Compression Schemes with Data Reordering for Ordered Data

Chun-Hee Lee, Department of Computer Science, KAIST (Korea Advanced Institute of Science and Technology), Daejeon, Korea

Chin-Wan Chung, Department of Computer Science, KAIST (Korea Advanced Institute of Science and Technology), Daejeon, Korea

ABSTRACT

Although there have been many compression schemes for reducing data effectively, most schemes do not consider the reordering of data. In the case of unordered data, if the users change the data order in a given data set, the compression ratio may be improved compared to the original compression before reordering data. However, in the case of ordered data, the users need a mapping table that maps the original position to the changed position in order to recover the original order. Therefore, reordering ordered data may be disadvantageous in terms of space. In this paper, the authors consider two compression schemes, run-length encoding and bucketing scheme as bases for showing the impact of data reordering in compression schemes. Also, the authors propose various optimization techniques related to data reordering. Finally, the authors show that the compression schemes with data reordering are better than the original compression schemes in terms of the compression ratio.

Keywords: *Bucketing Scheme, Compression Scheme, Data Reordering, Ordered Data, Run-Length Encoding*

INTRODUCTION

Currently, a large volume of data in various environments is generated. Such a large volume of data consumes valuable resources such as space, network bandwidth, and CPU. In order to save the resources, data compression schemes have been developed and applied in many applications.

However, they do not consider the effect of data reordering. If we reorganize data, the compression ratio for the reorganized data may be improved compared to that for the original data. Some papers deal with data reordering problems in very limited environments (Apaydin, Tosun & Ferhatosmanoglu, 2008; Blandford & Blelloch, 2002; Johnson, Krishnan, & Chhugani, 2004; Pinar, Tao & Ferhatosmanoglu, 2005). The work of Apaydin et al. (2008), Blandford and Blelloch (2002), Johnson et al. (2004), and Pinar et al (2005) assumes that the order of data does not have to be preserved, that is, the input data is unordered data. However, in general, the order of data should be preserved and has the important information. For example, time series data should be ordered by the time. If we change the order of the time series data, it will lose much information. Therefore, the approaches in Apaydin et al. (2008), Blandford and Blelloch (2002), Johnson et al. (2004), and

DOI: 10.4018/jdm.2014010101

Table 1. Notational convention

Notation	Meaning
RLE(D)	Compressed data for *D* using the run-length encoding
BUCKET(D, ε)	Compressed data for *D* using the bucketing scheme with an error bound *ε*
d_i	the *i*-th element in data set *D*
$d_{i,j}$	$\{d_i, d_{i+1}, \cdots, d_{j-1}, d_j\}$, where $i \le j$
<X>	token *X* in the run-length encoding or the bucketing scheme
≪X≫	movement information *X*

Pinar et al. (2005) cannot be applied to the ordered data such as time series data (Chen, Dong, Han, Wah, & Wan, 2002; Korn, Jagadish, & Faloutsos, 1997; Reeves, Liu, Nath, & Zhao, 2009; Elmeleegy, Elmagarmid, Cecchet, Aref, & Zwaenepoel, 2009).

Consider the run-length encoding with the ordered data D = {1, 1, 1, 3, 3, 1, 1, 4, 4, 4}. The run-length encoding is one of the widely used lossless compression schemes. It replaces repeated values with <value, count>, where count is the number of repeated values. We can represent D by the run-length encoding as follows. Note that RLE(D) is the compressed data for D using the run-length encoding. See Table 1 for the detailed notational convention.

RLE(D) = {<1,3>,<3,2>,<1,2>,4,3>},

where in the pair <a,b>, a is value and b is count.

We can reduce the number of elements in RLE(D) by reordering elements in D. Consider D′ = {1, 1, 1, 1, 1, 3, 3, 4, 4, 4} which is the data after reordering elements in D. Then, RLE(D′) = {<1,5>,<3,2>,<4,3>}. Since |RLE(D′)| = 3 is less than |RLE(D)| = 4, we can improve the compression ratio by reorganizing data if the data is unordered data. However, if the data is ordered data, we should keep the following mapping table (presented in Box 1) to reconstruct the original data from RLE(D′),

The space benefit by data reordering may be less than the space overhead for storing the mapping table. That is, the compression ratio by the compression scheme with data reordering may be worse than that by the original compression scheme without data reordering. In this case, data reordering is useless. Therefore, we should carefully consider how to store the mapping table effectively in order to apply data reordering techniques for ordered data.

In this paper, we first investigate general principles to improve compression schemes by data reordering. We do not keep the total mapping table since the size of the mapping table is too big. Instead, we keep the movement information for the portion of a whole data set. The movement information is represented by ≪start, end, newStart≫. It means that the

Box 1.

```
Original position:  1   2   3   4   5   6   7   8   9  10
                    ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓
Changed position:   1   2   3   6   7   4   5   8   9  10
```

portion of data between start position and end position is moved to newStart position. In the above ordered data D, the movement information is represented by ≪6,7,4≫ (i.e., <start position, end position, new start position>) instead of the total mapping table. And, we reorganize data when the space for storing the list of movement information is less than the space benefit by data reordering. Based on the principles, we consider two compression schemes, run-length encoding (Salomon, 2004; Wikipedia, 2012; Abadi, Madden, & Ferreira, 2006) and bucketing scheme (Buragohain, Shrivastava, & Suri, 2007; Gandhi, Nath, Suri, & Lie, 2009), which have similar properties. The compressed data form in the bucketing scheme is the same as that in the run-length encoding.

Although storing the list of the movement information ≪start, end, newStart≫ needs less space than storing the total mapping table, it still needs much space. Therefore, we propose a method to represent the movement information compactly in the run-length encoding and the bucketing scheme. Based on the compact movement representation, we devise an effective run-length encoding and bucketing scheme with data reordering. Also, by analyzing the characteristics of the run-length encoding and the bucketing scheme, we propose various optimization techniques related to data reordering.

Our contributions are as follows:

- **General principles to improve a compression scheme by data reordering** Although previous compression schemes can reduce a large volume of data effectively, there are still new approaches to reduce data further. If we reorganize data, the compression ratio may be improved. Based on this observation, we establish general principles to improve the compression ratio for ordered data by data reordering.
- **Run-length encoding with data reordering** We propose run-length encoding and decoding algorithms with data reordering. While data compressed by the original run-length encoding is composed of the list of <value, count>, that by the run-length encoding with data reordering has additionally a list of compact movement information.
- **Bucketing scheme with data reordering** We propose encoding and decoding algorithms for the bucketing scheme with data reordering which are similar to run-length encoding and decoding algorithms with data reordering.
- **Various optimizations for data reordering in the run-length encoding and the bucketing scheme** Although the run-length encoding and bucketing scheme with data reordering can improve the compression ratio compared to the original compression schemes, we can reduce data more effectively by two optimization techniques, Movement Dropping and Neighbor Merging.
- **Experimentation to evaluate our proposed approaches** Through an experimental study, we show that the compression ratio of the compression schemes with data reordering is considerably improved compared to that of the original compression schemes.

Related Work

Data compression schemes have been developed in a wide range of areas to reduce the amount of space, transmission bandwidth, data mining time, or query processing time. In the case of multimedia data, time series data and graph data, the volume of data is too huge and data has similar patterns. Therefore, we can reduce the volume of data using compression schemes (DeVore, Jawerth, & Lucier, 1992; Elmeleegy, Elmagarmid, Cecchet, Aref, & Zwaenepoel, 2009; Pennebaker & Mitchell, 1993; Rao & Yip, 1990; Reeves, Liu, Nath, & Zhao, 2009). In networking systems, to reduce the transmission bandwidth, various compression schemes have been used (Chen, Li, & Mohapatra, 2004; Degermark, Engan, Nordgren, & Pink, 1996; Deligiannakis, Kotidis, & Roussopoulos, 2004; Deligiannakis, Kotidis, & Roussopoulos, 2007). In the data mining area, the compression or synopsis techniques have been applied to improve the efficiency of mining (Zhang, Liu, Ling, Bruckner, & Tjoa, 2006; Hai, Lenco, Poncelet, & Teisseire, 2013). Also, in the database literature, in order to improve the execution time of query processing, compression schemes have been applied (Abadi, Madden, & Ferreira, 2006; Graefe & Shapiro, 1991; Goldstein, Ramakrishnan, & Shaft, 1998; Iyer & Wilhite, 1994; Johnson, 1999; Ray, Haritsa, & Seshadri, 1995). Although there are many effective compression schemes as data reduction tools, they may have a further chance to reduce data by data reordering techniques.

Some papers deal with data ordering problems in compression schemes. Blandford and Blelloch (2002) consider document reordering problems in the inverted index. Since there are too many documents in the web environment, the size of the inverted index is very large. Therefore, we can apply the difference coding to the list of document numbers in order to reduce the size of the index. If we permute document numbers considering the difference coding, we can improve the compression ratio of the difference coding. Therefore, Blandford et al. use a hierarchical clustering technique for numbering documents effectively. They first construct a similarity graph for documents using consine measures and apply the hierarchical clustering to the graph. Then, they order clusters hierarchically.

Ouyang et al. (2002) propose a framework for compressing a collection of files based on the delta compression. Since the delta compression is applied to a collection of files, a sequence for performing the delta-compression affects the compression ratio. They transform the problem of finding an optimal sequence to that of finding the maximum branching in the graph theory. However, it needs a large amount of the computation time to construct a graph and find the maximum branching. Therefore, to reduce the computation time, Ouyang et al. use the techniques in document clustering.

Johnson et al. (2004) deal with the boolean matrix reordering problem. In many environments, large boolean matrices are generated. To store a large boolean matrix effectively, we can use the run-length encoding. If we reorganize columns in the matrix, we can reduce the number of runs. An example (Johnson, Krishnan, & Chhugani, 2004) for reordering the matrix is shown in Figure 1.

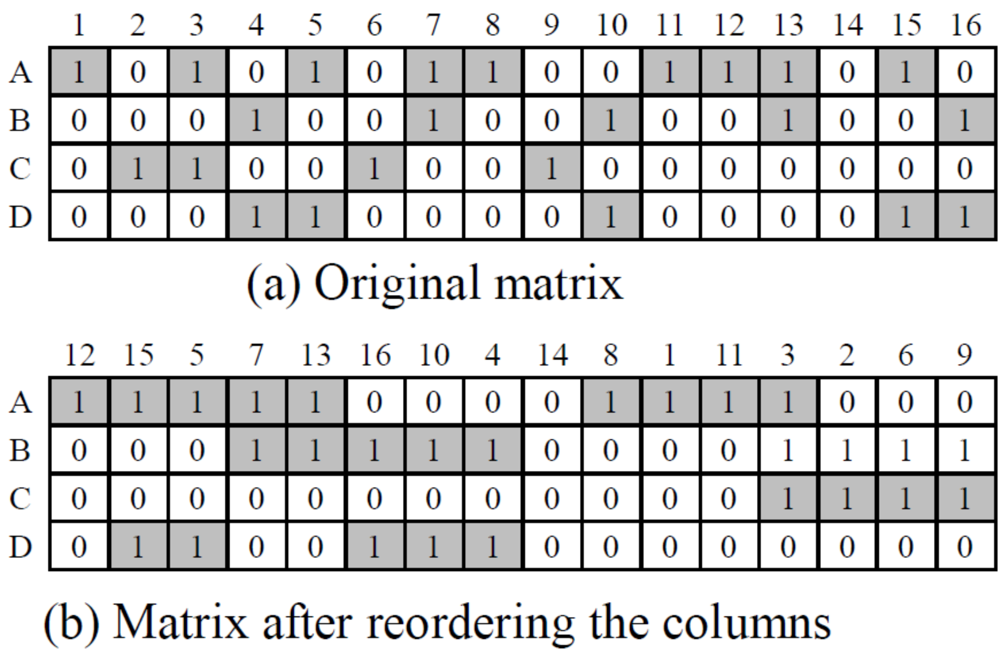
If we reorder columns like Figure 1 (b), we can reduce the number of runs compared to that of the original matrix in Figure 1 (a). However, the matrix reordering problem is NP-hard. Therefore, Johnson et al. (2004) transform the matrix reordering problem to the traveling salesman problem in the Hamming space. To solve the large traveling salesman problem, they propose instancepartitioning and sampling as heuristics.

Pinar et al. (2005) solve the problem similar to the work of John et al. (2004). They consider the tuple reordering problem in the bitmap table which is used for various scientific applications. They propose gray code ordering as heuristics for tuple reordering. The gray code ordering arranges numbers with respect to the gray code in which adjacent numbers differ at only one-bit. In addition, Apaydin and Tosun (2008) theoretically analyze two data reordering techniques, lexicographical ordering and gray code ordering, in the context of bitmap indexes.

However, the approaches in Apaydin and Tosun (2008), Blandford and Blelloch (2002), Johnson et al. (2004), and Pinar et al. (2005) assume the environment that the input data is unordered. Thus, we do not need to store a mapping information in such an environment. Therefore, we cannot apply them to ordered data. In this paper, we consider compression schemes with data reordering for ordered data.

As a string transformation technique for compression, Burrows-Wheeler Transform (BWT) (Burrows & Wheeler, 1994) is a well-known compression scheme. BWT transforms a string into the string which is easy to compress. The transformed string by BWT may contain many repeated alphabets. Therefore, the transformed string is more compressible than the original string. A string transformed by BWT should be compressed by another compression scheme. Therefore, our approach

Figure 1. Example for the boolean matrix reordering (Johnson et al., 2004)



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	1	0	1	0	1	0	1	1	0	0	1	1	1	0	1	0
B	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1
C	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0
D	0	0	0	1	1	0	0	0	0	1	0	0	0	0	1	1

(a) Original matrix

	12	15	5	7	13	16	10	4	14	8	1	11	3	2	6	9
A	1	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0
B	0	0	0	1	1	1	1	1	0	0	0	0	1	1	1	1
C	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
D	0	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0

(b) Matrix after reordering the columns

is orthogonal to BWT. We can use our approach after applying BWT.

Preliminaries

In this section, we explain the run-length encoding (Abadi, Madden, & Ferreira, 2006; Salomon, 2004; Wikipedia, 2012) and the bucketing scheme (Buragohain, Shrivastava, & Suri, 2007; Gandhi, Nath, Suri, & Lie, 2009) as preliminaries. The notational convention used in this paper is shown in Table 1.

Run-Length Encoding

The run-length encoding as one of lossless compressions replaces repeated values with <value, count>, where count is the number of repeated values. In the case of data sets that have many repeated values, the run-length encoding can reduce space significantly. Although there are run-length encoding algorithms in many versions, we use the run-length encoding algorithm in Figure 2.

In this paper, we assume that the size of value and the size of count are fixed. Thus, in Statement 7, we add the condition count == $2^{\text{RUN LENGTH SIZE}}$ - 1 since if the number of repeated values is more than the possible maximum number for count, the allocated space for count is exceeded. Since the algorithm in Figure 2 is trivial, we skip the detailed explanation.

Example 1: Let D = {1, 1, 1, 1, 1, 2, 2, 3, 3, 1, 1, 1} and RUN_LENGTH_SIZE = 2. By the algorithm of Figure 2, D is encoded into RLE(D) = {<1, 3>,<1, 2>,<2, 2>,<3, 2>,<1, 3>}.

Bucketing Scheme

The bucketing scheme (Buragohain, Shrivastava, & Suri, 2007; Gandhi, Nath, Suri, & Lie, 2009) as one of lossy compressions is based on piece-wise constant approximations. It represents data by piece-wise constant functions. In the bucketing scheme, we make a bucket for a portion of data D = {d_1, d_2, …, d_n} which has

Figure 2. Run-length encoding algorithm

```
Function Run_Length_Encoding(ordered data d[ ])
begin
 1: // The index starts from 1
 2: val := d[1];
 3: count := 1;

 4: for i:=2; i≤ the size of d[ ]; i++
 5: {
 6:  // RUN_LENGTH_SIZE is the size of the variable count (e.g., the number of bits)
 7:    if (d[i] ≠ val) or (count == 2^RUN_LENGTH_SIZE -1)
 8:    {
 9:        make token <val, count> and append it to the result list;
10:        val := d[i];
11:        count := 1;
12:    }
13:    else
14:        count++;
15: }
16: make token <val, count> and append it to the result list
end
```

the minimum value (minVal), the maximum value (maxVal) and the length (count) of the bucket. We add data d_i to one bucket if the difference between minVal and maxVal in the bucket is less than or equal to 2ε, where ε is the given error bound. If the difference is more than 2ε, we make a new bucket. We approximate elements in the bucket by the average value (=(minVal+maxVal)/2). Therefore, we can guarantee $\max_i |d_i - \hat{d}_{ij}| <= \varepsilon$, where $\hat{d}_i$ is the approximate value for d_i.

The algorithm for the bucketing scheme is shown in Figure 3. In Statement 1-4, we initialize the bucket. Then, in Statement 9-16, if the difference between the minimum value and the maximum value is more than 2ε, we make the token with the bucket by computing the average value, append the token to the result list and create a new bucket. Note that in the bucketing scheme, the form of compressed data is the same as that of the run-length encoding. Otherwise, we add the data to the bucket and update the variables of the bucket in Statement 17-22. Finally, we make the token for the remaining elements in Statement 24. Like the algorithm of Figure 2, we add the condition count == $2^{\text{BUCKET LENGTH SIZE}}$ - 1 in Statement 10.

Example 2: Let D = {1, 1.2, 1.4, 1.3, 1.1, 2.4, 2.6, 4}, BUCKET LENGTH SIZE = 2 and ε = 0.5. By the algorithm of Figure 3, D is encoded into BUCKET(D, 0.5) = {<1.2,3>,<1.2,2>,<2.5,2>,<4, 1>}.

Figure 3. Bucketing scheme algorithm

```
Function Bucketing_Scheme(ordered data d[ ], error bound ε)
begin
 1: // The index starts from 1
 2: minVal : = d[1];
 3: maxVal := d[1];
 4: count := 1;
 5: for i:=1; i≤the size of d[ ]; i++
 6: {
 7:     newMinVal := min(minVal, d[i]);
 8:     newMaxVal := max(maxVal, d[i]);
 9:    // BUCKET_LENGTH_SIZE is the size of the variable count
10:     if (newMaxVal – newMinVal  >  2ε)  or
           (count == 2^BUCKET_LENGTH_SIZE -1)
11:     {
12:         make token < (minVal+maxVal)/2,  count>
                for the bucket and append it to the result list;
13:         minVal := d[i];
14:         maxVal := d[i];
15:         count := 1;
16:     }
17:     else
18:     {
19:         minVal := newMinVal;
20:         maxVal := newMaxVal;
21:         count++;
22:     }
23: }
24: make token < (minVal+maxVal)/2,  count>
        for the bucket and append it to the result list;
end
```

Motivation

In the case of unordered data, we can generally improve compression schemes by reorganizing data. However, reordering ordered data may be disadvantageous. This is because we need much space to store a mapping table that maps the original position to the changed position. In the case of ordered data, if we can devise a method to store or encode the mapping table effectively so that the space benefit by data reordering is more than the space overhead for storing the mapping table, it is beneficial to reorganize data in order to reduce the amount of data further.

Motivated by this point, we consider compression schemes with data reordering for ordered data. Since we need much space to keep the mapping table, we keep a list of movement information instead of the mapping table. The movement information consists of ≪start, end, newStart≫, where start and end are the start and end position of data to be moved, and newStart is the position to be inserted. Figure 4 shows how the data movement is represented by ≪start, end, newStart≫. If we use the movement information to reorganize data, we do not have to keep the mapping table with a big size.

Based on the movement information, we make general principles to improve compression schemes by data reordering as follows:

- We do not keep a mapping table. Instead, we store a list of movement information with the form ≪start, end, newStart≫. Therefore, we reduce the space cost for reorganizing data.
- The space overhead for storing a list of movement information should be less than the space benefit generated from data reordering. If the space benefit is small compared to the space overhead for the list of movement information, we should not reorganize data.

As bases for showing the impact of data reordering, we choose two compression schemes, run-length encoding and bucketing scheme, since we can exploit data reordering techniques well in the two compression schemes. Through the proposed run-length encoding and bucketing scheme with data reordering, we show the possibility to improve compression schemes by data reordering.

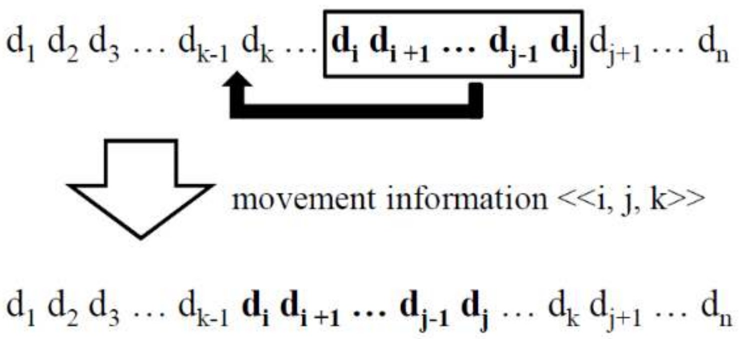
Run-Length Encoding With Data Reordering

In this section, we consider the run-length encoding with data reordering. By merging the run-length encoding and data reordering, we propose a more effective run-length encoding with data reordering.

According to the general principles, we move the portion of data in order to improve the compression ratio of the run-length encoding. For the sake of explanation, we explain our technique with the ordered data D = {1, 1, 2, 2, 2, 3, 3, 2, 2, 4, 4, 5, 5, 2, 5, 5, 5}. In the data D, we can move $d_{8,9}$ = {2, 2} to the position between d_5 and d_6. Now, we should check if we can reduce the space through such a movement. In this case, we can decrease the space for one token corresponding to $d_{8,9}$ = {2, 2} since $d_{8,9}$ = {2, 2} is integrated with $d_{3,5}$ = {2, 2, 2}, and {d_3, d_4, d_5, d_8, d_9} composes one token <2,5>. On the other hand, we need the space for the movement information ≪8,9,6≫.

Therefore, if the size of the movement information is less than the size of one token in the run-length encoding, we can improve the run-length encoding by reorganizing data. For example, if the size of the value is 8 bytes, the

Figure 4. Movement information



$d_1\ d_2\ d_3 \ldots d_{k-1}\ d_k \ldots \mathbf{d_i\ d_{i+1} \ldots d_{j-1}\ d_j}\ d_{j+1} \ldots d_n$
movement information <<i, j, k>>
$d_1\ d_2\ d_3 \ldots d_{k-1}\ \mathbf{d_i\ d_{i+1} \ldots d_{j-1}\ d_j} \ldots d_k\ d_{j+1} \ldots d_n$

size of the run length is 2 bytes and the size of the position is 2 bytes, we can get the space benefit 4 bytes = (8+2)-2*3. However, if the size of the value is not much bigger than the size of the position, we cannot get the space benefit by reorganizing data.

By analyzing the characteristics of the run-length encoding, we propose an effective representation for the movement information in the run-length encoding. Since in the run-length encoding, continually repeated values are represented by only one token, we move data by the unit of one token. Therefore, we describe data reordering in the encoded data using the original run-length encoding.

In the run-length encoding, one token is composed of<value, count>. If we utilize count (i.e., run length) and impose some constraints for the movement, we can represent the movement information very effectively by ≪start, newStart≫. Consider the following example.

D = {1, 1, 2, 2, 2, 2, 3, 3, 2, 2, 4, 4, 5, 5, 2, 5, 5, 5}
R = RLE(D) = {<1,2>,<2,3>,<3,2>,<2,2>,<4,2>,<5,2>,<2,1>,<5,3>}

We can move token r_4=<2,2> in RLE(D) to the position after r_2 =<2,3> with the compact movement representation ≪8,6≫. Then, the above data D is transformed as follows:

D′ = {1, 1, 2, 2, 2, 2,2, 3, 3, 4, 4, 5, 5, 2, 5, 5, 5}
R′ = RLE(D′) = {<1,2>,<2,5>,<3,2>,<4,2>,<5,2>,<2,1>,<5,3>}

The problem is that how we can decode R′ = RLE(D′) with the compact movement representation ≪8,6≫. In the movement representation ≪8,6≫, the new start position is 6. By accumulating count of the token in R′ sequentially, we can know that the new start position newStart is contained in the token r'_2=<2,5> since start position for r_2 < newStart position < end position for r_2 (3 < 6 < 7). If we assume that token r_i=<v_i, c_i> is moved and integrated with r_j=<v_j, c_j> such that 1) $v_i = v_j$, 2) r_j is a preceding token (j < i), and 3) r_i is integrated with r_j at the back (i.e., newStart = the end position of r_j +1) as shown in Figure 5, we can know that r'_2 was made by merging two tokens in R. Therefore, we can separate r'_2 into original two tokens, <2,3> and <2,2> by cutting off r'_2 at the position between newStart-1 and newStart.

Lemma 1. Let R = {<v_1, c_1>, <v_2, c_2>, · · ·, <v_n, c_n>} be the encoded data by the original run-length encoding. If $v_i = v_j (j < i)$, we can move token <v_i, c_i> to the position after token <v_j,c_j>. Let RLE(D′) = {<v'_1, c'_1>,<v'_2, c'_2>, · · ·, <v'_{n-1}, c'_{n-1}>} be the data after <v_i, c_i> is merged with <v_j, c_j>. Let start = $\sum_{a=1}^{i-1} c_a + 1$ and newStart= $\sum_{a=1}^{j} c_a + 1$. Then,

$$c_i = \sum_{a=1}^{k} c'_a - newStart + 1 \text{ and}$$

$$c_j = c'_j - c_i \text{, if}$$

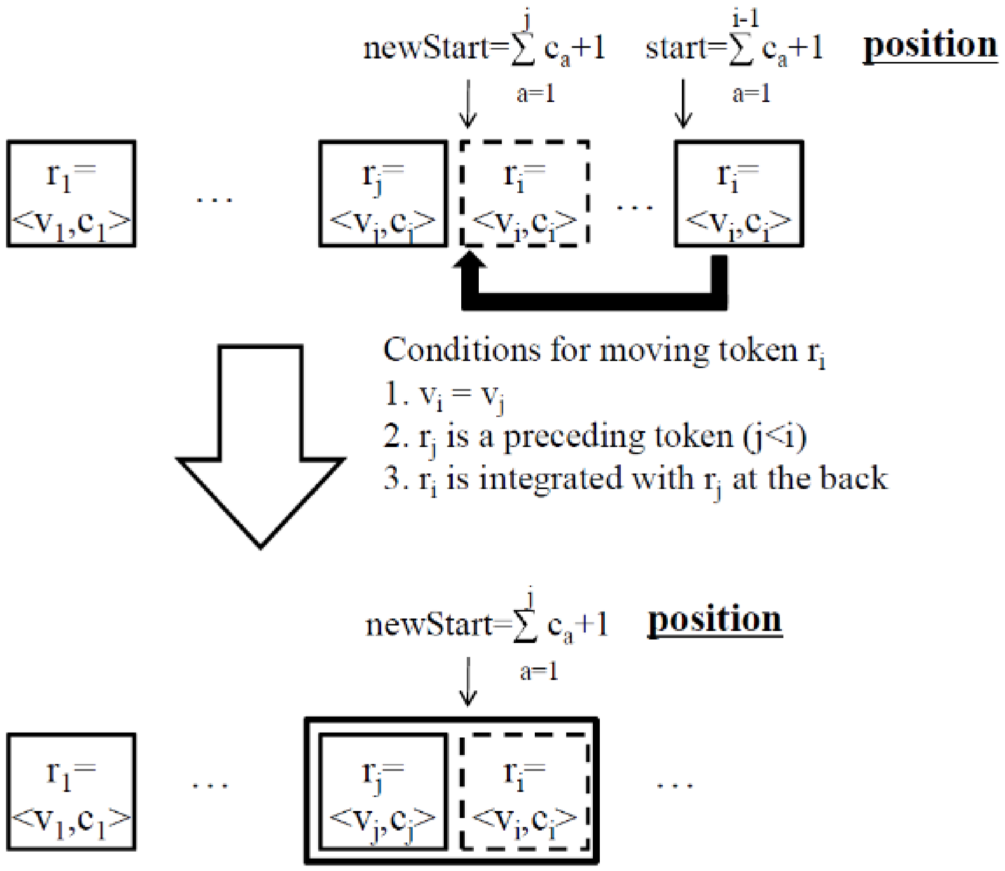
$\sum_{a=1}^{k-1} c'_a + 1 < newStart \le \sum_{a=1}^{k} c'_a$ (Note that k=j).

Proof . We can derive the formula intuitively in Figure 5.

From Lemma 1, we can recover the changed data by reordering of data using R′ and the movement information. Generally, since the size of the position × 2 is less than the size of one token, we can improve the compression ratio of the run-length encoding by data reordering. Until now, we considered only one movement information. In the case that there are many movements, if we keep track of movement information, we can decode data by applying Lemma 1 sequentially. We can move the token r'_6 to the position after r'_2 in R′ with the movement ≪14,8≫. Then, D′ is transformed as follows:

D″ = {1, 1, 2, 2, 2, 2, 2, 2, 3, 3, 4, 4, 5, 5, 5, 5, 5}

Figure 5. Movement information

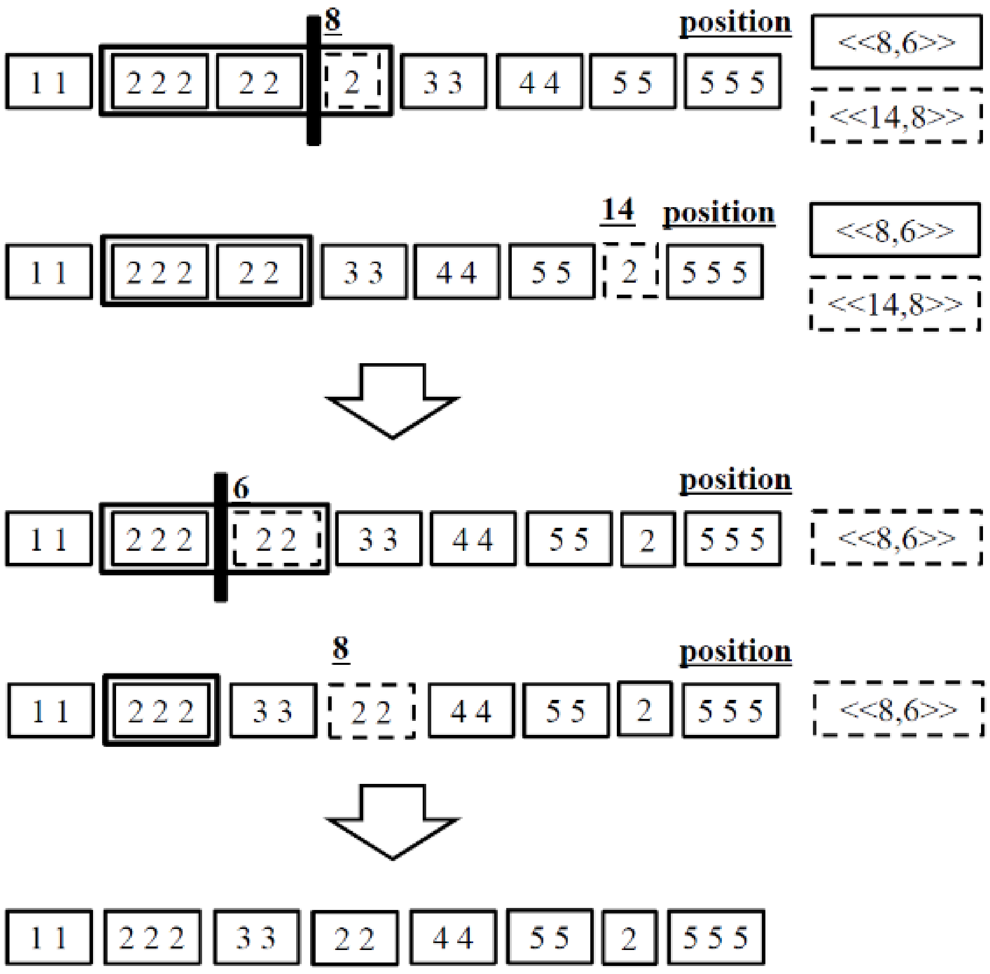

newStart=$\sum_{a=1}^{j} c_a+1$
start=$\sum_{a=1}^{i-1} c_a+1$
position
r_1=<v_1,c_1>
r_j=<v_j,c_j>
r_i=<v_i,c_i>
Conditions for moving token r_i
1. $v_i = v_j$
2. r_j is a preceding token (j<i)
3. r_i is integrated with r_j at the back

R″ = RLE(D′) = {<1,2>,<2,6>,<3,2>,<4,2>,<5,2>,<5,3>}

From R″ and the movement list ≪8,6≫, ≪14,8≫, we can reconstruct R. To reconstruct R, we traverse the movement list reversely. The decoding process for R″ is shown in Figure 6. We first perform decoding with the movement information ≪14,8≫. Using Lemma 1, we reconstruct D′. Then, we perform decoding with the movement information ≪8,6≫ in the same way.

The run-length encoding algorithm with data reordering is shown in Figure 7. In the algorithm of Figure 7, we first encode data using the original run-length encoding in Statement 1. tokenList consists of the list of tokens that have <value, start, count> instead of <value, count>. In the original run-length encoding, we keep only <value, count> for each token, but we revise the run-length encoding to keep <value, start, count>. Then, we read tokens (currToken) sequentially in Statement 6. In the first loop, currToken is set to the second token in tokenList. If currToken can be merged with prevToken (Statement 14), we store the movement information in the form of ≪start, newStart≫ (Statement 16) and merge currToken with prevToken (Statement 17). After that, we set flag to TRUE in order to update the start position of the following tokens (Statement 18). In Statement 13, we update the start position since the current token is moved. In Statement 21-22, if the current token is merged with the previous

Figure 6. Decoding process for R''



token, we remove the current token. Finally, we store <value, count> for each token in tokenList and the movement list in Statement 24.

The run-length decoding algorithm with data reordering is shown in Figure 8. The decoding algorithm is based on Lemma 1. First, we reverse the order in the movement list (moveList) in Statement 1. Then, we read the current movement (currMove) sequentially from the movement list. In Statement 6 and 7, we find the merged token (mToken) corresponding to the current movement (currMove) and divide the merged token (mToken) into the left token (leftToken) and the right token (rightToken) using Lemma 1. leftToken and rightToken are r_j and r_i in Lemma 1, respectively. Note that

$$(mToken.start + mToken.count) - currMove.newStart = \left(\sum_{a=1}^{k-1} c'_a + 1 + c'_k\right) - currMove.newStart = \sum_{a=1}^{k} c'_a - currMove.newStart + 1 (Statement 7)$$

Finally, we move the right token (rightToken) separated from the merged token (mToken) to the original position, and change the start positions of tokens following rightToken due to the removal of rightToken.

Figure 7. Run-length encoding algorithm with data reordering

```
Function Run_Length_Encoding_Reordering (ordered data d[])
begin
  1: tokenList := Run_Length_Encoding(d);
  2: // The index starts from 1
  3: for i:=2; i$\le$the size of tokenList; i++
  4: {
  5:    // initially, the current token is set to the first token in tokenList
  6:    set currToken to the next token in tokenList;
  7:    // flag represents whether there are tokens to be merged
  8:    flag := FLASE;
  9:    for j:=1;  j<the position of currToken in tokenList; j++
 10:    {
 11:       set prevToken to the j-th token in tokenList;
 12:       if flag == TRUE
 13:       {  prevToken.start := prevToken.start + currToken.count ;  }
 14:       else if  (currToken.value== prevToken.value ) and
                  (prevTokn.count+currToken.count $\le 2^{RUN\_LENGTH\_SIZE}-1$)
 15:       {
 16:          add the movement information
                <<currToken.start, prevToken.start + prevToken.count>>;
 17:          prevToken.count := prevToken.count + currToken.count;
 18:          flag := TRUE;
 19:       }
 20:    }
 21:    if flag == TRUE
 22:    {  remove currToken from tokenList;  }
 23: }
 24: store <value, count> for each token in tokenList and
          the movement list;
end
```

Figure 8. Run-length decoding algorithm with data reordering

```
Function Run_Length_Decoding_Reordering (tokenList, moveList)
// tokenList: tokens by run-length encoding with data reordering
// moveList: list of compact movement information
begin
1: reverse the order in moveList;
2: // The index starts from 1
3: for i:=1; i≤ the size of moveList; i++
4: {
5:      currMove := the i-th element in moveList;
6:      find the merged token (mToken) corresponding to
           currMove.newStart in tokenList which satisfies
           the following property;
              $mToken.start < currMove.newStart$ and
              $currMove.newStart \leq (mToken.start + mToken.count - 1)$
7:      divide the merged token (mToken) into
           the left token (leftToken) and the right token (rightToken)
           by computing the following formulas;
              $rightToken.value := mToken.value$
              $rightToken.start := currMove.start$
              $rightToken.count := (mToken.start + mToken.count) -$
                                   $currMove.newStart$
              $leftToken.value := mToken.value$
              $leftToken.start := mToken.start$
              $leftToken.count := mToken.count - rightToken.count$
8:      move the right token (rightToken) to the original position
           updating the start information of tokens in tokenList;
9: }
end
```

Optimization and Partitioning

Observe R′ and R″ carefully. By moving token r'_6=<2,1>, r'_5=<5,2> and r'_7=<5,3> become adjacent in R″. When we merge r''_5=<5,2> and r''_6=<5,3>, we store the movement information ≪15,15≫. In this case, we can drop the movement information. We call this optimization Movement Dropping. In Movement Dropping, when two tokens become adjacent by moving tokens between the two tokens, we merge the two tokens and drop the movement information ≪start, newStart≫, where start=newStart. By merging two tokens, r''_5 and r''_6, in R″, the data D″ is transformed as follows:

$D''' = D'' = \{1, 1, 2, 2, 2, 2, 2, 2, 3, 3, 4, 4, 5, 5, 5, 5, 5\}$
$R''' = RLE(D'') = \{<1,2>,<2,6>,<3,2>,<4,2>,<5,5>\}$

Consider the decoding process with R‴ and two movement information ≪8,6≫ and ≪14,8≫. We do not have the movement information for dividing r'''_5 =<5,5> into r''_5=<5,2> and r''_6=<5,3>. We use a similar idea to dropping of end information in the movement representation ≪start, end, newStart≫. Figure 9 shows the decoding process with R‴ and the movement information ≪14,8≫. With the new start position 8, we divide the merged tokens into two tokens <2,5> and <2,1> as shown in State 1 of Figure 9. Then, we insert <2,1> to the proper position with the start position 14. However, the 14-th position is in token <5,4>. If we do not drop the movement information for adjacent tokens, the start position should be the start position of some token. However, since we drop the movement information for adjacent tokens, if the start position is in the token, we divide the token into two tokens. Thus, we divide token <5,4> into token <5,2> and token <5,2> as shown in State 2 of Figure 9. Then, we insert token <2,1> between token <5,2> and <5,2> in State 3 of Figure 9.

To apply Movement Dropping, we revise the encoding algorithm of Figure 7 and the decoding algorithm of Figure 8. Figure 10 shows the run-length encoding with data reordering using Movement Dropping. In Statement 6-7, we check whether the adjacent two tokens (i.e., leftAdjToken, currToken) are merged. If so, we can merge adjacent two tokens and do not store the movement information (Statement 9-10). Otherwise, we perform the same process in the encoding algorithm of Figure 7.

Figure 11 shows the run-length decoding algorithm with data reordering using Movement Dropping. In Statement 1, we find and divide the merged token corresponding to currMove.newStart in the same way as the decoding algorithm of Figure 8. Then, with currMove.start, we find and divide the merged token by Movement Dropping in Statement 2-4. We

Figure 9. Decoding process for R‴

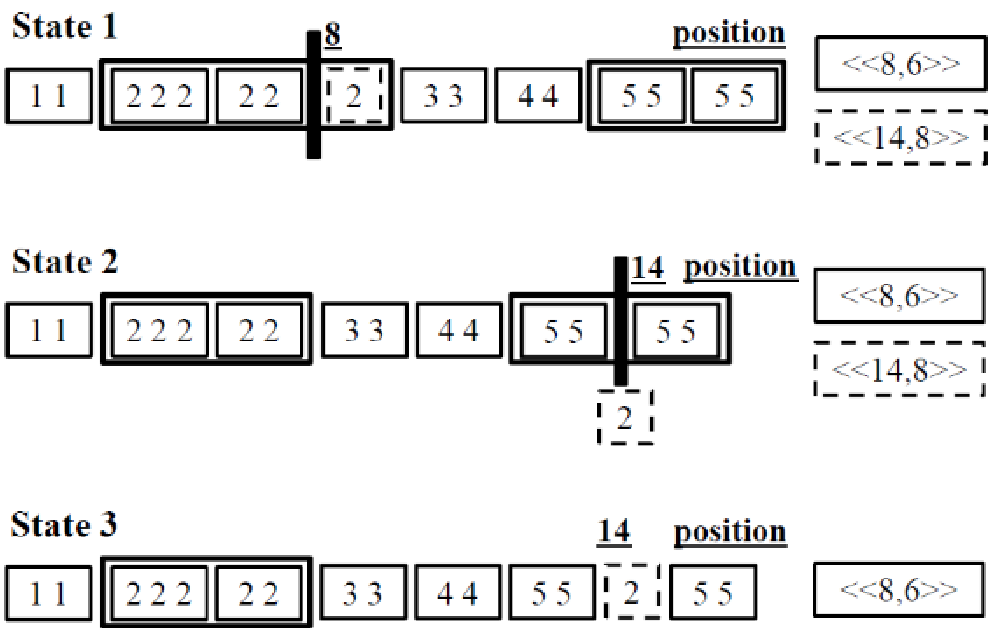
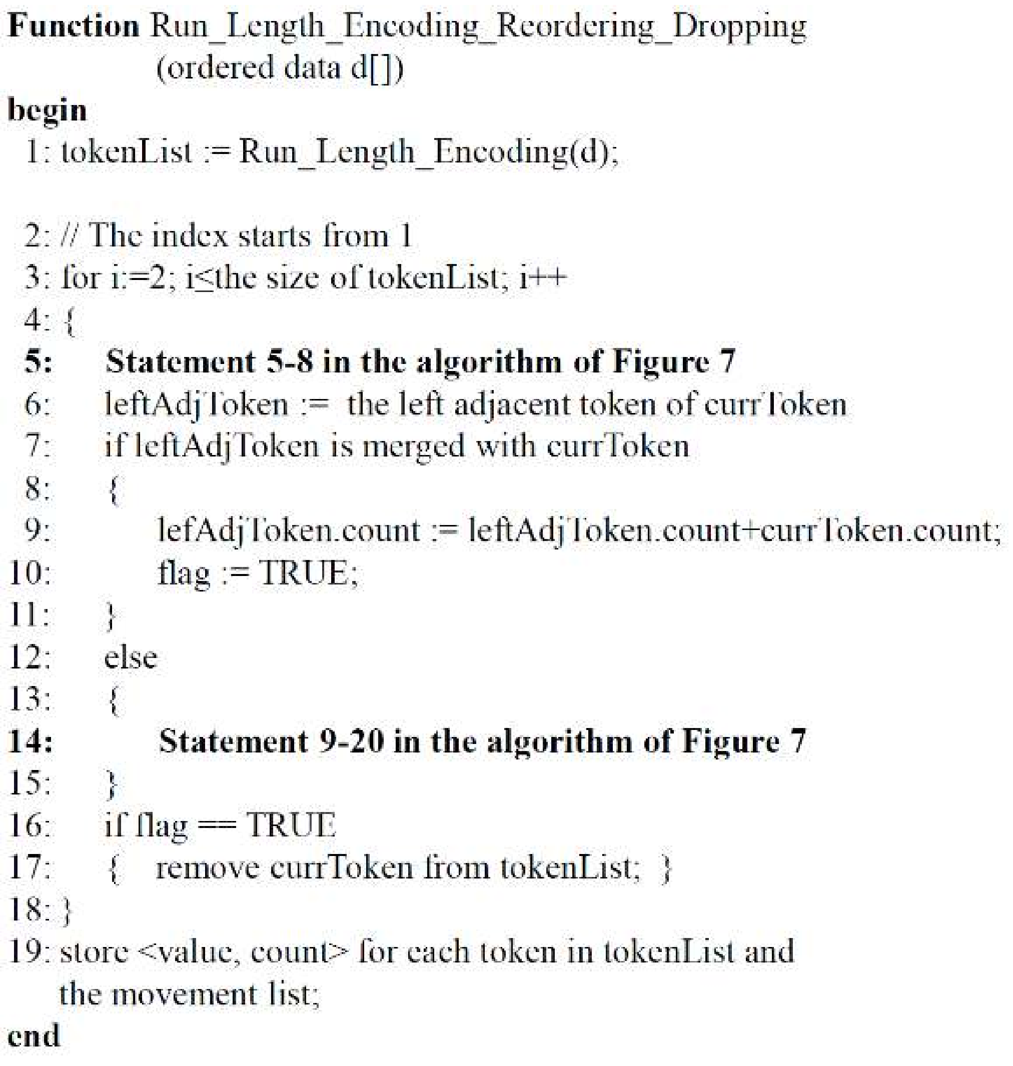


Figure 10. Run-length encoding algorithm with data reordering using Movement Dropping



```
Function Run_Length_Encoding_Reordering_Dropping
            (ordered data d[])
begin
  1: tokenList := Run_Length_Encoding(d);

  2: // The index starts from 1
  3: for i:=2; i≤the size of tokenList; i++
  4: {
  5:     Statement 5-8 in the algorithm of Figure 7
  6:     leftAdjToken := the left adjacent token of currToken
  7:     if leftAdjToken is merged with currToken
  8:     {
  9:         lefAdjToken.count := leftAdjToken.count+currToken.count;
 10:         flag := TRUE;
 11:     }
 12:     else
 13:     {
 14:         Statement 9-20 in the algorithm of Figure 7
 15:     }
 16:     if flag == TRUE
 17:     {   remove currToken from tokenList;  }
 18: }
 19: store <value, count> for each token in tokenList and
     the movement list;
end
```

traverse the token list (tokenList) in order to find tToken which satisfies condition (A or B) in Statement 2. If tToken satisfies condition A, it means that tToken is the merged token and the movement dropping occurred. If there is no token which satisfies condition A, there should be the token which satisfies condition B. In Statement 3, if condition A is satisfied, we divide tToken into two tokens, Token1 and Token3. Then, we insert rightToken (Token2) between Token1 and Token3. Otherwise, we insert rightToken after tToken in Statement 4.

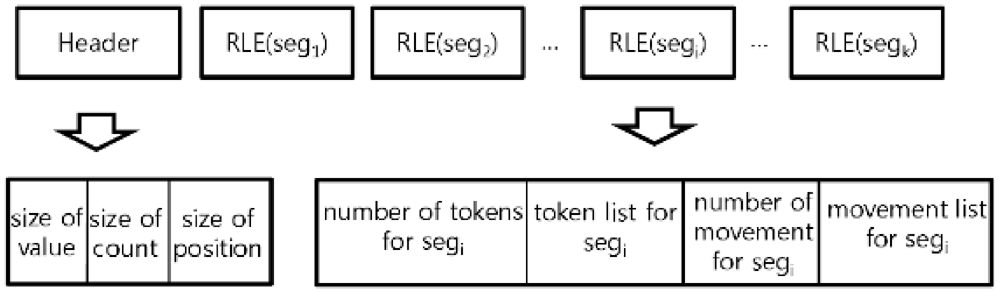
The proposed run-length encoding with data reordering can improve the compression ratio compared to the original run-length encoding. However, if the number of elements is large, the size of the position will be large. Then, we cannot get the space benefit by data reordering. In this case, we can partition ordered data D into segments, seg_1, seg_2, and seg_k, with the same size. Then, we compress each segment using the above run-length encoding with data reordering. Finally, we store compressed data for each segment with the format of Figure 12.

Figure 11. Run-length decoding algorithm with data reordering using movement dropping

```
Function Run_Length_Decoding_Reordering _Dropping
            (tokenList,  moveList)
// tokenList: tokens by run-length encoding with data reordering
// moveList: list of compact movement information
begin
1: Statement 1-7 in the algorithm of Figure 8
2:    find the token (tToken) in tokenList which satisfies
        condition (A or B) while updating the start information
        in tokenList using rightToken.count;
            A: tToken.start < currMove.start  and
                currMove.start ≤ (tToken.start + tToken.count -1)
            B: currMove.start == tToken.start + tToken.count
        * There exists only one token which satisfies condition (A or B).

3:    if A is satisfied, delete tToken and
        insert the following three tokens;
      [Token1] <tToken.value, tToken.start, currMoveData.start – tToken.start>
      [Token2] <rightToken.value, currMoveData.start, rightToken.count>
      [Token3] <tToken.value, currMoveData.start+rightToken.count,
                 (tToken.start + tToken.count) – currMoveData.start>
4:    else if B is satisfied, insert rightToken after tToken;
5: }
end
```

Figure 12. Format to store segments



Since we partition data, we record the number of tokens for seg_i and the number of movement information for seg_i to distinguish segments. The header information includes the size of the value, the size of the count, and the size of the position. Some applications may need real-time processing. To utilize our approach in the applications, we can also apply the above partition concept. If a small number of data is collected at the application, we apply the data reordering techniques to the collected data immediately. Therefore, we can reduce the compression time.

Bucketing Scheme with Data Reordering

In this section, we consider the bucketing scheme with data reordering. The basic concept in the bucketing scheme with data reordering is similar to that in the run-length encoding with data reordering. We explain our technique with the following example. Note that we assume that a token in the bucketing scheme with data reordering is composed of <minVal, maxVal, count> and | means the delimiter between buckets for easy understanding. At the end of the encoding algorithm for the bucketing scheme with data reordering, we store token <(minVal+maxVal)/2, count>.

D = {1.3, 1.45|2.0, 2.2|1.45, 1.35, 1.4|1.6, 1.7, 1.8|1.4, 1.5|1.7, 1.8}
B = BUCKET(D, 0.1) =
{<1.3,1.45,2>,<2.0,2.2,2>,
<1.35,1.45,3>,<1.6,1.8,3>,<1.4,1.5,2>,<1.7,1
.8,2>}

In the run-length encoding with data reordering, if values in two tokens are the same, the two tokens are merged. However, the condition for merging two buckets in the bucketing scheme with data reordering is different from that in the run-length encoding with data reordering. In the bucketing scheme with data reordering, we merge two buckets, b_i and b_j, if they satisfy the following condition instead of the condition $v_i = v_j$.

$M - m \leq 2\varepsilon$,
where m = min(b_i.minVal, b_j.minVal),
and M = max(b_i.maxVal, b_j.maxVal)

That is, if the difference between the minimum value and the maximum value in the merged bucket is less than or equal to 2ε, we can merge two buckets guaranteeing maxi $|d_i - \hat{\ } d_i| \leq \varepsilon$, where d'_i is the approximate value for d_i. For example, in B, we can merge b_1=<1.3,1.45,2> and b_3=<1.35,1.45,3> with the movement information ≪5,3≫. Then, the data D is transformed as follows:

D′ = {1.3, 1.45, 1.45, 1.35, 1.4|2.0, 2.2|1.6, 1.7, 1.8|1.4, 1.5|1.7, 1.8}
B′ = BUCKET(D′, 0.1) = {<1.3,1.45,5>,<2.0, 2.2,2>,<1.6,1.8,3>,<1.4,1.5,2>,<1.7,1.8,2>}

Also, we can merge b'_1=<1.3,1.45,5> and b'_4=<1.4,1.5,2> with the movement information ≪11,6≫ as follows:

D″ = {1.3, 1.45, 1.45, 1.35, 1.4, 1.4, 1.5|2.0, 2.2|1.6, 1.7, 1.8|1.7, 1.8}
B″ = BUCKET(D″, 0.1) = {<1.3,1.5,7>,<2.0, 2.2,2>,<1.6,1.8,3>,<1.7,1.8,2> }

Since the encoding and decoding algorithms for the bucketing algorithm with data reordering are similar to those of the run-length encoding and decoding algorithms with data reordering, we do not mention them in detail.

Optimization and Partitioning

In the bucketing scheme with data reordering, we can also use Movement Dropping. Consider B′ and B″. By moving the bucket b'_4=<1.4,1.5,2>, b'_3=<1.6,1.8,3> and b'_5=<1.7,1.8,2> become adjacent. Therefore, without storing the movement≪13,13≫, we merge b''_3=<1.6,1.8,3> and b''_4=<1.7,1.8,2> as follows:

D‴ = {1.3, 1.45, 1.45, 1.35, 1.4, 1.4, 1.5|2.0, 2.2|1.6, 1.7, 1.8, 1.7, 1.8}

B‴ = BUCKET(D″, 0.1) = {<1.3,1.5,7>,<2.0,2.2,2>,<1.6,1.8,5>}

Finally, we store the following compressed data.

Token List: {<1.4,7>,<2.1,2>,<1.7,5>}

Movement List: {≪ 5, 3 ≫,≪ 11, 6 ≫}

The decoding process for the above compressed data is shown in Figure 13. We divide the merged token <1.4,7> into two tokens, <1.4,5> and <1.4,2> using newStart 6 (State 1). Then, we find and divide the merged token with start

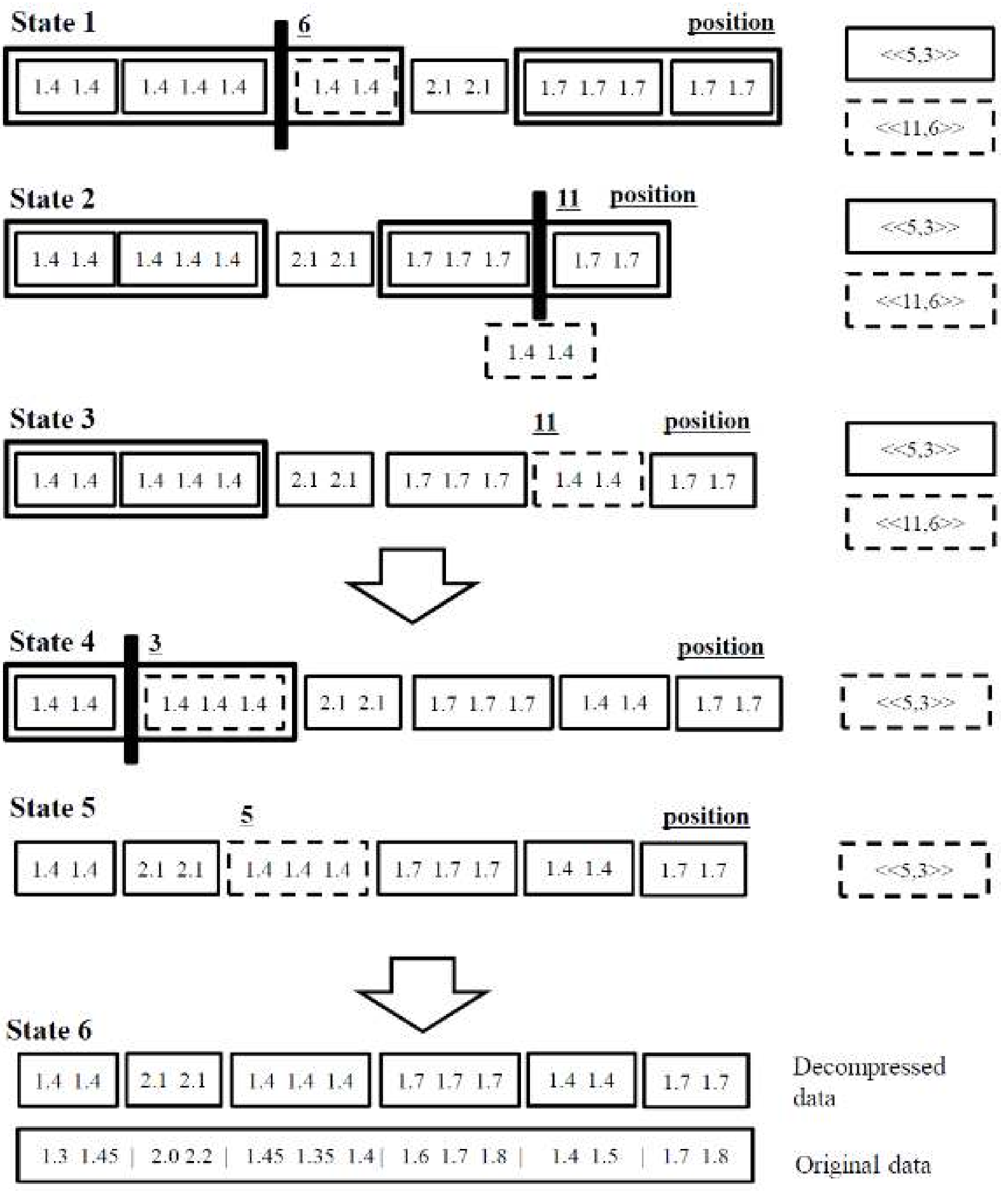


Figure 13. Example for Bucketing Scheme with data reordering

11 (State 2). We insert token <1.4,2> between token <1.7,3> and token <1.7,2> (State 3). Next, we find the merged token with newStart 3 (State 4). Then, we find the position to be inserted. In this case, Movement Dropping does not occur. Finally, we insert token <1.4,3> (State 5).

In the bucketing scheme with data reordering, we devise another optimization technique, Neighbor Merging, as an improved version of Movement Dropping. Consider the following example.

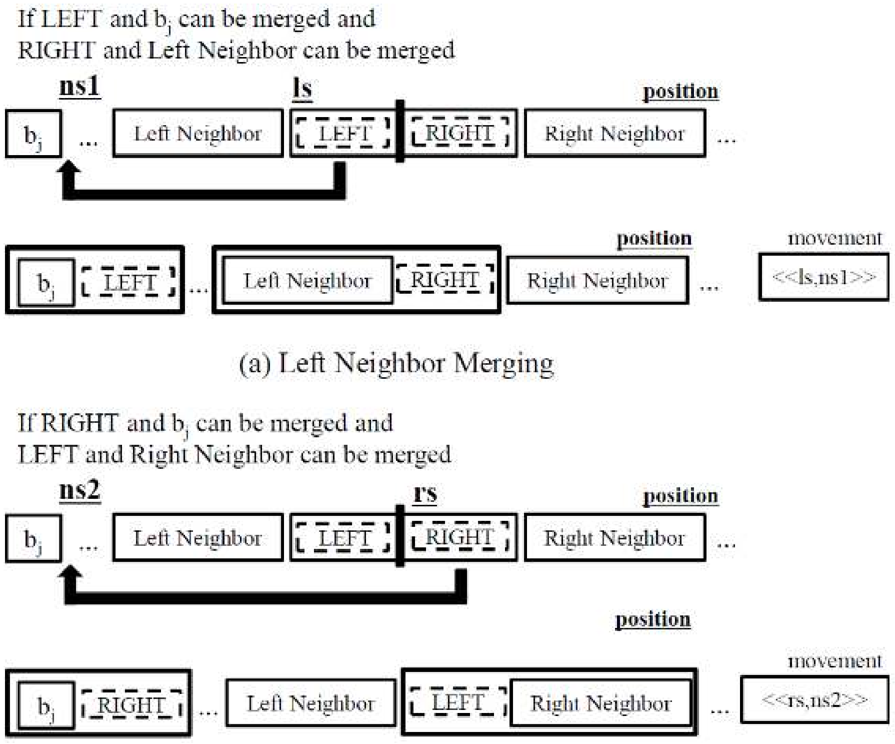
D = {1.3, 1.45|1, 1.1, 1.25|1.35, 1.3, 1.2, 1.3|1.5, 1.6}
B = BUCKET(D, 0.1) = {<1.3,1.45,2>,<1,1.25,3>,<1.2,1.35,4>,<1.5,1.6,2>}

Observe $d_{6,9}$ = {1.35, 1.3, 1.2, 1.3} corresponding to b_3 carefully. {1.2, 1.3} which is the right portion of $d_{6,9}$ can be merged with b_2, and {1.35, 1.3} which is the left portion of $d_{6,9}$ can be merged with b_1. If we move $d_{6,7}$ = {1.35, 1.3}, $d_{8,9}$ = {1.2, 1.3} will become adjacent to b_2 so we will be able to drop the movement information. Although b_3 cannot be merged with any other bucket, we can merge b_3 by dividing it into two parts, {1.35, 1.3} and {1.2, 1.3}. We move {1.35, 1.3} with the movement information ≪6,3≫ as follows:

D′ = {1.3, 1.45, 1.35, 1.3, |1, 1.1, 1.25, 1.2, 1.3|1.5, 1.6}
B′ = BUCK(D′, 0.1) = {<1.3,1.45,4>,<1,1.3,5>,<1.5,1.6,2>}

There are Left Neighbor Merging and Right Neighbor Merging in Neighbor Merging as shown in Figure 14. If the current bucket cannot be merged with any other previous buckets, we try to do Left Neighbor or Right Neighbor Merging. In Left Neighbor Merging, RIGHT is the maximum right portion of the current

Figure 14. Neighbor merging



If LEFT and b_j can be merged and
RIGHT and Left Neighbor can be merged
ns1
ls
position
b_j
Left Neighbor
LEFT
RIGHT
Right Neighbor
movement
<<ls,ns1>>
(a) Left Neighbor Merging
If RIGHT and b_j can be merged and
LEFT and Right Neighbor can be merged
ns2
rs
<<rs,ns2>>

bucket which is merged with Left Neighbor and LEFT is the portion of the current bucket except RIGHT. Therefore, if we move LEFT, RIGHT can be merged with Left Neighbor. We find a preceding bucket which will be merged with LEFT. If there is such the preceding bucket, we merge the preceding bucket with LEFT, and Left Neighbor with RIGHT. In Right Neighbor Merging, LEFT is the maximum left portion of the current bucket which is merged with Right Neighbor and RIGHT is the portion of the current bucket except LEFT. Similarly, we merge the preceding bucket with RIGHT, and LEFT with Right Neighbor. The decoding is performed like Movement Dropping. Note that each bucket should keep elements which are contained in the bucket to apply Left Neighbor or Right Neighbor Merging, while the original bucketing scheme keeps only minVal, maxVal, and Count for the bucket.

If the number of elements is large, we partition data into various segments like the run-length encoding with data reordering.

Due to lack of space, the time complexity analysis is described in the supplemental document (http://islab.kaist.ac.kr/JDM_Reordering_Supplemental_Document.pdf).

Experiments

In this section, we show the effectiveness of compression schemes with data reordering. By comparing our compression schemes with the original compression schemes, we present that our approaches are better than the original compression schemes in terms of the compression ratio.

Experimental Environment

To evaluate the run-length encoding with data reordering and the bucketing scheme with data reordering, we use 10000 temperature readings and 10000 wind speed readings which are collected at one place every minute and are rounded off to the first decimal place (DataSet, 2012). Both readings contain repeated values. However, the temperature readings show more regular patterns than the wind speed readings. The detailed data set description is shown in the supplemental document (http://islab.kaist.ac.kr/JDM_Reordering_Supplemental_Document.pdf). We assume that the size of the value is 4 bytes. We determine the optimal size of the count and the optimal size of the position with 1000 sample data.

Experimental Results

We show experimental results for the run-length encoding with data reordering in Section 7.2.1 and those for the bucketing scheme with data reordering in Section 7.2.2.

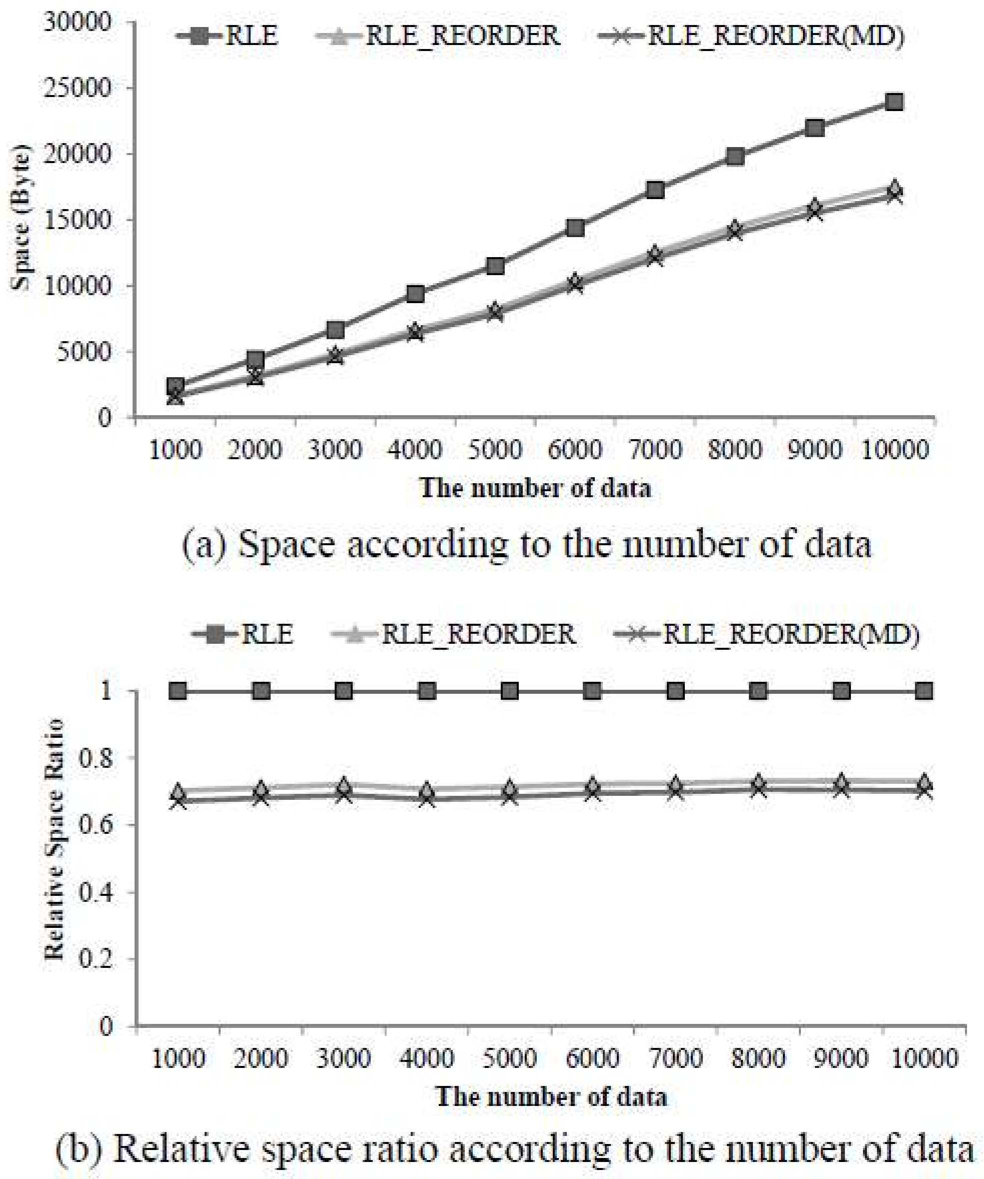
Run-Length Encoding with Data Reordering

Figure 15 shows the amount of space according to the number of data in temperature readings. Varying the number of data from 1000 to 10000, we conduct experiments. In graphs, we denote the original run-length encoding by RLE, the proposed run-length encoding with data reordering by RLE REORDER, and the proposed run-length encoding with data reordering using Movement Dropping by RLE REORDER(MD). In Figure 15-(a), the space increases in proportion to the number of data in all approaches as expected. RLE REORDER and RLE REORDER(MD) are better than RLE in terms of space. Also, RLE REORDER(MD) spends a little less space than RLE REORDER. Therefore, we can improve the run-length encoding by data reordering. Figure 15-(b) shows the relative space ratio for temperature readings. We define the relative space ratio as follows:

Relative Space Ratio =
(Compressed data size by compression scheme with data reordering) /
(Compressed data size by original compression scheme)

Since RLE is the original compression scheme, the relative space ratio of RLE is 1. The relative space ratio of RLE REORDER is approximately between 0.70 and 0.73 and

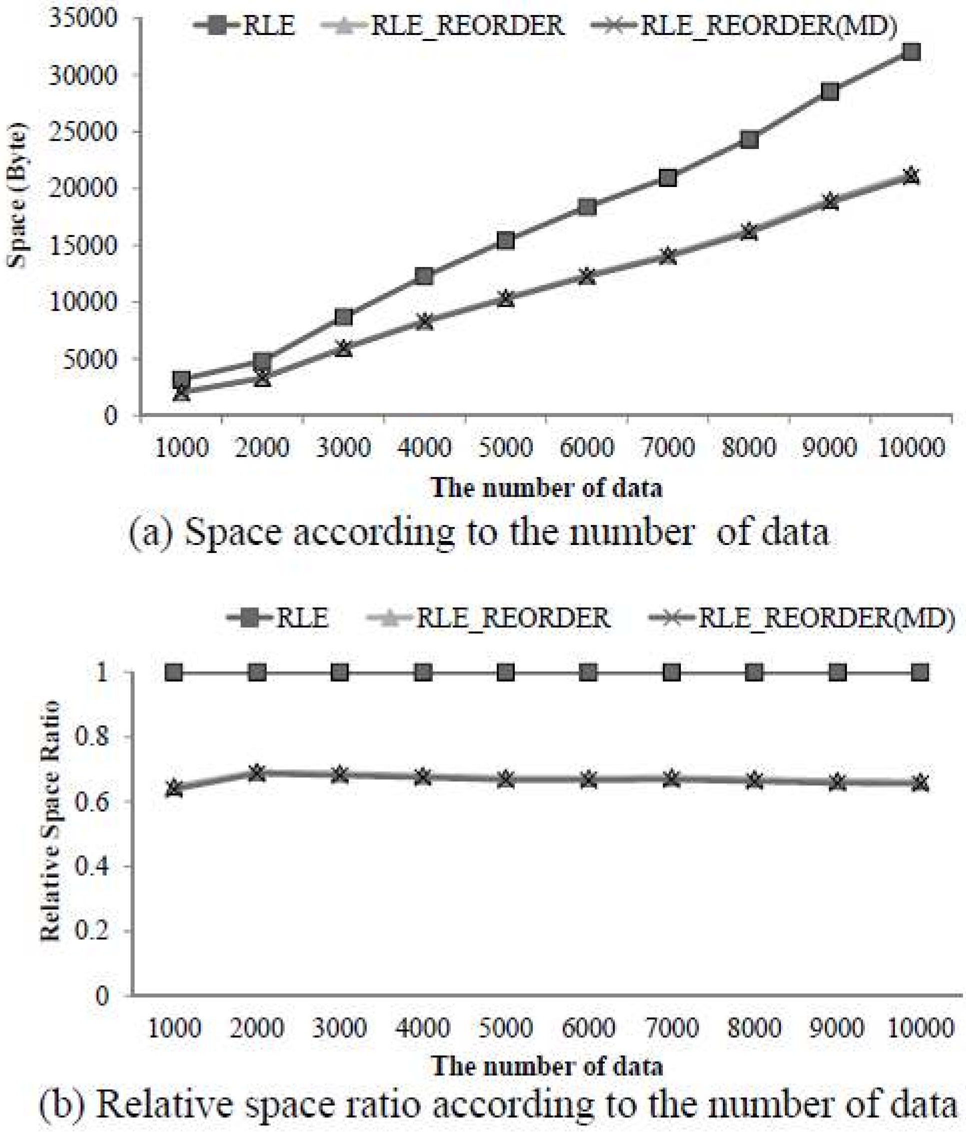
Figure 15. Experiment according to the number of data with temperature readings (Run-length encoding)



that of RLE REORDER(MD) is between 0.67 to 0.71. Therefore, if we use the run-length encoding with data reordering, we can get the space benefit approximately 30% compared to the original run-length encoding. Also, if we use Movement Dropping, we can get the space benefit 3% compared to the method without Movement Dropping. Meantime, since the relative space ratio is nearly constant, we do not scale up to the large sized data. Although the data set is small, we can show the effectiveness of our approach.

Figure 16 depicts the performance of compression schemes with wind speed readings.

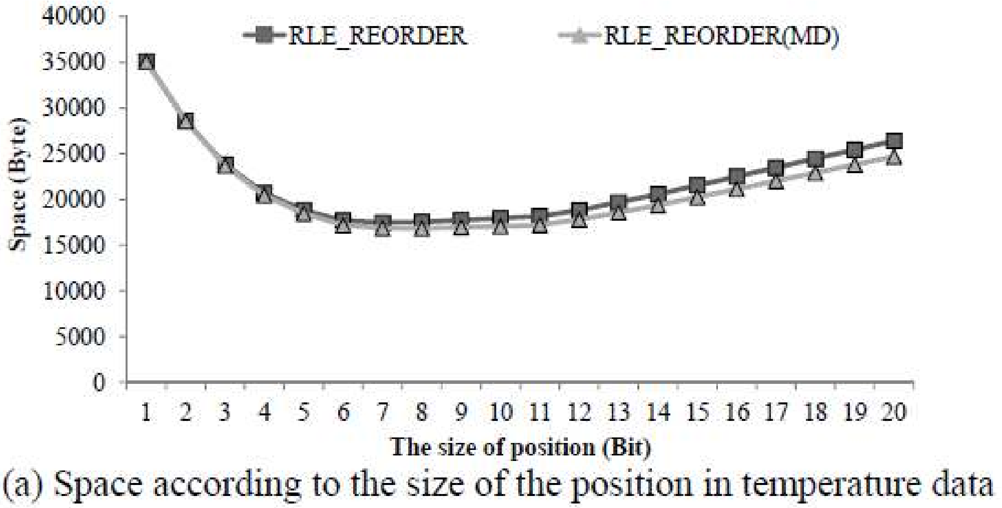
Figure 16. Experiment according to the number of data with wind speed readings (Run-length encoding)



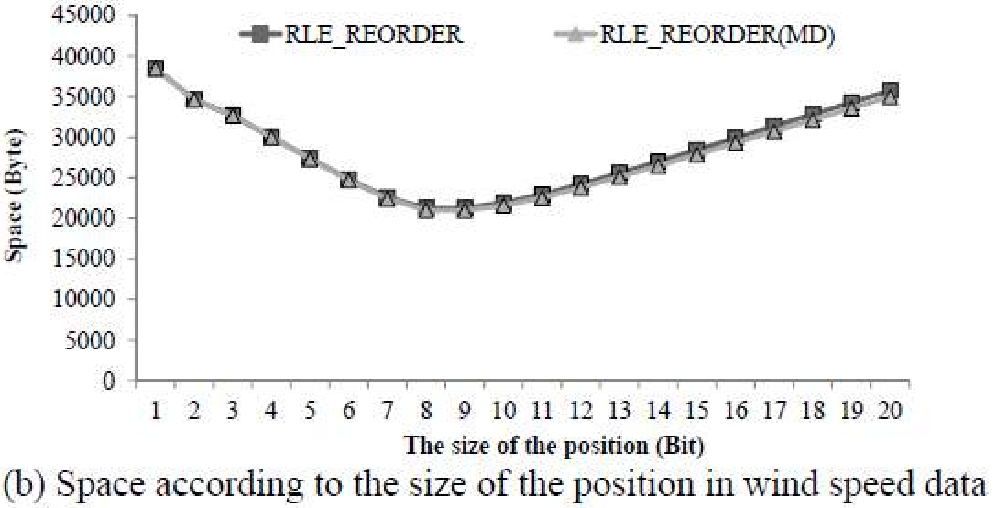
The tendency in Figure 16 is similar to that in Figure 15. However, the relative space ratios of RLE REORDER and RLE REORDER(MD) in Figure 16 are better than those in Figure 15. In Figure 16, the relative space ratio of RLE REORDER is approximately between 0.65 and 0.69 and that of RLE REORDER(MD) is between 0.64 and 0.69. Temperature readings are considered as a more well-organized data set than wind speed readings since the wind speed readings change more sharply than the temperature readings. It means that the temperature readings have less data to be reorganized than the wind speed readings. In addition, since the wind speed readings have irregular patterns, Movement Dropping does not occur frequently as shown in Figure 16.

The size of the position affects the compression ratio of the run-length encoding with data reordering. Figure 17 shows how the amount of space changes according to the size of the position. Note that we partition data with the size 2the $^{\text{size of the position}}$. Therefore, if the size of the position is too small, one segment contains a small number of data. In this case, the chance that a token is moved and merged with other token is reduced. Therefore, the amount of space is big when the size of the position is small in Figure 17. On the other hand, if the size of the position is too big, we need a large amount of

Figure 17. Experiment according to the size of the position in the run-length encoding with data reordering



(a) Space according to the size of the position in temperature data



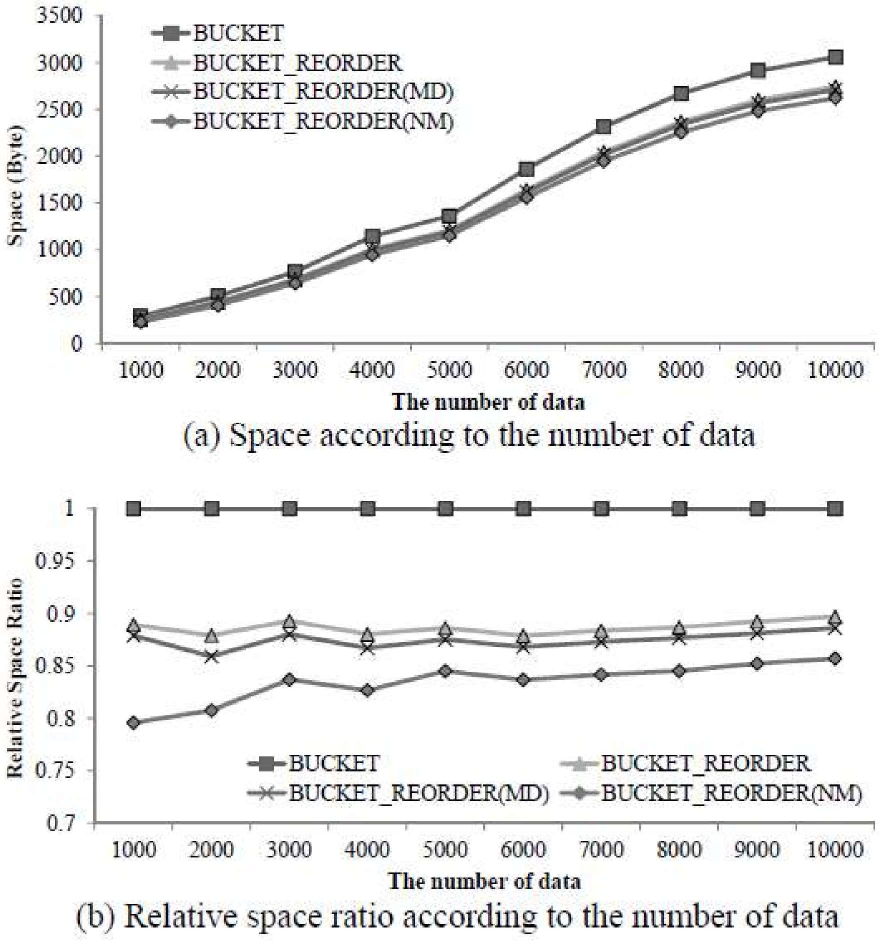
(b) Space according to the size of the position in wind speed data space to store the movement information although we can merge a large number of tokens. Therefore, in Figure 17, the amount of space is big when the size of the position is big. The size of the position is set when the amount of space is minimized.

Bucketing Scheme with Data Reordering

To show the effectiveness of the bucketing scheme with data reordering, we conduct experiments with temperature readings and wind speed readings. In these experiments, we set the error bound to 0.3. Also, we denote the original bucketing scheme by BUCKET, the bucketing scheme with data reordering by BUCKET REORDER, the bucketing scheme with data reordering using Movement Dropping by BUCKET REORDER(MD), and the bucketing scheme with data reordering using Movement Dropping and Neighbor Merging by BUCKET REORDER(NM).

Figure 18-(a) shows the experiment according to the number of data with temperature readings. We can see that the three approaches with data reordering are better than the original

Figure 18. Experiment according to the number of data with temperature readings (Bucketing scheme)

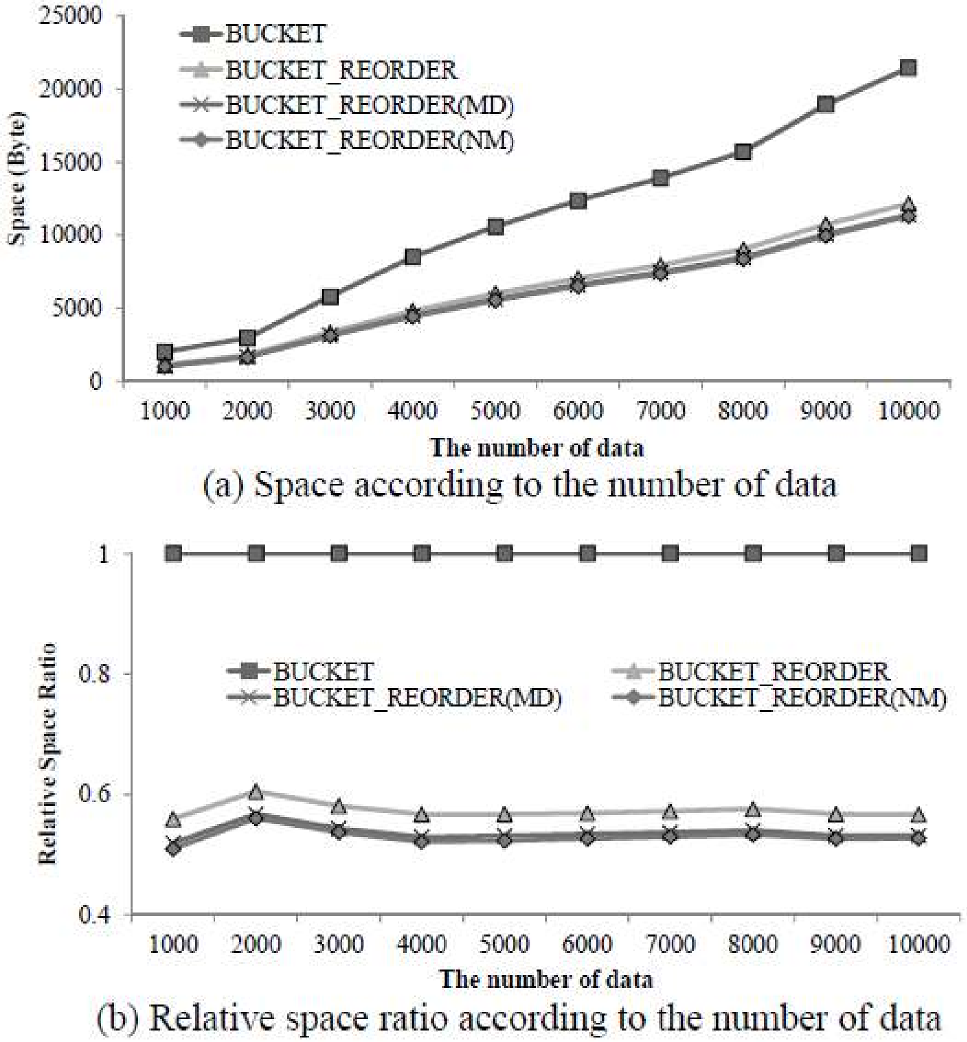


bucketing scheme. Figure 18-(b) shows the relative space ratio according to the number of data. Among three approaches with data ordering, RLE REORDER(NM) shows the best performance as expected. And, BUCKET REORDER(MD) is a little better than BUCEKT REORDER.

Figure 19 shows the experiment according to the number of data with wind speed readings. As mentioned before, the temperature readings change smoothly compared to the wind speed readings. It means that the temperature data has less data to be reorganized than the wind speed data. Therefore, the relative space ratios in Figure 19 are much better than those in Figure 18. The ratios in Figure 18 are between 0.80 and 0.90, while those in Figure 19 are between 0.51 and 0.61.

Experimental results for compression/de-compression time are shown in the supplemental

Figure 19. Experiment according to the number of data with wind speed readings (Bucketing scheme)



document (http://islab.kaist.ac.kr/JDM_Reordering_Supplemental_Document.pdf).

CONCLUSION

The compression schemes can be considered with data reordering. Some previous works have dealt with data reordering in compression schemes. However, it can be applied to only unordered data. In this paper, we consider the run-length encoding with data reordering and the bucketing scheme with data reordering for ordered data. By analyzing the compression schemes, we represent the data movement information for reorganizing data by the compact representation≪start, newStart≫. Based on the compact representation, we propose the encoding and decoding algorithms for both compression schemes. Moreover, we propose two optimization techniques, Movement Dropping and Neighbor Merging. Finally, experimental results show that our approaches with data reordering are better than the original compression schemes in terms of space. Through the experimental results, we show the possibility that we can improve compression schemes by combining data reordering. Based on these results, we will analyze other compression schemes with data reordering.

ACKNOWLEDGMENT

We would like to thank the editor and anonymous reviewers. This work was supported by the National Research Foundation of Korea grant funded by the Korean government (MSIP) (No. NRF-2009-0081365).

REFERENCES

Abadi, D. J., Madden, S., & Ferreira, M. (2006). Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Chicago, IL (pp. 671–682).

Apaydin, T., Tosun, A. S., & Ferhatosmanoglu, H. (2008). Analysis of basic data reordering techniques. In *Proceedings of the Scientific and Statistical Database Management Conference (SSDBM)*, Hong Kong, China (pp. 517-524).

Blandford, D., & Blelloch, G. (2002). Index compression through document reordering. In *Proceedings of the Data Compression Conference (DCC)*, Snowbird, UT (pp. 342-351).

Buragohain, C., Shrivastava, N., & Suri, S. (2007). Space efficient streaming algorithms for the maximum error histogram. In *Proceedings of the International Conference on Data Engineering (ICDE)*, Istanbul, Turkey (pp. 1026–1035).

Burrows, M., & Wheeler, D. J. (1994). *Block-sorting lossless data compression algorithm*. Digital Equipment Corporation.

Chen, H., Li, J., & Mohapatra, P. (2004). RACE: Time series compression with rate adaptivity and error bound for sensor networks. In *Proceedings of the IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS)*, Fort Lauderdale, FL (pp. 124-133).

Chen, Y., Dong, G., Han, J., Wah, B. W., & Wan, J. (2002). Multi-dimensional regression analysis of time-series data stream. In *Proceedings of the Very Large Data Bases (VLDB)*, Hong Kong, China (pp. 323-334).

DataSet. (2012). *Earth climate and weather*. Retrieved October 1, 2012, from http://www-k12.atmos.washington.edu/k12/grayskies/

Degermark, M., Engan, M., Nordgren, B., & Pink, S. (1996). Low-loss tcp/ip header compression for wireless networks. In *Proceedings of the International Conference on Mobile Computing and Networking (MOBICOM)*, Rye (pp. 1–14).

Deligiannakis, A., Kotidis, Y., & Roussopoulos, N. (2004). Compressing historical information in sensor networks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Paris, France (pp. 527-538).

Deligiannakis, A., Kotidis, Y., & Roussopoulos, N. (2007). Dissemination of compressed historical information in sensor networks. *The VLDB Journal*, *16*(4), 439–461. doi:10.1007/s00778-005-0173-5

DeVore, R. A., Jawerth, B., & Lucier, B. J. (1992). Image compression through wavelet transform coding. *IEEE Transactions on Information Theory*, *38*(2), 719–746. doi:10.1109/18.119733

Elmeleegy, H., Elmagarmid, A. K., Cecchet, E., Aref, W. G., & Zwaenepoel, W. (2009). Online piece-wise linear approximation of numerical streams with precision guarantees. In *Proceedings of the VLDB Endowment (PVLDB)*, Lyon, France (pp. 145-156).

Gandhi, S., Nath, S., Suri, S., & Lie, J. (2009). GAMPS: Compressing multi sensor data by grouping and amplitude scaling. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Providence, RI (pp. 771-784).

Goldstein, J., Ramakrishnan, R., & Shaft, U. (1998). Compressing relations and indexes. In *Proceedings of the International Conference on Data Engineering (ICDE)*, Orlando, FL (pp. 370–379).

Graefe, G., & Shapiro, L. D. (1991). Data compression and database performance. In *Proceedings of the ACM/IEEE-CS Symp. On Applied Computing* (pp. 22–27).

Hai, P. N., Lenco, D., Poncelet, P., & Teisseire, M. (2013). Mining representative movement patterns through compression. *Advances in Knowledge Discovery and Data mining, 7818*, 314-326.

Iyer, B. R., & Wilhite, D. (1994). Data compression support in databases. In *Proceedings of the Very Large Data Bases (VLDB)*, Santiago, Chile, (pp. 695–704).

Johnson, D., Krishnan, S., Chhugani, J., Kumar, S., & Venkatasubramanian, S. (2004). Compressing large boolean matrices using reordering techniques. In *Proceedings of the Very Large Data Bases (VLDB)*, Toronto, Canada (pp. 13-23).

Johnson, T. (1999). Performance measurements of compressed bitmap indice. In *Proceedings of the Very Large Data Bases (VLDB)*, Edinburgh, Scotland (pp. 278-289).

Korn, F., Jagadish, H. V., & Faloutsos, C. (1997). Efficiently supporting ad hoc queries in large datasets of time sequences. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, AZ (pp. 289-300).

Liu, W., Kan, A., Chan, J., Bailey, J., Leckie, C., Pei, J., & Ramamohanarao, K. (2012). On compressing weighted time-evolving graphs. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, Maui, HI (pp. 2319-2322).

Ouyang, Z., Memon, N., Suel, T., & Trendafilov, D. (2002). Cluster-based delta compression of a collection of files. In *Proceedings of the International Conference on Web Information Systems Engineering (WISE)*, Singapore (pp. 257-268).

Pennebaker, W. B., & Mitchell, J. L. (1993). *Still image data compression standards*. Springer.

Pinar, A., Tao, T., & Ferhatosmanoglu, H. (2005). Compressing bitmap indices by data reorganization. In *Proceedings of the International Conference on Data Engineering (ICDE)*, Tokyo, Japan (pp. 310-321).

Rao, K. R., & Yip, P. (1990). *Discrete cosine transforms - Algorithms, advantages, applications*. Academic Press.

Ray, G., Haritsa, J. R., & Seshadri, S. (1995). Database compression: A performance enhancement tool. In *Proceedings of the International Conference on Management of Data (COMAD)*, Pune, India.

Reeves, G., Liu, J., Nath, S., & Zhao, F. (2009). Managing massive time series streams with multi-scale compressed trickles. In *Proceedings of the VLDB Endowment (PVLDB)*, Lyon, France (pp. 97-108).

Salomon, D. (2004). *Data compression: The complete reference*. Springer.

Toivonen, H., Zhou, F., Hartikainen, A., & Hinkka, A. (2011). Compression of weighted graphs. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Diego, CA (pp. 965-973).

Wikipedia. (2012). *Run-legnth encoding*. Retrieved October 1, 2012, from http://en.wikipedia.org/wiki/Run-length_encoding

Zhang, J., Liu, H., Ling, T. W., Bruckner, R. M., & Tjoa, A. M. (2006). A framework for efficient association rule mining in XML data. *Journal of Database Management*, *17*, 19–49. doi:10.4018/jdm.2006070102

Chun-Hee Lee received the Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST) in 2010. His research interests include data compression, sensor networks, stream data management, and graph databases.

Chin-Wan Chung is a professor in the Department of Computer Science at the Korea Advanced Institute of Science and Technology (KAIST), Korea. He received a B.S. degree in electrical engineering from Seoul National University, Korea, and a Ph.D. degree in computer engineering from the University of Michigan, Ann Arbor, USA. He was a Senior Research Scientist and a Staff Research Scientist in the Computer Science Department at the General Motors Research Laboratories. He has published 118 papers in the international journals and conferences, published 120 papers in the domestic journals and conferences, and registered 24 international and domestic patents. He received the best paper award at ACM SIGMOD in 2013. He was in the program committees of major international conferences including ACM SIGMOD, VLDB, IEEE ICDE, and WWW. He was an associate editor of ACM TOIT, and is an associate editor of WWW Journal. *He will be the General Chair of WWW 2014. His current research interests include Web, social networks, graph databases, and spatio-temporal databases.*