a lot for both methods. We achieved a significant storage reduction for all the dimensions with respect to Dwarf, because of the elimination of lexeme storage and data cube storage format in PCVDC. Again, in our system this storage was reduced by a factor of 1/N, where N is the number of column server. This is

because different cuboids were stored by different servers according to the schedule tree. So it is clear that PCVDC dominate all other methods in the case of storage reduction, where suffix redundancy is a crucial factor in the overall performance. Therefore, our parallel system architecture scales well with the number of dimensions. Here a constant running time can be maintained as the dimension is increased by adding a proportion number of processors.

Table 3: Data Cube storage cost comparison with other system

Dim	BSF	Dwarf EPIS	PCVDC EPIS	Dwarf ES	PCVDC ES	Storage /column server
10	2333 MB	1322 MB	538.33 MB	62 MB	25.26 MB	2.53 MB
15	106 GB	42.65 GB	21.57 GB	153 MB	77.38 MB	5.16 MB
20	4400 GB	1400 GB	838.37 GB	300 MB	179.65 MB	8.98 MB
25	173 TB	44.8 TB	30.83 TB	516 MB	355.09 MB	14.20 MB
30	6.55 PB	1.43 PB	1.16 PB	812 MB	658.69 MB	21.95 MB

6.3 PCVDC Construction Time

The data cube was generated directly from this compressed vector in parallel, hence a great deal of reduction in the ultimate construction time was achieved. We compared the construction time of PCVDC with Dwarf (Fig. 7). The same data set was used here. Dwarf construction time was proportional to its size, and required much less time as it avoids computing large parts of the cube by eliminating the suffix redundancy. On the contrary, PCDVC creation time was little higher for smaller dimensions with respect to Dwarf. This is because, during the construction of root cuboid, it faced the data transmission overhead through the network channel. But after that all other cuboids were constructed independently in parallel from their parent cuboids in different column servers. Due to this parallel nature, a significant reduced creation time was achieved for bigger dimensions.

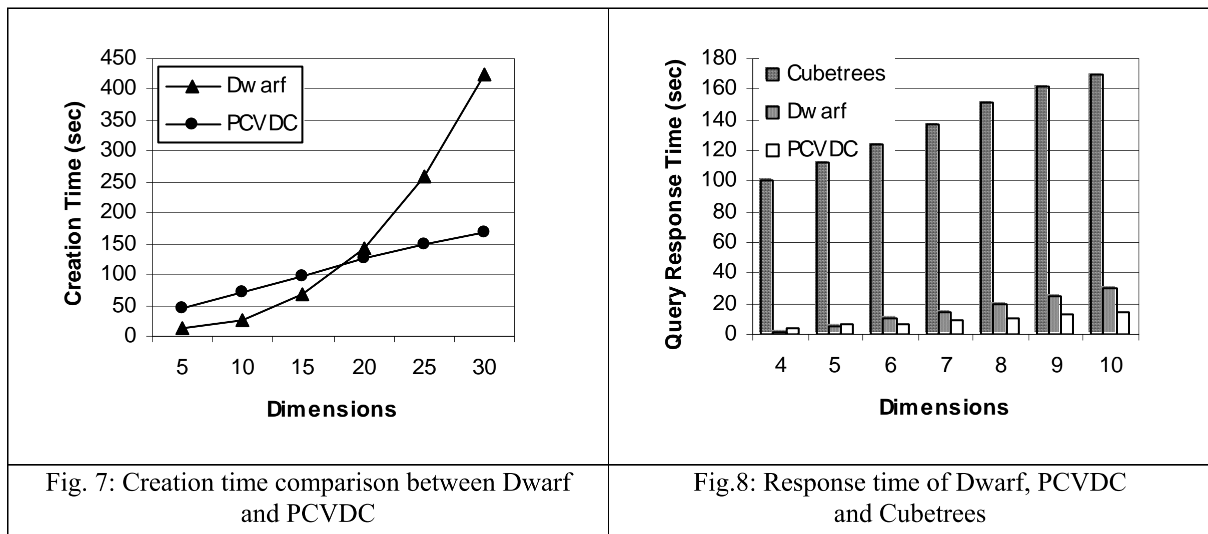

Fig. 7: Creation time comparison between Dwarf and PCVDC

Fig.8: Response time of Dwarf, PCVDC and Cubetrees

6.4 Query Response Time

We speed up the processing of a query by executing different operations in parallel in a query expression. Therefore we achieved a significant reduced query response time by applying the intraquery and interquery parallelism. The response time was experimented with Dwarf [12] and Cubetrees [15] (Fig. 8). Here the query was made against a complete cube of 4−10 dimensions with 300000 tuples in the fact table. We made almost 1000 queries on projection operation and multiple predicates select queries which include multiple attributes in the where clause. Both range and point queries were made. Query against a data cube was actually be reduced to a sub-query against only one particular cuboid or a union of such sub-queries, and this is done by the query manager. The full Cubetrees [15] took a huge response time since it calculates the entire cube. Dwarf cube performed better in case of smaller dimensions but as the dimension grows it became difficult to manage the condense storage in the main memory. In

contrast with that, PCVDC performed much better for higher dimensions with a little sacrifice in the query tokenization time. But it was overcome by the parallelization of the query operation. Up to 10 dimensions, the average speedup in the response time were 16.34 and 1.59 with respect to Cubetree [15] and Dwarf Cube respectively. This speedup would be further accelerated for higher dimensions due to our parallel architecture.

To compare the access speed of projection operation we choose 4 example queries shown in Table 4. Query Q1, Q2, Q3 and Q4 represents the projection operation of 1, 2, 3 and 4 attributes respectively. The set of quires were imposed onto 1 million tuples. 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. The PCVDC accesses only the information related to the attributes used in query. The average access speed for projection operation is a factor of 45.78 with respect to SQL Server.

Table 4: Comparison of Access Speed on Projection Queries

Sl. No.	**Number of Attributes**	**Average Elapsed Time (sec)**		**Access Speed (T_s/T_c)**
		SQL Server (T_s)	**PCVDC (T_c)**	
Q1	Single attribute projection	220	6.29	34.98
Q2	Two attribute projection	285	6.85	41.60
Q3	Three attribute projection	342	7.13	47.97
Q4	Four attribute projection	457	7.80	58.59

We also imposed the Multiple Predicates SELECT queries onto one million tuples. We take here four queries given in Table 5. Query Q5, Q6, Q7 and Q8 has one, two, three and four attributes respectively in the WHERE clause. We run the same query in SQL Server and PCVDC. The average access speed we achieved was a factor of 38.8.

Table 5: Multiple Predicates SELECT Queries

Sl. No.	**SQL Command**	**Average Elapsed Time (sec)**	
		SQL Server (T_s)	**PCVDC (T_c)**
Q5	SELECT * FROM Sales_Table WHERE Ai='XXX' ; ($2 \leq i \leq 5$)	245.43	8.24
Q6	SELECT * FROM Sales_Table WHERE Ai='XXX' AND Aj='YYY'; ($2 \leq i, j \leq 5$ and $i \neq j$)	337.67	9.17
Q7	SELECT * FROM Sales_Table WHERE Ai='XXX' AND Aj='YYY' AND Ak='ZZZ' ; ($2 \leq i, j, k \leq 5$ and $i \neq j \neq k$)	388.44	9.45
Q8	SELECT * FROM Sales_Table WHERE Ai='XXX' AND Aj='YYY' AND Ak='ZZZ' AND Al='WWW' ; ($1 \leq i, j, k, 2 \leq 5$ and $i \neq j \neq k \neq l$)	444.55	10.55

7.0 CONCLUSION AND SCOPE OF FUTURE WORK

There are works with distributed and parallel architecture of data warehouse without compression. Again, there are works with data warehouse compression but no distribution. Therefore integrating these two technologies, in this thesis, we have first presented a distributed data warehouse architecture using compression-based multi-block vector structure for relational database. Using this architecture, we achieved a compression factor around 26.45 to 31.30 against the SQL Server for source relation. Afterwards, using our PCVDC Algorithm, the data cube was generated directly from this compressed form of data in a parallel environment without any decompression. Hence, it took significantly reduced creation and storage cost for the data cube. The PCVDC method was compared with BSF, Dwarf and Cubetrees. The cube size reduction effectiveness of PCDVC was around 18.88% to 59.26%, with an average of 39.77% better than that of the Dwarf cube and far more against BSF. The average speedup in the response time was 16.34 and 1.59 with respect to Cubetree and Dwarf Cube respectively. All of our experimental results are consistent with the analytical evaluation. Here we picked out the source relation as uniformly distributed, so that we can find out the upper bound of complexity for the data cube storage and query response time. If the

source relation is unevenly distributed then we would get a reduced complexity for both the case of compressed vector and data cube.

Our future work includes data migrating tools for compression based PCVDC system in a distributed and parallel environment. Using this tool, compressed data blocks can be migrated to remote servers and query can be answered from there without any decompression. Our architecture can also be used for parallel mining, where clustering and classification techniques can be applied in parallel onto the compressed vectors and virtual data cube. Back-up and recovery mechanism of these distributed data cube and compressed vectors can be considered in future. The system has been tested using synthetic data set. However, it can also be implemented in real life environment. We believe that the system will behave as our findings.

REFERENCES

[1] H. G. Molina, W. J. Labio, J. L. Wiener, and Y. Zhuge, "Distributed and Parallel Computing Issues in Data Warehousing", in *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, Mexico, 1998, pp. 77-85.

[2] R. Burns, D. Drake, and R. Winter, "Scaling the Enterprise Data Warehouse, Teradata's Integrated Solution," *A Winter Corporation White Paper*, Waltham, 2004, http://www.teradatalibrary.com/ pdf/teradata_7.pdf

[3] H. Ma, K. D. Schewe, and J. Zhao, "Cost Optimisation for Distributed Data Warehouses", in *Proceedings of the 38th Annual Hawaii International Conference on System Science,* Hawaii, 2005, pp. 283-288.

[4] J. L. Wiener, H. Gupta, W. J. Labio, Y. Zhuge, and H. G. Molina, "The WHIPS Prototype for Data Warehouse Creation and Maintenance," in *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Arizona, 1997, pp. 557–559.

[5] P. Howard, "A New Architecture for Data Warehousing", *Bloor Research White Paper*, June 2005. pp. 1-19. http://www.itdirector.com/research.php? viewpdf=742&mode=full.

[6] G. Kalra and D. Steiner, "Weather Data Warehouse: An Agent-Based Data Warehousing System," in *Proceedings of the 38th Annual Hawaii International Conference on System Sciences,* Hawaii, 2005, pp. 217-221.

[7] W. P. Cockshott, D. McGregor and J. Wilson, "High-Performance Operations Using a Compressed Database Architecture", *The Computer Journal*, Vol. 41, No. 5, 1998, pp. 283-296.

[8] A. S. M. L. Hoque, "Compression of Structured and Semi-Structured Information", Accepted for the degree of Ph. D, Dept. of Computer & Information Science, University of Strathclyde, Glasgow, UK, 2003.

[9] A. Thakar, A. Szalay, P. Kunszt and J. Gray, "Migrating a Multiterabyte Archive from Object to Relational Database", *IEEE Computer Society Digital Library*, 2003, Vol. 5, No. 5, pp. 16-29.

[10] M. A. Rouf, A. S. M. L. Hoque, "Dtabase Compression for Management of Terabyte Level Relational Data", in *Proceedings of 8th International Conference on Computer and Information Technology*, Dhaka, 2005, pp. 281-285.

[11] S. Agarwal, R. Agrawal., Deshpande, Gupta, Naughton, Ramakrishnan and Sarawagi, "On the Computation of Multidimentional Aggregates," in *Proceedings of the International Conference on Very Large Databases*, 1996, pp. 506-521.

[12] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, Y. Kotidis, "Dwarf: Shrinking the PetaCube," in *Proceedings of ACM SIGMOD*, Madison, USA, 2002, pp. 464-475.

[13] J. Feng, Q. Fang and H. Ding, "PrefixCube: Prefix-Sharing Condensed Data Cube," in *Proceedings of the 7th ACM International Workshop on Data Warehousing and OLAP*, Washington, DC, USA, 2004, pp. 38–47.

[14] Y. Chen, F. Dehne, T. Eavis and A. Rau-Chaplin, "Building Large ROLAP Data Cubes in Parallel," in *Proceedings of the International Database Engineering and Applications Symposium*, Montreal, 2004, pp. 367–377.

[15] R. Jin, K. Vaidyanathan, G. Yang and G. Agrawal, "Communication and Memory Optimal Parallel Data Cube Construction", *IEEE Transactions on Parallel and Distributed Systems*, 2005, Vol. 16, pp. 1105–1119.

[16] A. S. M. L. Hoque and F. H. Siddiqui, "Parallel Virtual Data Cube Construction from a Distributed Compressed", in *Proceedings of the International Conference on ICT for the Muslim World,* Malaysia, 2006, paper ID 52, pp. 1-8.

BIOGRAPHY

Fazlul Hasan Siddiqui received a B.Sc. (first class honors) in Computer Science & Information Technology from Islamic University of Technology (IUT), Bangladesh in 2002, and completed his coursework for the M.Sc. Engg. (first class honors) in Computer Science & Engineering in 2005 and is currently performing thesis work from Bangladesh University of Engineering & Technology (BUET), Bangladesh. He is a member of "Institute of Engineers Bangladesh (IEB)" and currently working as an Assistant Professor of Dhaka University of Engineering & Technology, Gazipur in Computer Science & Engineering Department. His research interest includes Data Warehouse, Data Mining, and Database Compression.

Abu Sayed Md. Latiful Hoque received his PhD in the field of Computer & Information Science from University of Strathclyde, Glasgow, UK in 2003 with Commonwealth Academic Staff Award. He obtained M.Sc. in Computer Science & Engineering and B.Sc. in Electrical & Electronic Engineering from Bangladesh University of Engineering & Technology (BUET) in 1997 and 1986 respectively. He has been working as a faculty member in the Department of Computer Science & Engineering at BUET since 1990 and currently his position is an Associate Professor. He is a Fellow of Institute of Engineers Bangladesh (IEB) and Bangladesh Computer Society. His research interest includes Data Warehouse, Data Mining, Information Retrieval and Compression in Database Systems.