VParC: A Compression Scheme for Numeric Data in Column-Oriented Databases

Ke Yan[1], Hong Zhu[1], and Kevin Lü[2]
[1]School of Computer Science and Technology, Huazhong University of Science and Technology, China
[2]College of Business, Arts and Social Sciences, Brunel University, UK

Abstract: *Compression is one of the most important techniques in data management, which is usually used to improve the query efficiency in database. However, there are some restrictions on existing compression algorithms that have been applied to numeric data in column-oriented databases. First, a compression algorithm is suitable only for columns with certain data distributions not for all kinds of data columns; second, a data column with irregular distribution is hard to be compressed; third, the data column compressed by using heavyweight methods cannot be operated before decompression which leads to inefficient query. Based on the fact that it is more possible for a column to have sub-regularity than have global-regularity, we developed a compression scheme called Vertically Partitioning Compression (VParC). This method is suitable for columns with different data distributions, even for irregular columns in some cases. The more important thing is that data compressed by VParC can be operated directly without decompression in advance. Details of the compression and query evaluation approaches are presented in this paper and the results of our experiments demonstrate the promising features of VParC.*

Keywords: *Column-stores, data management, compression, query processing, analytical workload.*

Received August 28, 2013; accepted October 23, 2014; published online March 8, 2015

1. Introduction

Column-oriented database systems (often refer to as column-stores) that store their content by columns (such as MonetDB, Sybase IQ and C-Store) have been shown to perform better than traditional row-oriented database management systems (refer to as row-stores, such as Oracle, IBM DB2 and SQL Service) on analytical workloads in data warehouses, decision support and business intelligence applications [2]. Compression is a technique known for improving system performance for DBMSs and other applications [6, 9]. Column-stores are more suitable to compression schemes than row-stores [2] due to its storage mode. Compression techniques further improve the system performance of column-stores and gradually become one of the indispensable techniques.

Recently, there are several studies reported on compression algorithms for column-stores [1, 12]. Numeric type (including: Integer, float, decimal and date typed data) is a commonly used data type and is different from other data types (e.g., string type). However, there are some limitations in the existing compression methods for numeric data columns in column-stores:

- A compression algorithm is suitable only for columns with certain data distributions. The size of a data column compressed with unsuitable schemes could even become larger than that of the original.
- A data column with irregular distribution is hard to be compressed by using any existing compression algorithms.
- Without direct operations on compressed data, compression becomes a trade off: Systems must pay the extra CPU cost to decompress in exchange for the I/O savings of reading less data from storage [3, 8].

In order to, alleviate the limitations described above, we examined the issues related the regularities in data distributions and found a new regularity in data distribution called Sub-Regularity. Consequently, a compression scheme called Vertically Partitioned Compression (VParC) is proposed, in which sub-regularity are used to compress numeric data in column-oriented databases and the direct operations on compressed data columns are supported. In summary, the main contributions of this study are:

- We introduced an approach to measure the regularities of data distribution through extracting three characteristic values for capturing the features of data distributions.
- Base on the regularities of data distribution, we proposed and implemented a compression scheme for column-stores, called VParC.
- We developed direct operations on compressed data.
- Experimental studies have been performed. Our experiments assessed the compression ratio, time required for compression and decompression, as well as issues related to direct operations.

The rest of this paper is organized as follows: Section 2 shows the related work. Definitions and analysis related to the regularity of data distribution are presented in section 3. In section 4, we give the introduction about the new compression algorithm VParC and the direct searching on compressed data column. Section 5 shows the experiments results. Finally, section 6 concludes the paper.

2. Related Works

Loss and lossless compression techniques are widely used to save storage space and improve access speed [4, 10, 13, 16]. Lossless compression techniques, including lightweight and heavyweight compression techniques, had been applied to column-stores [1, 3]. However, there are some limitations or restrictions on applying these compression techniques directly to numeric data columns.

One kind of lightweight compression technique cannot be suitable for all kinds of data columns. Run-Length Encoding (RLE) [17], where repeats of the same element are expressed as (value, run-length) pairs, is an attractive approach for compressing sorted data in a column-store. Nevertheless, it is only suited for columns that have reasonable-sized runs of the same value. Dictionary encoding and its improved algorithms are perhaps the most prevalent compression scheme in use in data management today [5, 15]. In these schemes, frequent patterns are replaced with smaller codes. Bit packing is then used on top of dictionaries to further compress the data [7]. However, this approach is only appreciated for columns that have a limited amount of different values. FOR compression scheme [20] uses a base value, a frame of reference and stores all values as offsets from the reference value. This method is useful when the values are not wide spread, since this will result in small offset values that can be represented by a small number of bits. Additionally, it is not suitable for float data column. Bitmap encoding is another widely used compression scheme [14]. It uses bit vectors to encode the values. One bit vector is required for each value, which makes it only useful when there is a limited amount of different values. Thus, each method is suitable for only one case. And, if a column without any of the characteristics described above, it cannot be compressed by any lightweight techniques. In order to address this problem, we proposed VParC to fit for all kinds of data columns.

Ziv and Lempel [19] Encoding is the most widely used Heavyweight technique for lossless compression. The basic idea of this method is to parse the input sequence into non-overlapping blocks of different lengths. Then, a dictionary of blocks is constructed, which leads to the subsequent appearances of these blocks being replaced by a pointer to an earlier occurrence of the same block. However, recent researches [1] illustrated that the data columns compressed using heavyweight compression algorithms are difficult to be operated directly. That is, the data columns compressed by using heavyweight algorithms should be decompressed before being operated. This will degrade the performance in the case that the extra CPU cost of decompression is more than the saving I/O cost of reading less data from storage. Different from heavyweight techniques, VParC could support the directly operations on compressed data, thus, we propose the corresponding method of directly query for VParC.

In C-Store [1, 3] each column is divided into blocks and each block is compressed using the decision tree compression scheme. This approach is used to aid the database designer to decide how to compress particular columns. However, this method also cannot support the direct operations on compressed data, just like heavyweight technique. And it costs much time on the decision of compression method. Our method is more efficient in compression than C-Store which can be illustrated in our experiment.

3. Regularity of Data Distribution

Compression algorithms are useful only when data distributions of a column contain certain regularities. Thus, issues related to data distribution regularities are important to compression.

3.1. Vertically Partitioned

Different types of data may be stored in different ways. For example, float data is stored using 4 bytes, double data is stored using 8bytes and integer data is stored using 4 or 8 bytes; decimal data and date data in different system may be stored in different ways. However, data in different types can be vertically partitioned into several sub-columns.

- *Example* 1: Figure 1 shows the vertical partitioning of an integer 32bit data column. This integer 32bit column is partitioned vertically into 4 sub-columns. Obviously, RLE is quite suitable for the first sub-column.

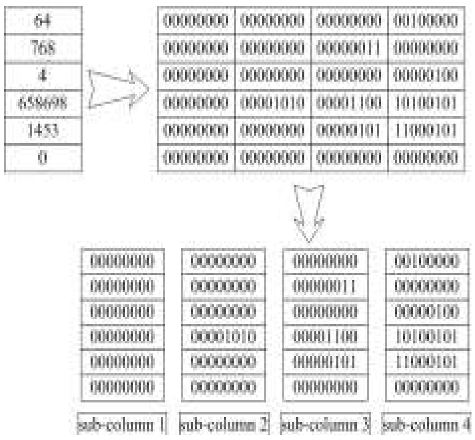

Figure 1. Vertical partition of an integer 32bit column.

3.2. Characteristic Values

Different compression schemes will be fit for different data columns. RLE is suitable for columns with few runs. The columns which have less number of different values prefer to dictionary and bitmap encoding. If the values of a column are not wide spread, FOR would be more appropriated. Thus, three key characteristic values of a column determine the compression effectiveness of these algorithms.

- ***Definition*** **1**. Difference Degree: The number of different values in a column.

- ***Definition* 2**. Continuity Degree: The number of runs in a column. A run is a sequence in which the same data value occurs consecutively.
- ***Definition* 3**. Max Distance: The difference between the maximal value and the minimal value in a column.

- *Example* 2: Figure 2 shows a column with 15 values. Difference Degree of this column is 4, because there are 4 different values in this column. These 4 different values are 0, 53, 52 and 175. Continuity Degree of this column is 7, since there are 7 runs in all (the first run is 0, its length is 3; the second run is 53, its length is 1 and so on). The max distance is 175. Because the maximal value in this column is 175 and the minimal value is 0, so the difference between them is 175. These characteristic values of a column can be obtained by traversing all the values in a column.

0	0	0	53	52	175	175	175	52	53	53	53	53	53	53

Figure 2. An instance of a column.

3.3. Compression Ratios

A compression algorithm is suitable for a column only when the compression ratio is smaller than 1. Clearly, the smaller the compression ratio is the more suitable the compression algorithm will be for a data column. Assumed that, the number of values in a data column t is $n(t)$ and each value is assigned $b(t)$ bits.

RLE [17] compresses runs in a column to a compact singular representation. A run is replaced with a tuple (value, run-length), where each element of a tuple is given a fixed number of bits. When a data column t is compressed using RLE, the element value is given $b(t)$ bits. The element run-length is assumed to be g bits. $dg_c(t)$ denotes the Continuity Degree of the column t. $space_{rle}(t)$ denotes the data size of the column t compressed by RLE and is defined as follows:

$$space_{rle}(\mathrm{t}) = dg_c(\mathrm{t}) \times (g + b(\mathrm{t})) \tag{1}$$

FOR [20] uses a base value, a frame of reference and stores all values as offsets from the base value. $dis_m(t)$ denotes the max distance of a column t. Thus, when the minimum value in a column t is the base value, the offsets from the base value need $\lceil log_2\ dis_m(t) \rceil$ bits. $space_{for}(t)$ denotes the data size of the column t compressed by FOR and is defined as follows:

$$space_{for}(\mathrm{t}) = n(\mathrm{t}) \times \left\lceil \log_2 dis_m(\mathrm{t}) \right\rceil + b(\mathrm{t}) \tag{2}$$

In bitmap encoding [14] a bit-string is associated with each value with a '1' in the corresponding position if that value appeared at that position and a '0' otherwise. $dg_d(t)$ denotes the difference degree of a column $t.space_{bit}(t)$ denotes the data size of the column t being compressed using bitmap encoding and is defined as follows:

$$space_{bit}(\mathrm{t}) = n(\mathrm{t}) \times dg_d(\mathrm{t}) + b(\mathrm{t}) \times dg_d(\mathrm{t}) \tag{3}$$

Dictionary compression schemes [7, 15] are perhaps the most prevalent compression schemes found in data management systems today. These schemes replace each value in a column t with a smaller dictionary code. Each code needs $\lceil log_2\ dg_d\ (t) \rceil$ bits. $space_{dict}(t)$ denotes the data size of the column t compressed by dictionary (with the addition of bit packing) and is defined as follows:

$$space_{dict}(\mathrm{t}) = b(\mathrm{t}) \times dg_d(\mathrm{t}) + n(\mathrm{t}) \times \left\lceil \log_2 dg_d(\mathrm{t}) \right\rceil \tag{4}$$

$space_{no}(t)$ denotes the data size of a column t without compression. It is equal to $n(t)*b(t)$. $r_{rle}(t)$, $r_{for}(t)$, $r_{bit}(t)$ and $r_{dict}(t)$ denote the compression ratio of RLE, FOR, Bitmap and Dictionary encoding for a column t and equal to $space_{rle}(t)$, $space_{for}(t)$, $space_{bit}(t)$ and $space_{dict}(t)$ divided by $space_{no}(t)$ respectively. Thus, based on Equations 1, 2, 3 and 4, compression ratios of each algorithm are:

$$r_{rle}(t) = \frac{space_{rle}(t)}{space_{no}(t)} = \frac{dg_C(t)}{n(t)} \times \left(1 + \frac{g}{b(t)}\right) \tag{5}$$

$$r_{for}(t) = \frac{space_{for}(t)}{space_{no}(t)} = \frac{\left\lceil \log_2 dis_m(t) \right\rceil}{b(t)} + \frac{1}{n(t)} \tag{6}$$

$$r_{bit}(t) = \frac{space_{bit}(t)}{space_{no}(t)} = \frac{dg_d(t)}{b(t)} + \frac{dg_d(t)}{n(t)} \tag{7}$$

$$r_{dict}(t) = \frac{space_{dict}(t)}{space_{no}(t)} = \frac{\left\lceil \log_2 dg_d(t) \right\rceil}{b(t)} + \frac{dg_d(t)}{n(t)} \tag{8}$$

3.4. Global-Regularity and Sub-Regularity

Based on the compression ratios of these lightweight compression algorithms, we define the data distribution regularity of a data column is proposed.

3.4.2. RLE-Regularity and Sub-RLE-Regularity

- ***Definition* 4**. RLE-Regularity: Given a data column t, if $r_{rle}(t)<1$, we call that the column t has the $r_{rle}(t)$ RLE-Regularity.
- ***Definition* 5**. Sub-RLE-Regularity: A column t is vertically partitioned into s sub-columns, which are denoted as $t(1)$, …, $t(s)$. If existing i ™1, 2, ..., s}, the i^{th} sub-column $t(i)$ has RLE-Regularity, t is considered to have Sub-RLE-Regularity.

- ***Lemma* 1**. Given a data column t with $n(t)$ elements, each element value is stored in $b(t)$ bits and each run length is stored in g bits. The probability of the column t having RLE-Regularity is:

$$p_{rle} = \sum_{i=1}^{\beta_{rle}} \frac{C_{n(t)-1}^{i-1} \times 2^{b(t)} \times \left(2^{b(t)} - 1\right)^{i-1}}{2^{b(t) \times n(t)}} \quad \text{where} \quad \beta_{rle} = \left\lfloor \frac{n(t) \times b(t)}{b(t) + g} \right\rfloor \tag{9}$$

- ***Proof***: According to Equation 5, RLE-Regularity implies that:

$$r_{rle}(t) = \frac{dg_c(t)}{n(t)} \times \left(1 + \frac{g}{b(t)}\right) < 1$$

That is $dg_c(t) < \frac{n(t) \times b(t)}{b(t) + g}$

dg$_c$(*t*) is an integer since it is the continuity degree of column *t*. We can obtain that:

$$dg_c(t) \le \beta_{rle}, \quad \text{where} \quad \beta_{rle} = \left\lfloor \frac{n(t) \times b(t)}{b(t) + g} \right\rfloor$$

There are $2^{b(t)}$ possible values for each element in column *t* since, each element is stored in *b*(*t*) bits. Thus, the number of possible outcomes of column *t* is $2^{b(t)n(t)}$.

dg$_c$(*t*)=*i* means that the number of runs in column *t* is *i*. Thus, the number of possible outcomes that *dg*$_c$(*t*) equals to *i* is:

$$C_{n(t)-1}^{i-1} \times 2^{b(t)} \times \left(2^{b(t)} - 1\right)^{i-1}$$

Where C_n^m denotes the number of combinations. Further, the probability of *dg*$_c$(*t*) equal to *i* is:

$$p\left(dg_C(t) = i\right) = \frac{C_{n(t)}^{i-1} \times 2^{b(t)} \times \left(2^{b(t)} - 1\right)^{i-1}}{2^{b(t) \times n(t)}}$$

The probability of a column *t* having RLE-regularity is the probability of *dg*$_c$(*t*) smaller and equal to β_{rle}. That is:

$$p_{rle} = p\left(dg_c(t) \le \beta_{rle}\right) = \sum_{i=1}^{\beta_{rle}} p\left(dg_c(t) = i\right)$$

As a result, we can obtain the Equation 9.

3.4.2. BM-Regularity and Sub-BM-Regularity

- ***Definition* 6**. BM-Regularity: Given a data column *t*, if $r_{bit}(t)<1$, the column *t* has the $r_{bit}(t)$ BM-Regularity.
- ***Definition* 7**. Sub-BM-Regularity: A column *t* is vertically partitioned into s sub-columns. If existing *i*, the i^{th} sub-column *t*(*i*) has BM-Regularity, the column *t* is considered to have Sub-BM-Regularity.

- ***Lemma* 2**. Given a data column *t* with *n*(*t*) elements, each element value is stored in *b*(*t*) bits. The probability of the column t having BM-regularity is:

$$p_{bit} = \sum_{i=1}^{\beta_{bit}} \frac{P_{2^{b(t)}}^{i} \times S_{n(t)}^{i}}{2^{b(t) \times n(t)}} \tag{10}$$

Where $\beta_{bit} = \left\lfloor \frac{b(\text{t}) \times n(\text{t})}{b(\text{t}) + n(\text{t})} \right\rfloor$

S_n^m: Denotes a stirling number of the second kind, and P_n^m: Denotes the number of m-permutations of *n*.

- ***Proof***: According to Equation 7, the situation in which column *t* has BM-Regularity implies that:

$$r_{bit}(t) = \frac{dg_d(t)}{n(t)} + \frac{dg_d(t)}{b(t)} < 1$$

That is $dg_d(t) < \frac{b(t) \times n(t)}{b(t) + n(t)}$

Where *dg*$_d$(*t*) must be an integer since, it is the Difference Degree of column *t*. We can obtain that:

$$dg_d(t) < \beta_{bit}, \quad \text{where} \quad \beta_{bit} = \left\lfloor \frac{b(t) \times n(t)}{b(t) + n(t)} \right\rfloor$$

dg$_d$(*t*)=*i* means that the number of different elements in column *t* is *i*. Thus, the number of possible outcomes that *dg*$_c$(*t*) equals to *i* is:

$$P_{2^{b(t)}}^{i} \times S_{n(t)}^{i}$$

Because the number of possible outcomes of column t is $2^{b(t)n(t)}$, the probability of *dg*$_d$(*t*) equal to *i* is:

$$p(dg_d(t) = i) = \frac{P_{2^{b(t)}}^{i} \times S_{n(t)}^{i}}{2^{b(t) \times n(t)}}$$

Probability of column *t* having BM-Regularity is the probability of *dg*$_d$(*t*) smaller and equal to β bit. That is:

$$p_{bit} = p\left(dg_d(t) \le \beta_{bit}\right) = \sum_{i=1}^{\beta_{bit}} p\left(dg_d(t) = i\right)$$

As a result, we can obtain the Equation 10.

3.4.3. DICT-Regularity and Sub-DICT-Regularity

- ***Definition* 8**. DICT-Regularity: Given a data column *t*, if $r_{dict}(t)<1$, the column *t* has the $r_{dict}(t)$ DICT-Regularity.
- ***Definition* 9**. Sub-DICT-Regularity: A column *t* is vertically partitioned into *s* sub-columns. If existing i, the i^{th} sub-column *t*(*i*) has DICT-Regularity, the column *t* is considered to have Sub-DICT-Regularity.

- ***Lemma* 3**. Given a data column *t* with *n*(*t*) elements, and each element value is stored in *b*(*t*) bits. The probability of the column t having DICT-Regularity is:

$$p_{dict} = \sum_{i=1}^{\beta_{dict}} \frac{P_{2^{b(t)}}^{i} \times S_{n(t)}^{i}}{2^{b(t) \times n(t)}} \tag{11}$$

Where $\beta_{dict} = \lfloor \beta \rfloor$, and β : Is the approximate solution of the equation:

$$n(t) \times log_2 dg_d(t) + b(t) \times dg_d(t) = n(t) \times b(t)$$

- ***Proof***: According to Equation 8, the situation in which column *t* has DICT-Regularity implies that:

$$r_{dict}(t) = \frac{\lceil \log_2 dg_d(t) \rceil}{b(t)} + \frac{dg_d(t)}{n(t)} < 1$$

If $r_{dict}(t)=1$, we can get an equation:

$$n(t) \times log_2 dg_d(t) + b(t) \times dg_d(t) = n(t) \times b(t)$$

It is an non-linearequation about *dg*$_d$(*t*) and we can obtain the approximate solution β using the method of

Newton. There are three reasons for $dg_d(t)$ satisfying $dg_d(t) \leq \beta_{dict}$, where $\beta_{dict}=\lfloor\beta\rfloor$: $dg_d(t)$ is an integer; $r_{dict}(t)<1$; $r_{dict}(t)$ is monotonically increasing at $dg_d(t)$. Because the probability of the column t having DICT-Regularity is the probability of $dg_d(t)$ smaller and equal to β_{dict}, we can get that:

$$p_{dict}=p\left(dg_d(t)\leq \beta_{dict}\right)=\sum_{i=1}^{\beta_{dict}} p\left(dg_d(t)=i\right)$$

As a result, we can obtain the Equation 11.

3.4.4. For-Regularity and Sub-for-Regularity

- ***Definition*** **10**. For-Regularity: Given a data column t, if $r_{for}(t)<1$, we call that the column t has the $r_{for}(t)$ for-regularity.
- ***Definition*** **11**. Sub-For-Regularity: A column t is vertically partitioned into s sub-columns. If existing i, the i^{th} sub-column $t(i)$ has For-Regularity, the column t is considered to have sub-for-regularity.

3.5. Global-Regularity and Sub-Regularity

For a data column t, it could have several kinds of regularities. For example, t has RLE-Regularity and BM-Regularity simultaneously.

- ***Definition*** **12**. *Global-Regularity*: If a data column t has one or more kinds of regularities, that is, $\lambda(t)=min\{r_{rle}(t), r_{bit}(t), r_{dict}(t), r_{for}(t)\}<1$, the column t has $\lambda(t)$ Global-Regularity.

If a data column has Global-Regularity, it can be compressed. It is clearly that the smaller the $\lambda(t)$ of a column is the more regular the column is and the more effective that the column can be compressed. If $\lambda(t)\geq 1$, we say that the column t is irregular.

- ***Definition*** **13**. Sub-Regularity: If a data column t has Sub-RLE-Regularity, Sub-BM-Regularity, Sub-DICT-regularity or sub-for-regularity, it is called to have sub-regularity.

A column t with Sub-Regularity can be compressed after vertically partitioned into sub-columns. It is possible that column t has several kinds of sub-regularities. For example, it could have sub-RLE-regularity and sub-bm-regularity.

The key issue to partition a column into sub-columns is how many sub-columns a column should be vertically partitioned into. Too many sub-columns a column is vertically partitioned into would lead to more compression and decompression costs. On the other hand, if a column is vertically partitioned into too few sub-columns, there is no much help to compression. Byte is the basic unit of data processing. Thus, a column could be vertically partitioned by byte.

Traditional compression methods take advantage of Global-Regularity to compress data columns. Obviously, if a column t has For-Regularity, it must have Sub-For-Regularity. Additionally, according to the lemmas 1, 2 and 3, a data column t is more likely to have Sub-RLE-Regularity, Sub-BM-Regularity and Sub-DICT-Regularity than to have RLE-Regularity, BM-Regularity and DICT-Regularity. That is, sub-columns that are vertically partitioned from a column t are more regular than the column t itself. Thus, we utilize Sub-Regularity to compress.

4. Compression and Direct Searching

Based on the analysis and proof above, we propose a compression algorithm called VParC. A column that is compressed using VParC is partitioned into several sub-columns. The key issue of searching directly on the compressed column is how to obtain the results by means of the direct searching on the compressed sub-columns.

4.1. Compression Procedures of VParC

The procedure of compressing a data column t using VParC can be divided into 3 steps:

1. Vertically partition the data column t into several sub-columns by byte, and each sub-column is a byte typed column.
2. Obtain three characteristic values of each sub-column. Three characteristic values of each sub-column can be obtained by traversal of each sub-column.
3. Compress each sub-column using suitable light-weight compression algorithms.

Sub-columns with different regularities need to be compressed using different compression algorithms. A key issue is how to choose a suitable compression algorithm for a sub-column $t(i)$. Based on characteristic values of a sub-column $t(i)$, $r_{rle}(t(i))$, $r_{for}(t(i))$, $r_{bit}(t(i))$ and $r_{dict}(t(i))$ can be obtained and the compressed size of the sub-column $t(i)$ can be obtain.

The purpose of compression is to improve the performance of searching [8, 11]. Thus, we choose the algorithm that is the most beneficial for searching. We assume that α is a constant factor regarding disk I/O capability. γ_{rle}, γ_{for}, γ_{dic} and γ_{bit} denote the cost of retrieving on data compressed by RLE, dictionary, FOR and Bitmap encoding respectively. Thus, an approximation of the time for searching on the sub-column $t(i)$ compressed by four kinds of compression methods can be obtained:

$$S(X)=ratio_X\times n\times(\alpha\times\gamma_X) \qquad (12)$$

Where X is RLE, FOR, dictionary or Bitmap encoding. Thus, we choose X to compress $t(i)$ where X leads to the minimum $S(X)$.

4.2. Equivalent Search and Rang Search

There are two kinds of searching on numeric data in column-stores, which is different from string typed data: Equivalent search and range search. An equivalent search means to find values in a column that are equal to a given value a. A range search means to find values in a column that are in a given interval. There are several patterns of intervals. Because the process of range search is similar for different kinds of intervals, only open interval (a, b) is discussed in this paper.

A column t is partitioned vertically into $u(t)$ sub-columns by byte. Accordingly, a value x in the column is partitioned into $u(t)$ bytes (called sub-values) and it is represented as $<x_1, x_2, \ldots, x_{u(t)}>$. A given value a is partitioned into $u(t)$ bytes as well, represented as $<a_1, a_2, \ldots, a_{u(t)}>$. x is equal to a if and only if each sub-value of x is equal to each corresponding sub-value of a, that is $x_1=a_1$, $x_2=a_2$, …, and $x_{u(t)}=a_{u(t)}$.

In a rang search, the lower bound and upper bound of an interval (a, b) can be partitioned into $u(t)$ sub-values, which are denoted as $<a_1, a_2, \ldots, a_{u(t)}>$ and $<b_1, b_2,\ldots, b_{u(t)}>$ separately. Result of range search can be obtained by means of comparing x_i with a_i and b_i (i=1, 2, …, $u(t)$). However, columns with different data types in different systems are stored in different ways. Thus, the methods of range search are varied depending on data types and storage formats. In this section, searching process is shown based on IEEE standard format.

4.3. Storage Format of Data Columns

An integer (32bits) 32 stored in IEEE standard format and in little-endian computer byte ordering is shown in Figure 3. It is stored in the form of complement.

11111111	11111111	11111111	11100000

Figure 3. An instance of storage format of integer data (32bits).

A float (32bits) -0.125 (to present as binary number is $-1.0*2^{-11}$) in IEEE standard format and in little-endian computer byte ordering is shown in Figure 4. From left to right, the first bit is sign bit; the next eight bits represent the shift code of exponent (means adding 127 to exponent) and the left bits represent mantissa.

10111110	00000000	00000000	00000000

Figure 4. An instance of storage format of float data (32 bits).

4.4. Range Search

In the implementation of range search, the relationships among x_i, a_i and b_i show whether the value x is in the interval (a, b). There are three cases which values of a and b can be in: The first case is $a\geq 0$ and $b>0$, the second case is $a<0$ and $b\leq 0$, the third case is $a<0$ and $b>0$. However, there is only the first case for date data.

- **Case 1**: $a\geq 0$ *and* $b>0$.
  1. if $a_1\neq b_1$, then:
     a. if $x_1<a_1$ or $x_1>b_1$, x is not in (a, b).
     b. if $x_1>a_1$ and $x_1<b_1$, x is in (a, b).
     c. if $x_1=a_1$, then it needs to compare x_2 and a_2.
        - if $x_2>a_2$, x is in the interval (a, b).
        - if $x_2<a_2$, x is not in the interval (a, b).
        - if $x_2=a_2$, then it needs to compare x_3 and a_3 in same way as *c*.
     d. if $x_1=b_1$, then need to compare x_2 and b_2.
        - if $x_2>b_2$, x is not in the interval (a, b).
        - if $x_2<b_2$, x is in the interval (a, b).
        - if $x_2=b_2$, then it needs to compare x_3 and b_3 in same way as d.
  2. if $a1=b1$, then:
     - if $x1\neq a1$, x is not in the interval (a, b).
     - if $x1=a1$, then it needs to compare $x2$, $a2$ and $b2$ in accordance with the steps 1 and 2.

In summary, there are three kinds of comparison results: $a_i=b_i$ and $x_i=a_i$, $a_i\neq b_i$ and $x_i=a_i$, $a_i\neq b_i$ and $x_i=b_i$. If the comparison result of a sub-value x_i is one of these three kinds, it needs to compare the next sub-value.

- **Case 2**: $a<0$ and $b\leq 0$.
  1. if a1≠b1, then:
     a. if $x_1<a_1$ or $x_1>b_1$, x is not in (a, b).
     b. if $x_1>a_1$ and $x_1<b_1$, x is in the interval (a, b).
     c. if $x_1=a_1$, then it needs to compare x_2 and a_2.
        - if $x_2>a_2$, for integer number, x is in (a, b); for other data types, x is not in (a, b)
        - if $x_2<a_2$, for integer number, x is not in (a, b); for other data types, x is in (a, b)
        - if $x_2=a_2$, then it needs to compare x_3 and a_3 in same way as *c*.
     d. if $x_1=b_1$, then need to compare x_2 and b_2.
        - if $x_2>b_2$, for integer number, x is not in (a, b); for other types, x is in (a, b).
        - if $x_2<b_2$, for integer number, x is in the interval (a, b); for other types, x is not in the interval.
        - if $x_2=b_2$, then it needs to compare x_3 and b_3 in same way as *d*.
  2. if $a_1=b_1$, then:
     a. if $x_1\neq a_1$, x is not in the interval (a, b).
     b. if $x_1=a_1$, then it needs to compare x_2, a_2 and b_2 in accordance with the steps 1 and 2.
- **Case 3**: $a<0$ and $b>0$.
  1. if $x=0$, x is in the interval (a, b).
  2. if $x>0$.
     a. if $x_1<b_1$, x is in the interval (a, b).
     b. if $x_1>b_1$, x is not in the interval (a, b).
     c. if $x_1=b_1$, then it needs to compare x_2 and b_2 in the same way as 2.
  3. if $x<0$.
     a. if $x_1<a_1$, for integer number x is not in (a, b); for other data types, x is in the interval.
     b. if $x_1>a_1$, for integer number x is in (a, b); for other data types, x is not in the interval.
     c. if $x_1=a_1$, then it needs to compare x_2 and b_2 in same way as a, b and c for $x<0$.

5. Experiments

To assess the features of VParC, experimental studies have been performed. The experiments are conducted on 2.5GHz Pentium Daul-Core, running Windows XP, with 3GB of main memory. For the experiments, a data generator called dbgen is used to generate an instance of the TPC-H data set at scale 1, which yields a total database size of approximately 1GB with 6 tables, named Supplier, Customer, Part, Partsupp, Orders and Lineitem respectively.

We test two types of numeric columns: Integer and float columns. First, we compare the compression data size, the compression time and decompression time of VParC with otherfour compression schemes: RLE, dictionary encoding (DICT for short), FOR, decision tree (DT for short). BITMAP is not presented because if a column t has BM-Regularity, it must has DICT-Regularity (the reason is that $r_{bit}(t)$ is always larger than $r_{dict}(t)$, according to the Equations 7 and 8 presented in section 3.3). Second, we test the performance of search on these two types of columns, which are compressed by using different compression schemes.

In VParC, each column is vertically partitioned into four sub-columns, since integer and float data are stored in four bytes. Then, characteristic values of each sub-column are evaluated, which are gained by scanning sub-columns. The scanning processes are included in the compression. The best algorithm is chosen to compress the sub-column according to the characteristic values. Apparently, sub-columns from the same column may be compressed by using different compression schemes.

5.1. Compression Performance on Integer Data

There are 13 integer columns in the instance of the TPC-H dataset. We use different compression schemes to compress these columns and compare their compression ratio, time for compression and decompression.

5.1.1. Compression Ratio on Integer Columns

We list the size of data compressed by VParC and other four encoding schemes on each integer column in Table 1. NC denotes the original size of a data column without being compressed.

Table 1. Compressed data sizes of integer columns (in KByte).

	NC	RLE	FOR	DICT	DT	VParC
1	40	75	7	7	7	7
2	40	79	18	57	18	11
3	586	1123	92	92	92	92
4	782	1532	147	147	147	147
5	3126	6251	1368	1407	1368	1368
6	3126	6250	1368	1407	1368	1368
7	3126	1563	1758	2540	1563	981
8	5860	11719	3296	3504	3296	3296
9	23443	45212	2198	2198	2198	2198
10	23443	45946	4396	4396	4396	4396
11	23443	46880	10257	10296	10257	10257
12	23443	46885	13187	13968	13187	13187
13	23443	11719	16580	21244	11719	4511

The size of a data column becomes larger than the original if the column is compressed by unsuitable compression algorithm. For example, the size of the first data column without compression is 40KB, but it increases to 75KB after being compressed by RLE. Thus, traditional compression scheme, such as RLE, FOR and DICT cannot be suitable for all kinds of columns.

In DT, the best algorithm is chosen for the column, thus, it achieves better compression results than RLE, FOR and DICT. However, for most columns VParC obtains the similar compression ratio as DT; and for other three columns, the columns 2, 7 and 13, VParC obtains better compression results than DT. Because of the storage structure of integer, the high bytes of the values in a column are the same. Some have Sub-RLE-Regularity although the column itself has no RLE-Regularity. Thus, these sub-columns can be compressed efficiently. For example, the column 2 has 0.45 FOR-Regularity, while two sub-columns in it have 0.0005 RLE-Regularity. Thus, for column 2, VParC is more efficient than DT.

5.1.2. Time for Compression on Integer Columns

Table 2 shows the compression time of each compression schemes. The time for compression is composed of the time to obtain the characteristic values and the time to compress data columns. In DT, most of the time is spent on the obtaining of the characteristic values. In VParC, although there are four sub-columns need to be processed, the required time is much less than DT due to the byte nature of sub-columns. Thus, the VParC cost less time to compress than DT.

Table 2. Time for compression on integer columns (in ms).

	RLE	FOR	DICT	DT	VParC	
1	0.2	4.3	4.9	5.6	1.8	**1**
2	0.2	12.3	13.8	613.5	2.3	**2**
3	3.1	66.1	74.4	82.4	28.9	**3**
4	4.1	103.9	121.7	146.6	37.4	**4**
5	17.9	998.8	1162.8	65447.4	153.1	**5**
6	15.1	987.5	1234.4	34083.2	156.7	**6**
7	10.4	1215.1	1435.1	8087.5	752.1	**7**
8	33.2	2390.2	2837.4	127285.1	297.4	**8**
9	121.5	1474.8	1812.6	1836.4	1157.9	**9**
10	125.5	3207.5	3745.4	4296.5	1101.2	**10**
11	121.1	7487.7	9172.3	255229.3	1096.4	**11**
12	118.6	9527.8	11920.1	535771.9	1070.5	**12**
13	96.7	11200.6	11844.8	61544.1	1746.6	**13**

5.1.3. Time for Decompression on Integer Columns

The time needed for decompression is shown in Table 3. RLE needs less time for decompression than FOR and DICT due to the bit shift in the FOR and DICT algorithms.

Table 3. Time for decompression on integer columns (in ms).

	RLE	FOR	DICT	DT	VParC
1	0.3	1.4	1.4	1.4	0.7
2	0.3	3.6	3.1	3.1	0.6
3	3.9	21.5	21.5	21.3	12.6
4	5.2	33.5	33.2	32.7	15.7
5	21.3	277.1	285.9	279.1	66.5
6	21.2	285.7	292.2	280.8	65.7
7	12.3	308.4	311.2	201.1	58.0
8	39.6	680.6	657.5	663.7	122.5
9	164.6	541.2	529.8	509.5	519.9
10	152.4	1061.7	1053.7	1025.6	456.3
11	155.3	2188.6	2208.8	2122.1	461.4
12	155.9	2719.2	2784.2	2687.2	463.9
13	109.4	2840.2	2643.9	171.9	468.9

In VParC, decompression is faster than DT except columns 9 and 13, because in these two columns, there is more than one sub-column compressed using FOR or DICT, which are more slower than RLE. For example, the column 13 has RLE-Regularity while there are two sub-columns have Sub-RLE-Regularity and other two sub-columns have Sub-DICT-Regularity. Therefore, in the columns with Sub-DICT-Regularity or Sub-DICT-Regularity, decompression time for VParC is more than DT.

5.2. Compression Performance on Float Data

There are 8 float columns in the instance of the TPC-H data in all Similar to the experiments for integer columns, we compared the compression ratio, compression time and decompression time of each algorithm on each float columns. Notice that, FOR is not suitable for float columns, thus, it is not tested in this experiment.

5.2.1. Compression Ratio on Float Columns

The sizes of the data columns compressed using each algorithm on each float columns are illustrated in Figure 5. It can be seen that VParC leads to more size for the columns 4, 6, 7 and 8 than DT. Because of the storage structure of float data, the column without RLE-Regularity, it will not have Sub-RLE-Regularity. For example, the column 6 has 0.125 DICT-Regularity, while each sub-column in column 6 has 0.375 DICT-Regularity. Thus, VParC is less efficient than DT for the columns with better global-regularity.

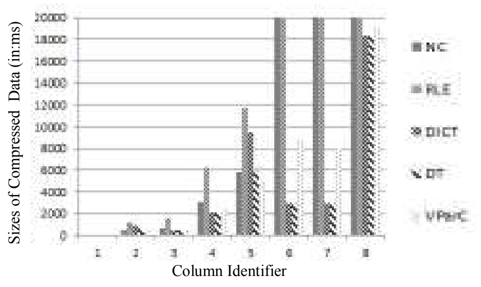

Figure 5. Data sizes of compressed float columns.

However, VParC shows better compression ratio for columns 1, 2, 3 and 5, because they are not much regular columns. Especially for columns 1, 2 and 5, they are irregular and cannot be compressed by any lightweight schemes. In general, although VParC has no advantage for float columns with good Global-Regularity, it shows better compression ratio for irregular columns.

5.2.2. Time for Compression on Float Columns

The time required for each compression scheme is shown in Figure 6. Just like integer columns, VParC requires less time to compress than DT because obtaining characteristic values is time-consuming for DT.

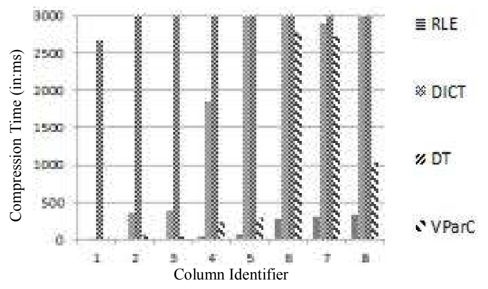

Figure 6. Time for compression on float columns.

5.2.3. Time for Decompression on Float Columns

Figure 7 illustrates the decompression time for each algorithms. Because columns 1, 2 and 5are irregular columns, they cannot be compressed by any lightweight algorithms except VParC. Thus, VParC cost more time to decompress for these columns than DT.

For columns 6 and 7, each sub-column in these two columns has DICT-Regularity. DICT and FOR is more time-consuming than RLE, which leads to more time for VParC to decompress than DT.

However, for the columns that are not much regular, such as the columns 3, 4 and 8, VParC shows better decompression performance than DT.

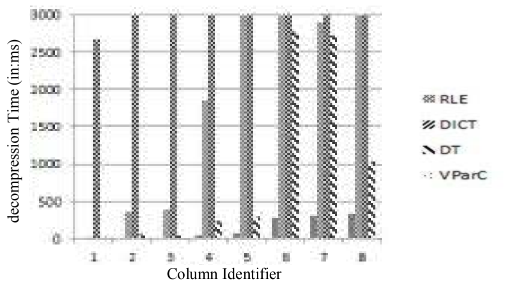

Figure 7. Time for decompression on float columns.

In conclusion, we compare the compression ratio, compression time and decompression time of four compression schemes for numeric data columns. Obviously, RLE, DICT and FOR are not suitable for all columns. Thus, our experiments focus on the comparison between DT and VParC.

The results of our experiments illustrated that VParC has advantages in the compression speed and VParC is not always superior to DT in the aspects of decompression speed and compression ratio. However, VParC is more universal than DT. That is, for columns with Global-Regularity that can be compressed by DT, they can be compressed by VParC; for irregular columns that cannot be compressed by DT, they also can be compressed by VParC.

5.3. Time Required for Equivalent Searching

In equivalent search, the search results needn't to be decompressed. Thus, the time for equivalent search includes the time of loading compressed data from disk to memory and the time of searching on compressed data. In our experiment, random values are used as searching value and the time of searching is an average time of 100 times. Notice than, RLE, FOR and DICT are not suitable for all columns, we only evaluate the time for equivalent searching on columns compressed by using DT and VParC.

5.3.1. Equivalent Search on Integer Columns

Figure 8 shows that searching on data compressed by VParC needs less time than DT, there are two reasons: VParC is more efficient for integer column compression, which is shown in Table 1. Thus, it costs less time for loading data compressed by VParC than by DT. In VParC, for the values that are not equal to the searching value, there is no need to compare all of the sub-values. Thus, the time of searching on compressed data is less than in DT.

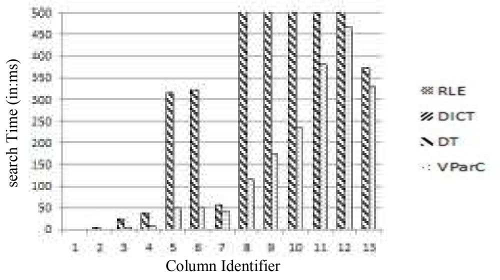

Figure 8. Time for equivalent search on integer column.

5.3.2. Equivalent Search on Float Columns

In Figure 9, the time of equivalent search on float columns for VParC and DT are compared. Just like the equivalent searching on integer columns, searching on data compressed by VParC cost less time than compressed by DT.

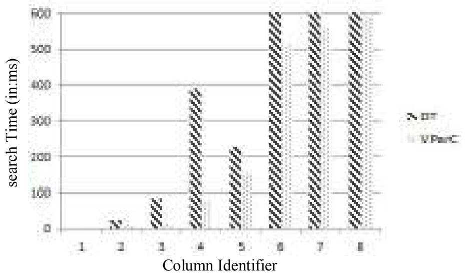

Figure 9. Time for equivalent search on float columns.

5.4. Time Required for Range Searching

For range search, the search time is composed of the time of loading, the time of searching and the time of decompression. Random range is used as searching range and the time of searching is an average time of 100 times.

5.4.1. Range Search on Integer Columns

Figure 10 shows the time for range search on integer columns. Because of fewer disks I/Os and less decompression time, range search on VParC is quick than DT for most integer columns except column 13. Because VParC need more decompression time for this column than DT, which is more than the time saving from compression.

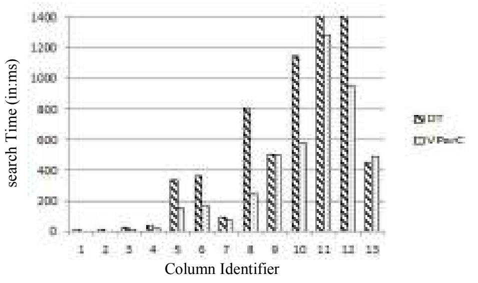

Figure 10. Time for range search on integer column.

5.4.2. Range Search on Float Columns

The time for range search on float columns is showed in Figure 11. For the columns 6 and 7, although the size of data compressed by VParC is larger than DT, VParC requires more or less the same time as DT for range search on this two columns due to less time of decompression. For the columns 1, 2 and 5, VParC needs a little more time than DT, because the time saved by less I/O is more than the time of searching and search caused by compression. However, for columns 3, 4 and 8, VParC required less time than DT, because of the less size of compressed data and less time for decompression.

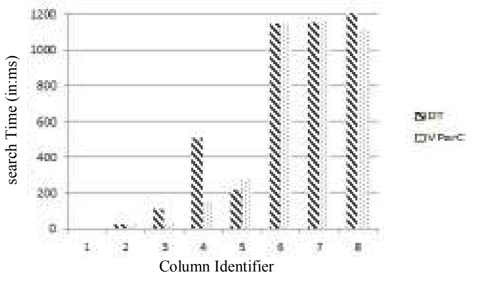

Figure 11. The time of range search for float columns.

We just list the experiment results of integer and float data columns. From the results, we conclude that for much regular data column, VParC is not better than DT; However, VParC is better than DT for not much regular columns and irregular columns. Date and decimal data are not listed because they show the similar results to the integer and float data.

6. Conclusions and Future Work

Compression is one of the most important techniques for column-oriented data management. In this paper, we prove that a column is more possible to have sub-regularity than to have global-regularity. Thus, a new compression scheme, called VParC, is proposed and related issues are examined. Furthermore, we implemented it and evaluation studies have been performed.

In VParC, a data column is vertically partitioned into several sub-columns; each sub-column is compressed by suitable compression algorithms according to its regularity. Only lightweight compression algorithms are used to compress sub-columns. In this way, equivalent search and range search can be implemented directly on compressed data columns. A searching method to operate on compressed sub-columns directly is designed to directly operate the data compressed by VParC without decompression in advance. The experiment results illustrated that the VParC and the corresponding searching method are very promising.

As a part of our future work, we plan to integrate more compression scheme in VParC for sub-columns and leads to improved compression ratio. Also, parallel processing and indexes on sub-columns are considered to improve searching performance.

References

[1] Abadi J., Madden R., and Ferreira M., "Integrating Compression and Execution in Column Oriented Database Systems," *in Proceedings of SIGMOD International Conference on Management of Data*, Chicago, USA, pp. 671-682, 2006.

[2] Abadi J., Madden R., and Hachem N., "Column-Stores vs. Row-Stores: How Different are They Really," *in Proceedings SIGMOD International Conference on Management of Data*, Vancouver, Canada, pp. 967-980, 2008.

[3] Abadi J., "Query Execution in Column-Oriented Database Systems," *MIT PhD thesis*, 2008.

[4] Akman I., Bayindir H., Ozleme S., Akin Z., and Misra S., "Lossless Text Compression Technique using Syllable Based Morphology," *the International Arab Journal of Information Technology*, vol. 8, no. 1, pp. 66-74, 2011.

[5] Binnig C., Hildenbrand S., and Faerber F., "Dictionary-based Order-Preserving String Compression for Main Memory Column-Stores," *in Proceedings of the SIGMOD International Conference on Management of Data*, pp. 283-296, 2009.

[6] Chen Y., Johannes G., and Flip K., "Query Optimization in Compressed Database Systems," *in Proceedings of the SIGMOD International Conference on Management of Data*, California, pp.271-282, 2001.

[7] Graefe G., "Efficient Columnar Storage in B-trees," *ACM SIGMOD Record*, vol. 36, no. 1, pp. 3-6, 2007.

[8] Harizopoulos S., Liang L., Abadi J., and Madden S., "Performance Tradeoffs in Read-Optimized Databases," *in Proceedings of the 32nd International Conference on Very Large Data Bases*, Seoul, pp. 487-498, 2006.

[9] Holloway L. and DeWitt J., "Read-Optimized Databases: In Depth," *VLDB Endowment*, vol. 1, no. 1, pp. 502-513, 2008.

[10] Hsiao-Ping T., De-Nian Y., and Ming-Syan C., "Exploring Application-Level Semantics for Data Compression," *IEEE Transaction Knowledge and Data Engineering*, vol. 23, no. 1, pp. 95-109, 2011.

[11] Larson A., Clinciu C., Hanson N., Oks A., Price L., Rangarajan S., Surna A., and Zhou Q., "SQL Server Column Store Indexes," *in Proceedings of the 31st SIGMOD*, pp. 1177-1184, 2011.

[12] Lemke C., Sattler U., Faerber F., and Zeier A., "Speeding up queries in Column Stores: A Case for Compression," *Data Warehousing and Knowledge Discovery of Lecture Notes in Computer Science*, pp. 117-129, 2010.

[13] Ningde X., Guiqiang D., and Tong Z., "Using lossless Data Compression in Data Storage Systems: Not for Saving Space," *IEEE Transaction Computers*, vol. 60, no. 3, pp. 335-345, 2011.

[14] O'Neil P. and Quass D., "Improved Query Performance with Variant Indexes," *in Proceedings of SIGMOD International Conference on Management of Data*, Tucson, USA, pp. 38-49, 1997.

[15] Roth A. and VanHorn J., "Database Compression," *ACM SIGMOD Record*, vol. 22, no. 3, pp. 31-39, 1993.

[16] Shah D. and Vithlani C., "VLSI-Oriented Lossyimage Compression Approach using DA-Based 2D-Discrete Wavelet," *the International Arab Journal of Information Technology*, vol. 11, no. 1, pp. 59-68, 2014

[17] Tanaka H. and Garcia L., "Efficient Run-Length Encodings," *IEEE Transaction Information Theory*, vol. 6, no. 28, pp. 880-890, 1982.

[18] Tsirogiannis D., Harizopoulos S., Shah A., Wiener L., and Graefe G., "Query Processing Techniques for Solid State Drives," *in Proceedings of SIGMOD International Conference on Management of Data*, Rhode Island, pp. 59-72, 2009.

[19] Ziv J. and Lempel A., "Compression of Individual Sequences via Variable-Rate Coding," *IEEE Transaction Information Theory*, vol. 24, no. 5, pp. 530-536, 1978.

[20] Zukowski M., Héman S., Nes N. and Boncz A., "Super-Scalar RAM-CPU Cache Compression," *in Proceedings of the 22nd International Conference on Data Engineering*, pp. 59, 2006.

Ke Yan received her BE and MS degrees from HuNan University in China in 2003 and 2006 respectively. Currently, she is a PhD candidate in School of Computer Science and Technology, Huazhong University of Science and Technology in China. Her research topic includes cloud data management and column-oriented database.

Hong Zhu received her BE, MS and PhD degrees from Huazhong University of Science and Technology in 1987, 1990 and 2001 respectively. Currently, she is a professor and a PhD supervisor in School of Computer Science and Technology, Huazhong University of Science and Technology in China. Her main research areas are database and security.

Kevin Lu is a lecturer in Brunel University in United Kingdom. He supervised six PhD student projects to completion. He obtained grants from EU, KTP program, ORS and other organizations. His main research areas are information management and business analytics.