High Performance Query Operations on Compressed Database

Ahsan Habib[1], A. S. M. Latiful Hoque[2] and Muhammad Shahidur Rahman[3]

[1]*Department of CSE, Metropolitan University, Sylhet, Bangladesh*

[2]*Department of CSE, Bangladesh University of Engineering and Technology Dhaka, Bangladesh*

[3]*Department of CSE, Shah Jalal University of Science and Technology Sylhet, Bangladesh*

ahabib@metrouni.edu.bd, asmlatifulhoque@cse.buet.ac.bd, msrahman.bd@gmail.com

Abstract

In this paper we have presented a loss-less compression technique namely H-HIBASE (compression enhancement of HIBASE technique using Huffman Coding). Due to disk based compression, H-HIBASE supports very large database with acceptable storage volume. Insertion, deletion and update mechanisms on the architecture have been presented and analyzed. The architecture executes query directly on compressed data and it is capable of executing all types of SQL queries. The experimental evaluation has been performed with synthetic and real data. The experimental result has been compared with DHIBASE and widely used Oracle database. We have evaluated the storage performance in comparison with DHIBASE and Oracle database. The storage performance that has been achieved in H-HIBASE is 25 to 40 percent better than the Oracle database for real and synthetic data. The query performance that has been achieved in H-HIBASE is 10 to 25 percent better than that of DHIBASE.

Keywords: *Database Compression, Huffman Coding, Query Performance, HIBASE*

1. Introduction

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

- Storage cost reduction
- Performance improvement

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

Most of the large databases are often in tabular form. The operational databases are of medium size whereas the typical size of fact tables in a data warehouse is generally huge [2]. These data are Write Once Read Many (WORM) types for further analysis.

Problem arises for high-speed access and high-speed data transfer. The conventional database technology cannot provide such performance. We need to use new algorithms and techniques to get attractive performance and to reduce the storage cost. High performance compression algorithm, necessary retrieval and data transfer technique can be a candidate solution for large database management system. It is difficult to combine a good compression technique that reduces the storage cost while improving performance.

1.1. Background

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

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

1.2. Objectives

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

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

1.3. Research Approach and Methodology

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

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

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

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

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

1.4. Storage Complexity

HIBASE:

$SCi = n * C_i$ bits
Where SCi = space needed to store column i in compressed form
n = number of records in the relation
Ci = number of bits needed to represent i^{th} attribute in compressed form
= $\lceil lg(m) \rceil$; where m is number of entries in the corresponding domain dictionary

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

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

H-HIBASE:

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

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

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

Compression Enhancement:

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

2. H-HIBASE: Implementation

2.1. H-HIBASE Dictionary

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

Huffman algorithm has been generated the position of values in the dictionary as well. The Huffman dictionary has generated as per following algorithm.

Algorithm 1: Huffman(C)

Huffman (C)

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

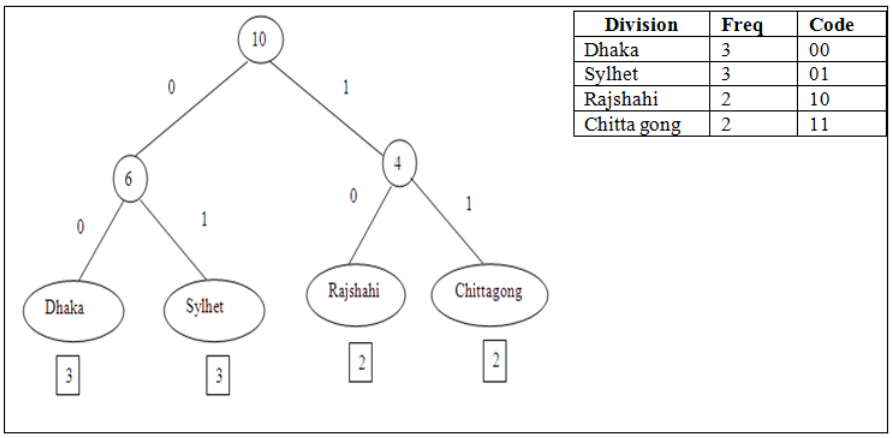
In the pseudocode that follows, we assume that C is a set of n strings and each string c € C is an object with a defined frequency f[c]. The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of |C| leaves and performs a sequence of |C| - 1 "meaning" operations to create the final tree. A min-priority queue Q, keyed on f, is used to identify the two least-frequent objects to merge together. The result of the merger of two objects is a new object whose frequency is the sum of the frequencies of the two objects that were merged [13].

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

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

2.2. H-HIBASE: Encoding

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



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

Figure 1. Construction of Huffman Tree for Division Column

The wighted external path length $\sum w_i l_i$ is minimum. For example, the Huffman tree corresponding to the source symbols { s_0, s_1, ……..,s_7} with the frequencies {3, 3, 2, 2} is shown in the Figure 1 the codeword set C{c_0,c_1,…….,c_7} is derived as {10, 11, 00, 01}. In addition, the codeword set is composed of a space with 2^d addresses, where d=2 is the depth of the Huffman tree.

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

Both C and C++ allow integer members to be stored into memory spaces smaller than the compiler would ordinarily allow [14]. These space-saving structure members are called *bit fields*, and their width in bits can be explicitly declared. The following structure has three bit-field members: kingdom, phylum, and genus, occupying 2, 6, and 12 bits respectively.

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

To store codeword we have declared an array of structure with bit field where data can be stored with 1 bit storage. This structure have 32 members variable named a, b, c,….., z, A, B, …, F and every member can be stored 1 bit. To put databits in this structure we have a function named putvalue (index_of_structure, data_variable, databit) which stores bit into the structure after reading the input from the dictionary [15].

Algorithm 2: Encode (index, name, databit, frequency)

Encode (Huffman_Dictionary hd)

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

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

In algorithm 2 it has been shown that index, frequency and codeword of a particular record has been read from the dictionary first. After that it is stored in the storage bitwise with the repetition of number of frequencies. And this process has continued until the last record of the dictionary. The required storage for the interval representation is n entries and the storage complexity is O (n).

2.3. Query Operation: Selection

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

Algorithm 3: Searching (Searched value)

```
Search (string searchedvalue)
1.    Input: The Searched Value
2.    Output: The matching interval
3.    BEGIN
4.      for  I ← 1 to number_of_coderword_in_dictionary
5.          if (inputdata=userdata)
6.                  position ← i
7.      sp ← findstartposition (position)
8.      ep ← findendposition (position)
9.      if the searched codeword is matched between the codeword of sp and ep
10.       print Found
11.     else
12.       print Not found
13.  END
```

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

2.4. Query Operation: Insertion

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

Algorithm 4: Insertion (Inserted value)

```
Insert (string InsertedValue)
1.   Take inserted value as input
2.   BEGIN
3.     Insert a new raw as the last tuple of input file
4.     Call Huffman (C)
```

5. Call Encode(Huffman_Dictionary)
6. END

2.5. Query Operation: Deletion

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

Algorithm 5: Deletion (Deleted value)
Delete (string DeletedValue)
1. Take deleted value as input
2. BEGIN
3. Search deleted item in the input database
4. if found delete the item by left shifting
5. Call Huffman (C)
6. Call Encode(Huffman_Dictionary)
7. else print "data cannot be deleted"
8. END

2.6. Query Operation: Update

Algorithm 6: Update (Old value, New value)
Update (string Oldvalue, string Newvalue)
1. Take updated value with old value as input
2. BEGIN
3. Search old value in the input file
4. If found update the input file by replacing new value with the old value
5. Call Huffman (C)
6. Call Encode (Huffman_Dictionary)
7. Else print "data cannot be updated"
8. END

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

2.7. Query Operation: Aggregate Function

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

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

2.7.1. Count: *Select branch_name, count (branch_name) from account.*

Algorithm 7: Count ()
Count ()
1. *Initialize count=0*
2. *Read dictionary*
3. *Loop until finish the number of tuple*
4. *Count++*
5. *Print Count*

Algorithm 7 is indicated that record has been counted from the dictionary, for each frequency it increases count by 1 until reach the last frequency.

2.7.2. Sum/Avg: *Select branch_name, sum (balance) from account.*

Algorithm 8: Sum ()/Avg ()
Sum/Avg ()
1. *Read dictionary*
2. *Initialize sum=0*
3. *Put number in an array*
4. *For 1 to size of array (count)*
5. *Sum=sum + number*
6. *Print sum*

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

2.7.3. Max/Min: *Select branch_name, max (balance) from account.*

Algorithm 9: Max ()/ Min ()
Max/Min ()
1. *Read dictionary*
2. *Initialize maximum=0*
3. *Put number in an array*
4. *For 1 to size of array (count)*
5. *If (current value>maximum value)*
6. *Maximum = current value*
7. *Print maximum*

Algorithm 9 has been used to find the maximum number. In the algorithm it has been shown that any larger number is replaced by smaller number in the array.

3. Result and Discussion

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

3.1. Storage Requirement

Storage requirement in different technique for real data has been shown in figure. Figure shows that the H-HIBASE has greater compression capability than DHIBASE which is more than 30%. Higher storage requirement has been avoided by using Huffman code-words in H-HIBASE technique. Moreover high performance has been ensured as most repeated attribute values get more weight and entered first in the dictionary i.e. domain dictionary values sorted in such a way that frequently occurred values accessed first than the rare values.

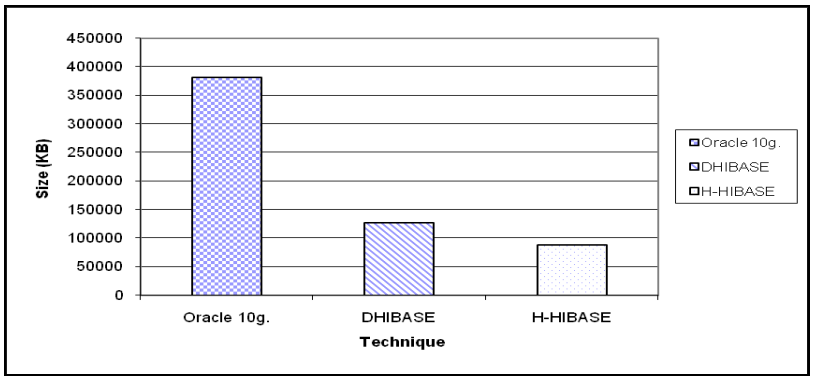


Figure2. Storage of Real Data in H-HIBASE, DHIBASE and Oracle 10g

Figure 2 shows the comparison of storage size among Oracle database, DHIBASE, and H-HIBASE. To store same number of record it is required approximately 380 MB, 125 MB, and 85 MB in Oracle 10g, DHIBASE, and H-HIBASE respectively. H-HIBASE technique has more compression capability than any other existing systems.

Figure 3 indicates the storage comparison between H-HIBASE and DHIBASE. In this figure H-HIBASE has better compression capability with the rate of more than 30%.

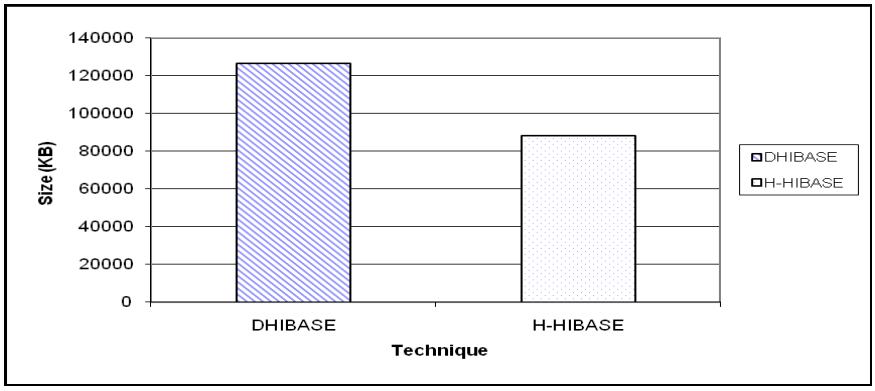


Figure 3. Storage of Real Data in H-HIBASE and DHIBASE

3.2. Query Performance

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

3.2.1. Single Column Projection: We have executed the following query and the result is shown in Figure 4.

select Name from Customer

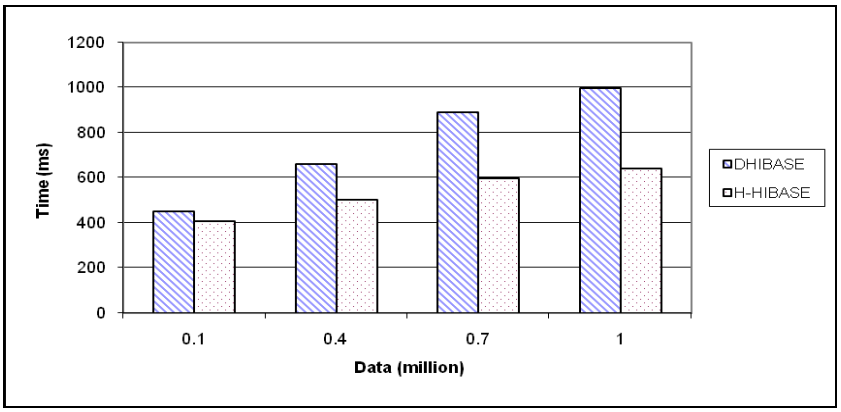


Figure 4. Single Column Projection

Figure 4 shows that H-HIBASE is faster than that of DHIBASE in case of projection operation. This is obvious because H-HIBASE stores data in compressed form with minimum storage. Therefore, to find a particular record it requires to search a smaller amount space. This act as the main reason of speed-gain in H-HIBASE system.

3.2.2. Single Predicate Selection

We have executed the following query and the result is shown in figure 5.

select Name from Customer where City = "Dhaka"

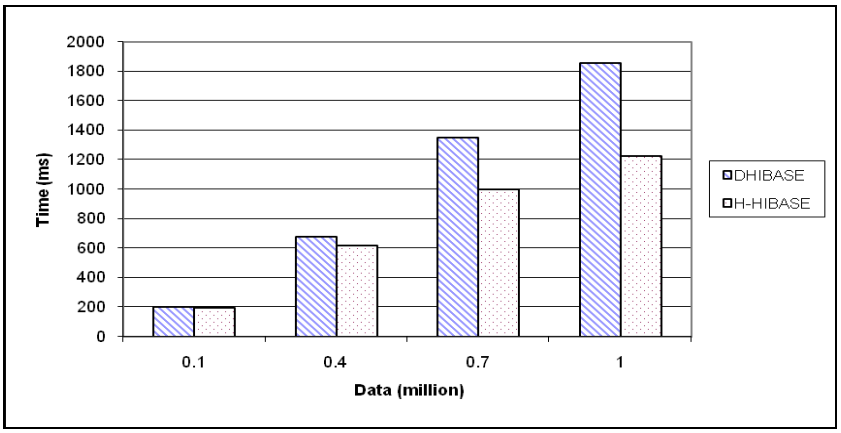


Figure 5. Single Predicate Selection

Figure 5 shows that H-HIBASE does not have better performance than DHIBASE in case 0.1 million to 0.4 million records but faster in case 0.7 million and 1.0 million records. In case of 1 million records, it first reads most repetitive values from the dictionary, and takes a reduced amount of time to access it from the storage. The processing speed of predicate selection query is enhanced because queries specify operations only on a subset of domains.

In a column-wise database only the specified column needs to be accessed. This requires only a fraction of the data that was required during processing by rows.

3.2.3. Five Percent Selectivity

We have executed the following query and the result is shown in figure 6.

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

Figure 6 shows that H-HIBASE performs better performance than DHIBASE in case of 5% selectivity. This is because within this 5% data, there are large numbers of repetitions.

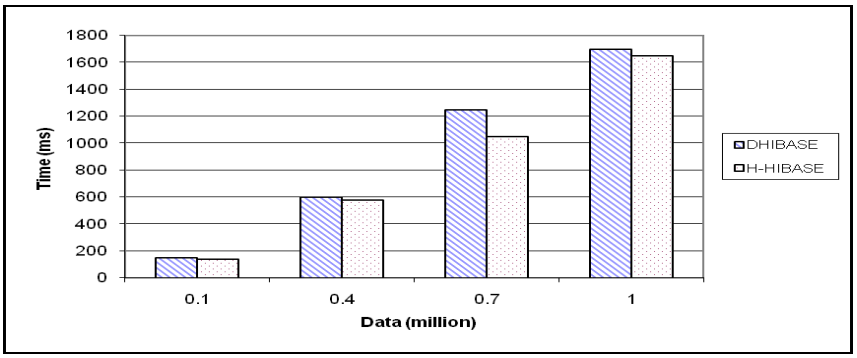


Figure 6. 5% Selectivity

3.2.4. Aggregate Function: Count

For aggregation queries we have considered the following relation:

account(account_no, branch_name, balance).

We have executed the following query and the result is shown in Figure 7.

Select branch_name, count (branch_name) from account.

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

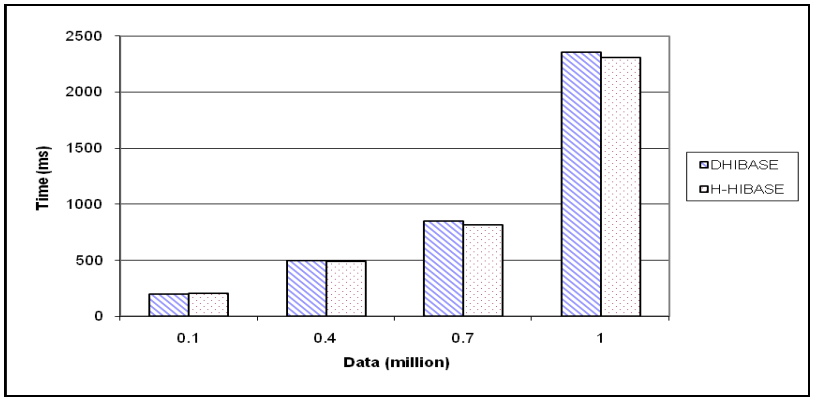


Figure 7. Aggregate Function: Count

3.2.5. Aggregate Function: Max/Min/Sum/Avg

We have calculated the following queries and the result is shown in figure 8
select account_no, max (balance) from account group by account_no
select account_no, sum (balance) from account group by account_no
select account_no, avg (balance) from account group by account_no

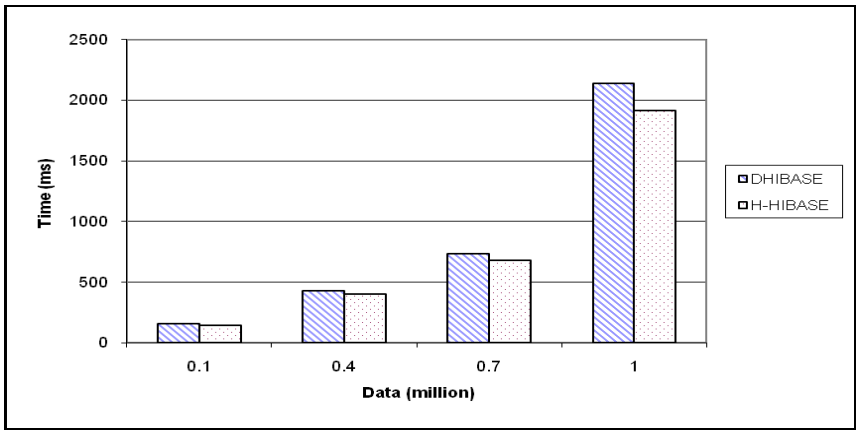


Figure 8. Aggregate Function: Max/Min/Sum/Avg

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

4. Conclusions

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

4.1. Fundamental Contributions of the Research

- The main contribution of this research is to develop a compression technique that is enhancement of HIBASE technique using Huffman coding (H-HIBASE) with better compression capability.
- Compressed data are stored using the H-HIBASE architecture with disk support. This overcomes the scalability problems of the memory resident DBMS.
- Considerable storage reduction has been achieved using the H-HIBASE architecture. The experimental results show that H-HIBASE architecture is 15 to 35 times space efficient than that of HIBASE and DHIBASE.

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

References

[1] C. B. Tashenberg, "Data management isn't what it was", Data Management Direct Newsletter, **(2002)** May 24.
[2] M. A. Rouf, "Scalable storage in compressed representation for terabyte data management", M. Sc. Thesis, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh, **(2006)**.
[3] G. V. Cormack, "Data compression on a database system", Communication of the ACM, vol. 28, no. 12, **(1985)**, pp 1336–1342.
[4] S. Helmer, T. Westmann, D. Kossmann and G. Moerkotte, "The implementation and performance of compressed databases", SIGMOD Record, vol. 29, no. 3, **(2000)**, pp. 55–67.
[5] M. A. Roth and S. J. V. Horn, "Database compression", SIGMOD Record, vol. 22, no. 3, **(1993)**, pp. 31–39.
[6] G. Graefe and L. Shapiro, "Data compression and database performance", ACM/IEEE-CS Symposium on Applied Computing, **(1991)** April, pp. 22-27.
[7] Oracle Corporation, "Table compression in Oracle 9i: a performance analysis, an Oracle whitepaper", http://otn.oracle.com/products/bi/pdf/o9ir2_ compression_performance_twp.pdf.
[8] R. Ramakrishnan, J. Goldstein and U. Shaft, "Compressing relations and indexes", Proceedings of the IEEE Conference on Data Engineering, Orlando, Florida, USA, **(1998)** February, pp. 370–379.
[9] M. Poess and D. Potapov, "Data compression in Oracle", Proceedings of the 29 VLDB Conference, Berlin, Germany, September, **(2003)**, pp. 937-947.
[10] A. Silberschatz, H. F. Korth and S. Sudarshan, "Database system concepts", 5th Edition, McGraw-Hill **(2006)**.
[11] D. McGregor, W. P. Cockshott and J. Wilson, "High-performance operations using a compressed architecture", The Computer Journal, vol. 41, no. 5, **(1998)**, pp. 283– 296.
[12] A. S. M. L. Hoque, "Compression of structured and semi-structured information", Ph. D. Thesis, Department of Computer and Information Science, University of Strathclyde, Glasgow, UK, **(2003)**.
[13] T. H. Coreman, C. E. Leiserson, R. L. Rivest and C. Stein, "Introduction to Algorithms", second edition, pp. 387-388.
[14] Declaring and Using Bit Fields in Structures: http://publib.boulder.ibm.com /infocenter/macxhelp /v6v81/index.jsp?%20 topic=%2Fcom.ibm.vacpp6m. doc %2Flanguage%2Fref%2Fclrc03 def bitf.htm, **(2011)**, July 31.
[15] A. Habib, A. S. M. L. Hoque, M. R. Hussain and S. Ismail, "Compression Enhancement of HIBASE Technique Using Huffman Coding", Proceedings of 14th International Conference on Computer and Infonnation Technology (ICCIT 2011), **(2011)** December 22-24, Dhaka, Bangladesh.