ABSTRACT

Row-oriented databases (or “row-store”) employ data compression methods (like dictionary encoding) to reduce the I/O cost by decreasing the data sizes. However, there are two limitations on row-stores when applying data compression schemes: (1) row-stores only allow encoding one single value at a time, and (2) they have to pay the decompression cost in query processing. The above shortcomings limit the wide usage of data compression in row-oriented databases.

On the contrary, column-oriented databases (or “column-store”) provide more opportunities for data compression as the values of the same attribute are stored consecutively. In a column-oriented database, compression schemes that encode multiple values crossing multiple rows at once are allowed, but such schemes do not work in row-stores. In addition, column-stores can sometimes perform queries directly on compressed data without decompression, which yields the ultimate performance boost, since the I/O cost is saved by accessing less data and the decompression cost is also avoided. However, column-stores do not consider heavy-weighted compressions since they can not support random accesses, though they have high compression qualities.

This research exam report surveys different compression techniques on both row-stores and column-stores. For each compression scheme, we first review the existing work, then present our ideas. In addition, we introduce a class of queries that both row-stores and column-stores miss key opportunities to answer them efficiently. Then we present a new data structure, which allows both light-weighted and heavy-weighted data compressions, to answer the queries efficiently.

1. INTRODUCTION

Data compression methods like Huffman coding [25] and Lempel-Ziv (LZ) encoding [52] have exhibited their importance in many areas [56, 51, 50]. However traditional database systems (i.e., row-oriented databases) do not widely apply data compression techniques [3], since data compression methods can not help database systems to gain much performance improvement. Data compression indeed reduces the I/O cost by reducing the size of stored data, but row-oriented databases need to pay additional cost on decompressing the data. If the CPU cost of decompressing the data outweighs the saving of the I/O cost, then the performance of the database is reduced [31, 29, 33]. Database community cares more about query performance than space cost, since improving the performance is the goal instead of saving disk space. Disk space is cheap and is becoming cheaper rapidly [3, 2].

Row-oriented databases store values belong to an entity/tuple together; that is values of an attribute are distributed in different tuples. This kind of architecture design is perfect for OLTP (Online Transactional Processing) query workloads, since they usually require to find/add/delete an entity. Row-oriented databases often employ dictionary encoding [49, 17] to map a value from a large code to a smaller one. For example, a dictionary for the state column in Figure 1(b) may map “California” to “43”, “Arizona” to “4” and so on (Section 3 introduce the detailed dictionary encoding). Sometimes, they also apply symbol frequencies based compressions like Huffman encoding [25, 24]. Data compression in row-oriented database has two limitations: (1) it can only encode one value at a time, and (2) it requires additional decompression time. The first limitation can not be solved due to the design of the row-oriented database architecture.

Assume the state column is encoded by dictionary encoding and a user wants to find all the customers in California, then the following SQL query is issued:

```
SELECT Customer_Name FROM Customer WHERE State = 'California'
```

To answer this query, one straightforward method (called eager decompression [33, 31]) is to decompress the whole column first, then find all the rows such that the state is “California”. However, this eager decompression method costs too much on decompressing unnecessary values. They ignore the fact that some operations can be directly applied on compressed data, e.g., projection. [29, 57] proposed another strategy called lazy decompression: data stays in the compressed format in main memory and is only decompressed when it is necessary. To answer the above query, the query condition “California” is first encoded by dictionary encoding, i.e., mapping “California” to “43”, then the query is translated as the following:

```
SELECT Customer_Name FROM Customer WHERE State = 43
```
Then the state column can be kept in the compressed format to answer this query. Compared with decompressing the whole column, compressing the query condition is more efficient. However, lazy decomposition increases the number of I/O of later operations in the query plan since it increases the size of intermediate results [17, 16]. Therefore, to avoid the increase of intermediate results, [17, 16] proposed another strategy called transient decomposition, which allows data in its compressed format in the input and output of each operator and is temporarily decompresses only when it is required within the operator. However, this strategy also has one limitation, i.e., it may cause the same attribute to be decompressed multiple times.

Since the state column is encoded by run-length encoding, thus the answer of this query can be directly obtained from the encoding, which is 1000, without decompression. Performing queries directly on compressed data yields the ultimate performance boost, since column-stores not only save the I/O cost by accessing less data but also improve the performance by not paying the decompression cost. However, column databases only consider light-weighted compressions, since heavy-weighted compressions do not support random accesses [3].

In this research exam report, we survey different compression techniques on both row-stores and column-stores. For each compression scheme, we first review the existing work, then discuss its advantages and disadvantages and also present our own ideas. In addition, we introduce a class of queries, called relationship queries that both row-store and column-store miss key opportunities to answer it efficiently. Then we present a new data structure, which allows both light-weighted and heavy-weighted data compressions, to answer such queries efficiently.

Roadmap The rest of the paper is organized as follows. Section 2 provides the background of row- and column- databases. Section 3, Section 4, Section 5, and Section 6 illustrate the widely utilized data compression methods in row- and column- databases. Section 7 and Section 8 show the strategies of decompression and compression respectively. We introduce the relationship queries in Section 9. Finally, Section 10 concludes this research exam report.

2. BACKGROUND

2.1 Row-oriented databases vs. column-oriented databases

Conceptually, a database table is a relation table, which is a two dimension representation. Each row corresponds to a distinct real-world entity or a relationship, and each column is an attribute of entities [27, 26]. For example, each row in Figure 2(a) refers to a customer entity, and each column is an attribute of customers. The State column stores all the state attributes of entities.

The state-of-the-art database architectures to store such...
relation tables physically are row-oriented databases and column-oriented databases.

Row-oriented databases (also called row-stores) store the table row-by-row, which means all information about an entity is stored together. For instance, as shown in Figure 2(b), all the information of the first customer is stored together, then followed by all the information of second customer, etc. The early database systems\(^1\) (Microsoft SQL Server [13], IBM DB2 [35], Oracle [41], PostgreSQL [55] and MySQL [44]) are all row-oriented databases, since this design is optimized for the most common database application at the time: online transactional processing (OLTP). For example, one example is to update an order made by customer X on Y time. The OLTP queries tend to access data on the granularity of an entity, thus a row-oriented database is preferable, since all the information of an entity is stored together in row databases.

In recent years, businesses started to ask analytical queries on their data in order to make decisions. For example, the CEO wants to know the number of customers in California. The analytical queries (also called OLAP(Online Analytical Processing)) usually read large portion of data from few attributes, row-stores are inefficient to answer such OLAP queries. On the contrary, Column-oriented databases (also called column-stores) store all attribute information together. For example, as shown in Figure 2(c), all the customer IDs are stored together, then all of the customer names, etc. A column database is preferable for OLAP queries, since all the values in an attribute are stored together. The well-known column-stores include the academic systems MonetDB [11, 45] and C-Store [54] and also the commercial system SybaseIQ [42]. In addition, many database vendors combined the column-store into their existing systems, e.g., SAP [22, 23] and Oracle.

Row-stores and column-stores are designed for different query workloads. Consider the query SELECT Count(*) from Customer WHERE Gender=“Female” in Figure 3(a), the row-store need to access all the tuples first, then compute the number of appearances of females on the Gender column\(^2\), while the column-store only needs to access the Gender column. On the contrary, for query SELECT * from Customer WHERE CName=“Linda Lee” in Figure 3(b), the row-store only need to access one particular tuple, while column-store has to access all the columns in order to construct the output result.

**Discussion.** Based on the above analysis, we make the following conclusions.

1. Row-oriented databases is preferable for applications that process a single tuple/record at one time.

2. Column-oriented database is suitable for applications that process many tuples/records on few attributes.

Therefore, there is no one-fit-all database design, and businesses nowadays tend to maintain two databases, one is row-store for the transactional queries while the other one is column-store for the analytical queries.

---

\(^1\)Before 1990s, databases are all row-oriented.

\(^2\)The gray colored cells indicate unnecessary values, while the blue colored cells mean the necessary values.
2.2 Storage of column-oriented databases

In column databases, each column is stored separately. In order to successfully recover the original tuples, columns from a same table should maintain the same order. To meet this requirement, one simplest way to represent a column in a column-oriented database is to assign each column a tuple identifier (see Figure 4(b)), which can be regarded as an integer key column. This additional key column increases the size of data and the I/O cost. But it allows each column to maintain its own order. Then in the recovery phase, the additional identifier column can be applied to find the correct order in the table.

However, modern column databases maintain virtual identifier (see Figure 4(a)) instead of explicit one by using the position offset of the tuple in the column. Here has an assumption: each column is fixed-width. For example, the i-th value in column \( D \) can be obtained at the location \( \text{start}(D) + i \times \text{width}(D) \), where \( \text{start}(D) \) and \( \text{width}(D) \) mean the start address of the column \( D \) and the width of the column \( D \) respectively. Note that, this fixed-width with virtual ids storage has at least two advantages: (1) random access to i-th value is efficient, and (2) fixed-width column-stores is easier to be compressed.

3. DICTIONARY ENCODING

In this section, we introduce the most widely used compression method Dictionary encoding\[10, 32, 18, 3\] in database community. The main idea of the dictionary encoding is to use smaller code for each value. There are two kinds of dictionary encoding schemes: word-aligned dictionary encoding and bit-aligned dictionary encoding.\(^3\) Word-aligned dictionary encoding can be applied in both row-stores and column-stores. However, bit-aligned dictionary encoding can only be utilized in column-stores. In the following, we will introduce these two in details.

3.1 Word-aligned dictionary encoding

Word-aligned dictionary encoding maintains a global mapping table \( M \). Each tuple \((s, t) \in M\) is a mapping from source entity \( s \) to a target code \( t \). Note that, \( t \) is a 4 bytes integer\(^4\). Databases apply dictionary encoding mainly on columns with string type and small domain. The reason is that (1) columns with string type usually cost large space, and (2) small domain requires small global table.

\[\begin{array}{c|c|c|c|c|c}
\text{Customer} & \text{CID} & \text{CName} & \text{State} & \text{Gender} \\
\hline
1 & 1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 & 3 \\
4 & 4 & 4 & 4 & 4 \\
\end{array}\]

(a) Column store with virtual ids

\[\begin{array}{c|c|c|c|c|c}
\text{Customer} & \text{CID} & \text{CName} & \text{State} & \text{Gender} \\
\hline
1 & 1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 & 3 \\
4 & 4 & 4 & 4 & 4 \\
\end{array}\]

(b) Column store with explicit ids

Figure 4: Column store with virtual Ids and with explicit Ids. Column-store with virtual Ids save more space than that with explicit Ids, but column-stores with explicit Ids are more flexible and each column can maintain its own order.

\[\begin{array}{c|c|c|c|c|c|c}
\text{State} & \text{State Dictionary} \\
\hline
California & Alabama \\
Arizona & Alaska \\
\end{array}\]

Figure 5: Dictionary encoding on Sales.State column.

Join on string columns is inefficient, since string matching operation slows down the performance. However, if \( D_1 \) and \( D_4 \) are encoded with dictionary encoding, then the join condition can be transformed from string matching to integer equivalent checking, which is much efficient. Note that, in this query, we do not need to decompress \( D_1 \) and \( D_4 \), since we only need to check whether there exists equivalent values in both columns.\[31\] also claims that equi-joins can be executed directly on compressed data.

\[\text{SELECT } A.D_2, \text{COUNT(*) FROM } A,B \text{ WHERE } A.D_1 = B.D_4\]

3.2 Bit-aligned dictionary encoding

Word-aligned dictionary encoding can be utilized in both row-stores and column-stores. However, it maps the original

\(^4\)Assume the number of distinct values is no more than \(2^{32}\).
values to words, which is not the most compact coding strategy. For example, word-aligned dictionary encoding can not compress integer columns, since the size of each original value is already 4 bytes. Here we give an example to state that word-aligned dictionary encoding wastes space. Consider the Gender column, there are only two distinct values, “male” and “female”. Theoretically, we can use one bit (bit-aligned dictionary encoding) to represent them, i.e., 0 for female and 1 for male. However, the word-aligned dictionary encoding need to use 4 bytes for each value, which costs 16 times larger space than bit level encoding.

Though using bit-aligned dictionary encoding can achieve higher compression quality, row-stores cannot apply it. The main reason is that in row-stores, values in the same attribute are stored separately. Thus, each value of a tuple/entity should be encoded individually. In this way, row-stores will read unnecessary data even it is bit-aligned encoded. For example, to achieve an encoded Gender value, row-store first gets a page, then finds the corresponding tuple, then read at least one byte to get a 1-bit code, which means the other 7-bits are useless. In addition, the other attributes in the tuple are useless. Therefore, row-stores cannot achieve bit level compression quality.

However, column-stores totally change this landscape. [3] proposed a bit-aligned dictionary strategy in column-databases. Let $D$ be the cardinality of a column, them each value can be encoded in $\lceil \log_2 D \rceil$ bits. Given a column, it first calculates the number of bits $n$ utilized for the domain $D$ by using the formula $\lceil \log_2 D \rceil$. Then [3] computes how many codes can fit in 1, 2, 3 or 4 bytes. Consider the State example again, each value can be expressed in 6-bits (that is $\lceil \log_2 50 \rceil$). Table 1 shows the number of such 6-bits codes in 1, 2, 3 or 4 bytes. One of the following will be chosen to produce the final mapping table. For example, 3 encoded values in 2 bytes is selected, then each set of three original values are combined together to be encoded in 2 bytes.

### Table 1: Number of 6-bit codes that in different bytes.

<table>
<thead>
<tr>
<th># of 6-bit codes</th>
<th>1 byte</th>
<th>2 bytes</th>
<th>3 bytes</th>
<th>4 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For example, the values Arizona and Texas are encoded as 000011 and 101011 respectively. Then there is an entry $e$ in the dictionary:

$UUUU000001101011 \rightarrow \text{Arizona Texas}$

where $U$ is an unused bit.

The decoding algorithm works as follows:

1. Read 2-bytes to get an entry $e$;
2. Lookup the entry $e$ in dictionary to obtain three values at once.

The advantage of appending to bytes is that systems will obtain large CPU savings. It is a fact that column-stores are I/O efficient but CPU bounded [67, 3]. Therefore, to get rid of being CPU-limited when using data compression on column-stores, appending to bytes is a good engineering design choice.

**Retrieve a single value** As we described above, several values are encoded together, say three values are encoded into one 2 bytes. Then a natural question is how to get a single decoded value. The answer is to perform bits shifting by using masks like 0000000000011111 and 0000111111000000, which is an efficient operation. For example, for the encoded value $UUUU000011101011$, we can use masks 0000000000 111111 and 000111111000000 to get two single encoded values instead of three combined values as follows:

$\langle (UUUU 000011101011 \& 0000000000111111) \gg 0 \rangle = 101011$

$\langle (UUUU 000011101011 \& 0001111111000000) \gg 6 \rangle = 000011$

**Cache-aware Optimization** In our above discussion, we pack encoded values into 2 bytes, which is motivated by the size of L1/L2 cache. In order to make sure the high performance of the decompression, we want the dictionary mapping table to be able to be stored in L1/L2 cache. Consider the above example again, each entry is fit into 2 bytes, and the number of entries is $64^2=4096$. Thus, the size of the dictionary is only $4096 \times 2 = 8KB$, which can be easily fit in L1/L2 cache.

**Variable Length Dictionary** Till now, the dictionary encodings that we introduced are all fixed length encodings. [7, 6, 9] proposed variable-length dictionary encoding methods, however, the variable-length dictionary encodings can not gain good performance compared with the fixed-length dictionary encodings. Therefore, column databases prefer to adopt fixed-length dictionary encodings in order to gain the benefits of high performance.

### 3.3 Discussion

One limitation of the existing dictionary encodings is that an additional mapping table is required. We observe that if a column is integer column like many foreign key columns (e.g., CustomerID column and ProductID column in the Order table), there is no need to maintain such a mapping table, since from each integer is mapped to a unique binary expression. For example, assume the domain size is 50, then an integer 17 is transformed as 010001. The first 0 is a padding bit to make it to 6-bits, since this domain size requires 6-bits coding. Once get rid of the mapping table, decompression of the dictionary encoding becomes even faster and the encoded data is easier to be fit into L1/L2 cache.

### 4. RUN-LENGTH ENCODING

Compared with row-stores, column-stores allow a larger variety of data compression algorithms. For example, run-length encoding (RLE) [3, 2, 4, 1, 48, 21, 36, 50] is an attractive approach for compressing data in a column-store.\(^5\)

\(^5\)Actually, RLE can be utilized in in row-oriented systems, but it is only used for large string attributes that have many blanks or repeated characters. It has very narrow applications in row-stores.
Run-length encoding compresses continuous duplicate values in a column to a compact singular representation. For example, RLE compresses \( k \) continuous duplicates whose value is \( t \) into one tuple \((t, k)\), i.e., (value, count) pair. RLE is widely used in column-oriented databases where attributes are stored consecutively and runs of the same value are common.

![Figure 6: Example of run-length encoding.](image)

Figure 6 gives an example of applying run-length encoding to the `CustomerID` column. The size of the original column is \( 4 \times 10^8 \) bytes (4 GB), while the size of the encoded column (the standard run-length encoding) is only \( 8 \times 10^6 \) bytes (8 MB), which is only 0.2% of the uncompressed one. RLE is an attractive compression in column-store not only because it has a high compression quality, but also it can speedup the performance. Assume a user wants to know how many orders that customer 2 has made. Then the following query is issued.

```
SELECT Count(*) FROM Order WHERE CustomerID = 2
```

To answer this query, one straightforward way is to uncompressed the encoded column, then perform that query upon the uncompressed data. However, this method is inefficient. Actually this query can be directly answered on the compressed data. Since the encoded column has tuple \((2, 50)\), so 50 is the number of the orders made by customer 2. The ability of answering queries directly on compressed data makes column-stores more efficient than row-stores.

### 4.1 Varieties of run-length encodings

The standard run-length encoding does not support random access. Consider the query \( \pi_B \sigma_{A=1}(R) \), where \( A,B \) are two columns of \( R \). Assume \( A \) is encoded by run-length encoding. To answer this query, column \( A \) should be uncompressed first, since using \( t \) to find a tuple \((t, k)\), we only know the number of the duplicates is \( k \). But we do not know the start position in the column. Thus, it is impossible to get the corresponding values in \( B \). To solve the above limitation, there are two possible extensions: (1) using a triple (value, start position, run length) in the encoded value, and (2) using (value, start position). For the first one, it is trivial to see that, using the start position and length, it is each to obtain a range that satisfies \( \sigma_{A=1}(R) \). However, for the second one, the length can be calculated by \( s_1 - s_2 \) where \( s_1 \) and \( s_2 \) are the start position of \( t \) and \( t + 1 \) respectively.

### 4.2 Discussion

To apply run-length encoding to a column, the column itself should have the following features: 1. the column is sorted. 2. the fanout of this column is high. The first requirement is easy to understand. If the column is not sorted, then element with the same values are not grouped together. The second requirement is utilized to measure the average number of duplicates. Only when the number of duplicates is large, run-length encoding can get benefits. The definition of fanout is provided in the following.

**Definition 1. Fanout.** Given a relationship table \( R \) and a foreign key \( D \) of it, pointing to the ID of an entity table \( E \), we define as fanout of \( R.D \) in \( R \), also called fanout of \( E.ID \) in \( R \) the ratio of the number of tuples of \( R \), divided by the number of tuples in \( E \) (which is also the number of unique values of the ID of \( E \)).

To make a short conclusion: RLE saves space only when the column is sorted and the fanout is high. Otherwise, applying RLE may result in space waste.

### 5. Bitmap Encoding

In this section, we introduce bitmap index [14, 43, 60, 46, 66, 59, 30], which is designed for column-stores. The most attractive feature of the bitmap index is that, it is efficient to execute bit-wise operations AND, OR and NOT. We first introduce the uncompressed bitmap index, which may have large space cost if the size of column is large. Then we introduce compressed bitmap indices in order to save space. Note that, the compressed bitmap indices still support bitwise logical operations. Therefore, compressed bitmap not only save the space cost, but also improve the performance [63, 61, 65, 62, 39].

![Figure 7: Example of bitmap index.](image)
5.1 Uncompressed bitmap index

A bitmap index consists of a set of bit-vectors. Each bit-vector indicates the occurrences of one distinct value of the indexed attribute. That is, for each distinct value, one bit-vector is maintained. Figure 7 shows an example of uncompressed bitmap index for Order table\(^6\). The indexed attribute is the ProductID column, which has four distinct values inside. Then for each distinct value (i.e., 1, 2, 3, and 4), one bit-vector is built with size of the indexed attribute. The default values in a bit-vector are zeros. The i-th position of the j-th bit-vector is set to 1 if the i-th position in the indexed attribute has value j. In Figure 7, we label the four bit-vectors as \(b_1, b_2, b_3\) and \(b_4\) for convenience. For example, the 6-th position in the ProductID column is value 2, therefore, the 6-th position of the second bit-vector (i.e., \(b_2\)) is set to 1.

With such bitmap index, it is efficient to answer range queries. For example, consider the query `SELECT OrderID FROM Order WHERE ProductID \leq 2`. Then a bitwise logical operation \(b_1 \land b_2\) is the main operation in this query, which is proven to be efficient, since bitwise logical operations are supported by modern CPU [64, 47]. In addition, [15, 66, 14, 34] conducted experiments to compare the performance of B-tree with bitmap. The results shown that bitmap indices perform better than tree-based schemes in many data warehouse applications. [47] also studied how to extend the bitmap indices to support queries on multiple tables.

The above SQL query has condition on one single attribute. For queries having conditions on multiple attributes, it is straightforward to process such queries. Consider the query `SELECT OrderID FROM Order WHERE ProductID \leq 2 AND CustomerID \geq 100`, then a bitmap on ProductID and a bitmap on CustomerID are used separately to produce two intermediate temporary bitmaps. They are the result of “ProductID \leq 2” and “CustomerID \geq 100” respectively. Then the final answer can be obtained by operating an AND bitwise operation on these two temporary bitmaps. Using bitmaps to answer such queries is efficient if the bitmap index size is not large and the number of bitmap is few. However, in many real-life applications, both the sizes of bitmaps and the number of bitmaps are high, since the cardinalities of attributes are usually high and the table size is big.

To solve the limitation, compression is applied upon the bitmap index. One simple solution is to apply existing text compression schemes like LZ77\(^7\) [37, 38]. However, the performance of logical bitwise operations on such compressed bitmaps is slow, since they need to decompress the bitmap first, which is time-consuming. In order to improve the performance, [5, 8, 39, 59, 65] proposed specialized schemes that allow bitwise operations directly on compressed bitmaps. The representative compressed bitmaps are the Byte-aligned Bitmap Code (BBC) [5, 8] and the Word-Aligned Hybrid (WAH) [39, 59, 65]. Both of them are based on run-length encoding. They use a counter to represent a long run of 0s or 1s, and literal sequences to represent a mix of 0s and 1s.

Compare the uncompressed bitmap, BBC and WAH, we have the following observations:

- WAH outperforms BBC about 12 times faster and uses only 60% more space.
- WAH is faster than the uncompressed scheme and uses less space.

WAH achieves higher performance since it is a CPU-friendly design.

Details of WAH. Since WAH has been proven to be the most efficient compressed bitmap index, we introduce it in details here. WAH has two kinds of words, one is literal words and one is fill words. The first significant bit of a word is utilized as a flag to distinguish between a literal word (using 0) and a fill word (using 1). Let \(s\) be the number of bits in a word. The lower \(s - 1\) bits of a literal word contain the bit values from the original bitmap. That is directly copying from the bitmap to the literal word. For a fill word, the second most significant bit is a fill bit, which is utilized to indicate whether this is a 0s run or 1s run. The remaining \(s - 2\) bits store the fill length\(^8\). WAH only access words rather than bytes or bits, which is a CPU-friendly design.

Figure 8 is an example of representing 128 bits in WAH. Each literal word has 31 bits from the bitmap while each fill word represents the runs with lengths as a multiple of 31 bits. First, WAH first divides the bits into 31-bit groups (See the second line)\(^9\), then represents each 31-bit group using either literal word or fill word. The fourth line shows the final WAH words. To perform a logical operation on the WAH words, each 31-bit group should be matched from two sides, and then generate the result groups.

5.2 Discussion

Compressed bitmap indices are important in column-stores, since they not only save space but also improve the performance. As we mentioned before, bitmap indices are more efficient than B-tree indices in most applications. Bitmap indices are perfect for selection queries like we presented before, but for queries with join operations, there is still room for more researches to explore how to use bitmap index to speedup the join performance.

6. HEAVY-WEIGHTED ENCODINGS

Some other well-known data compression schemes like Huffman encoding [24, 25, 53], Arithmetic Encoding [58, 33, 48] and Lempel-Ziv Encoding [28, 52, 2] can also be applied in database. They have high compression quality.

\(^6\)To simplify the represent, we only choose a small subset of the Order table.

\(^7\)LZ77 is utilized in gzip.

\(^8\)Note that, WAH stores the length in the s-2 bits rather than the original bits, which is different from the literal word.

\(^9\)Its hexadecimal representation is shown in the third line.
Figure 8: A WAH bit vector. Each WAH word (last row) represents a multiple of 31 bits from the bitmap, except the last word that represents the four leftover bits.

but they do not support random accesses. We call this kind of compressions Heavy-weighted compression. To retrieve a single value in the compressed column, the whole column should be decompressed first. As we claimed before, the goal of databases is to achieve high performance rather than save space, so such heavy-weighted compressions are usually ignored by databases, especially column-oriented databases, which have high performance requirement.

In Section 9, we will introduce a new data organization, which is designed for relationship queries. This organization allows heavy-weighted compression, since no random accesses are required within each encoded structure.

Figure 9: Relative cost of different decompression strategies with different number of join conditions on string columns for queries with 1 – 2 join tables.

7. DECOMPRESSION STRATEGY

In the previous sections, we have introduced many well-known data compression methods in both row-stores and column-stores. In this section, we would like to discuss three different decompression strategies in databases: eager decompression [33], lazy decompression [29, 57] and transient decompression [16, 17].

7.1 Eager decompression

This is the most straightforward way to operate on compression data. That is decompressing the data before sending it to each operator, this approach is called eager decompression [33]. Eager decompression is utilized in early databases. It has one advantage, i.e., it has minimal requirement on code changes in existing databases, which means such strategy is easy to plugin to the existing database systems. However, its limitation is the slow performance, especially when the data is big. In addition, it also ignores the fact that some operators can be applied directly on the compressed data, no need to decompress the data. [31] shows that projection operation and natural join operation can be performed directly on compressed data.

Figure 10: Relative cost of different decompression strategies with different number of join conditions on string columns for queries with 3 – 4 join tables.

7.2 Lazy decompression

In order to solve the limitation of eager decompression, [29, 57] proposes another “opposite” decompression strategy, called lazy decompression. It allows data to keep in compressed format during the query processing as long as possible and is decompressed only when it has to. However, it brings another limitation, that is it increases the size of intermediate results due to the late decompression, which results in the increasing of the I/O cost of the later operations in the query plan like sorting [17].

7.3 Transient decompression

[16, 17] proposed another decompression strategy called transient decompression. The main idea is that the data stays compression in the input and output of each operator, but the data is temporarily decompressed within an operator. This strategy requires to modify the standard relational operators.

7.4 Discussion

Compared these three different decompression strategies, we have the following observations:

1. Lazy decompression usually outperforms eager decompression.
We theoretically compare the space cost of each compression in order to provide a guide of selecting the one with minimal space cost. Figure 13 shows the most compact way to encode a fragment in key/foreign key columns, as a function of (a) number N of elements in the column (vertical axis), and (b) the size of the underlying domain D (horizontal axis). Not surprisingly, uncompressed array (UA)\(^1\) are never the most compact method. (For one, they always take more space than bit-aligned compressed arrays.) In the common cases, where \(D > 27\) and \(N \leq D/8\), compressed bitmaps are more compact than bit-aligned compressed arrays when \(\frac{D}{128^{x}+1} \leq N < D/8\), for any \(x \geq 1\). Note that, the detailed proofs can be found in our own paper [40].

\[\text{Figure 13: Space cost comparison of different encodings.}\]

Each area is colored by the compression method with minimal space cost.

\[\text{In practise, such theoretical comparison may not be satisfied by database designers, since they want to find more straightforward approach for compression choice. Therefore, we give a decision-tree based compression selection approach (as shown in Figure 12), which is more accurate than the one proposed in [2].}\]

9. RELATIONSHIP QUERIES

In this section, we study a new kind of queries, called relationship queries. A relationship query performs joins, selections and semijoins (WHERE clause subqueries) over tables that correspond to entities and binary relationships (i.e., edges). Intuitively, it discovers target entities that are reachable from source entities specified by the query. Relationship queries also compute aggregated scores for the target entities by applying aggregation functions on measured attributes found on the target entities, the source entities and the paths from the sources to the targets. Relationship queries are a superset of graph reachability queries and of tree pattern queries. The formal definition of the relationship queries is shown below:

\[\text{Definition 2. Relationship Queries. An SQL query over relationship tables and entity tables, is a relationship query if (1) in its algebraic form, it involves \(\sigma, \pi, \Join, \times\) operators and an optional aggregation operator \(\gamma\) at the end; (2) each join and semijoin condition involves an equality between (primary or foreign) key attributes and (3) aggregate-}\]

\(^{1}\)UA means storing integers in an array without any compression.

---

**Figure 11:** Relative cost of different decompression strategies with different number of join conditions on string columns for queries with more than 5 join tables.

2. For numerical attributes, transient decompression usually outperforms eager decompression and lazy decompression.

Note that, decompressing numerical attributes are cheap, thus transient decompression usually outperforms eager decompression and lazy decompression. But for string attributes, it is not easy to make a choice between these three decompression strategies. Figure 9, Figure 10 and Figure 11 show the relative cost of each decompression strategy on TPC-H data [20]. As shown, for queries with fewer joins (Figure 9), lazy decompression beats transient decompression and eager decompression, since the lazy decompression does not pay too much penalty on increasing the intermediate results. However, for queries with many joins (Figure 11), transient decompression outperforms the others, since the size of intermediate results increased by lazy decompression dominates the whole query cost.

8. ANALYSIS OF DIFFERENT ENCODINGS

In this section, we analyze the different encodings that we mentioned before. Table 2 shows each encoding can or can not be applied by row-stores and column-stores and also gives the formula to calculate the space cost for each of them.

<table>
<thead>
<tr>
<th></th>
<th>Row-store</th>
<th>Column-store</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dictionary</td>
<td>Yes/No</td>
<td>Yes</td>
<td>(N \cdot \lceil \log_{2} D \rceil)</td>
</tr>
<tr>
<td>RLE</td>
<td>No</td>
<td>Yes</td>
<td>(4 \cdot \cdot \cdot S)</td>
</tr>
<tr>
<td>Bitmap</td>
<td>No</td>
<td>Yes</td>
<td>(8 \cdot \lceil \log_{128} D - N \rceil)</td>
</tr>
</tbody>
</table>

Table 2: Summary of different encodings. \(D\) is the domain size, \(N\) is the number of values and \(S\) is the number of distinct values. Since run-length encoding has many variations, so we use \(c\) to denote how many elements in each run-length entity, it can either be 2 or 3. For dictionary encoding, only word-aligned dictionary encoding can be applied in row-stores. For column-stores, both bit-aligned and word-aligned can be applied. So we say “Yes/No” in row-store for dictionary encoding.

---

**Figure 13:** Space cost comparison of different encodings.
tions group-by on a primary key or foreign key, i.e., they group-by on an entity set.

Existing databases cannot answer this kind of queries efficiently. The main limitations are: The following limitations hurt in-memory column databases when answering relationship queries.

1. Large intermediate results are materialized. Column databases use an operator-at-a-time “bulk-processing” model in order to avoid cache miss problem in traditional tuple-at-a-time iterator model based pipeline execution. [45] Bulk-processing produces complete intermediate results for each operator. When the size of intermediate results is large, materializing intermediate results is both time and space consuming. For example, the size of intermediate result in the third join operation of AS query can be hundreds of billions. Such big intermediate results cannot be fully loaded into caches, which results in lots of slow memory reads.

2. Heavy-weighted compressions are not considered. Though heavy-weighted compressions like Huffman encoding [24, 25] have high compression quality, they do not support random access. To randomly access one value in a compressed column, column databases need to decompress the whole column first. The decompressing time usually dominates the whole query processing time, especially when the data is big. Therefore, column databases do not apply heavy-weighted compressions for performance consideration.

<table>
<thead>
<tr>
<th>System</th>
<th>PubMed</th>
<th>SemmedDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Postgres</td>
<td>20.92</td>
<td>6.84</td>
</tr>
<tr>
<td>MonetDB</td>
<td>3.69</td>
<td>1.53</td>
</tr>
<tr>
<td>FastR</td>
<td>1.72</td>
<td>1.36</td>
</tr>
<tr>
<td>MonetDB</td>
<td>2.15</td>
<td>1.13</td>
</tr>
</tbody>
</table>

Table 4: Space cost for each system (in GB). Numbers in bold are the smallest ones.

To avoid the above limitations, we propose a new data organization with small storage granularity fragment (as shown in Figure 14). Informally, a column can be partitioned into s fragments, where s is the cardinality of this column. More precisely, for each relationship table \( R(F_1, F_2, M_1, \ldots, M_n) \) where \( F_1, F_2 \) are foreign keys while \( M_i \) (\( i \in [1, n] \)) is a measure attribute, we make two copies \( R_1 \) sorted by \( F_1 \) and \( R_2 \) sorted by \( F_2 \). For each copy, FastR produces \( n + 1 \) fragments for each value in the sorted column. Consider copy \( R_1 \), for each entry value \( t \in F_1 \), FastR produces \( n + 1 \) fragments \( \pi_{F_2} \sigma_{F_1=t}(R_1) \), \( \pi_{M_1} \sigma_{F_1=t}(R_1) \), \ldots \( \pi_{M_n} \sigma_{F_1=t}(R_1) \). Each fragment is encoded by different compression methods.

We propose the FastR algorithm running upon the index. FastR employs a bottom-up pipeline mechanism to avoid large intermediate results. Unlike column databases, FastR does not require random access within fragments, because FastR’s query processing either uses all values in a fragment or none of them at all. This feature provides opportunity to use compressions that do not support random accesses.

Experimental Results. We conducted experiments to evaluate the performance of FastR against the existing databases. We run five queries on PubMed dataset [12] and one query on SemmedDB dataset [13]. Each reported running time is the

---

**Figure 12: Decision tree of selecting proper compression scheme.**

**Figure 14: FastR Index \( \mathcal{I}_{R.D} \).** The lookup table stores offsets of fragments. The fragments of different columns have different encodings.
average time over 5 runs, excluding the first run on Postgres and MonetDB, which is used just to warm the memory up. The Postgres and MonetDB are state-of-the-art row-oriented database and column-oriented database respectively.

Table 3 reports the average running time of each query for each system, using 8 threads. We had the following observations.

- FastR outperforms MonetDB by about 170 times on the average (see ratio columns).
- MonetDB outperforms Postgres, which is explicable by the analytic nature of relationship queries.

In addition, Table 4 presents the overall space costs. FastR has the lowest space cost in both PubMed and SemmedDB. This is because each fragment is encoded by a suitable compression scheme. Seen from the above experiments, FastR outperforms all the existing databases and uses less space cost.

Applying different compressions. In addition, we also conducted experiments to evaluate the decompression performance of these encodings for two kinds of fragments: one is fragments on foreign key columns containing only unique values. The other one is fragments on measure attributes, which have many duplicates. For the first case, we generated columns in zipf distribution with factor $s = 1.5$ and the domain size is 1 billion. Then we randomly choose 8000 fragments whose sizes are located in [100000-1000, 100000+1000]. We observed from Table 5 that compressed bitmap achieves the highest compression quality (it saves 69.25% space) and the highest decompression performance (it is about 30 times faster than Huffman). Huffman has the worst performance, since the domain size is large which requires Huffman to maintain a large decoding table that can not be fitted in L1 or L2 caches.

Table 5: Space cost and decompression time of Dictionary, Bitmap and Huffman. Domain size is 1 billion, data follows zipf distribution with factor $s = 1.5$. Fragments only contain unique values, which simulates fragments in foreign-key columns.

<table>
<thead>
<tr>
<th>Query</th>
<th># elements</th>
<th># columns</th>
<th>compression ratio</th>
<th>1 thread</th>
<th>2 threads</th>
<th>4 threads</th>
<th>8 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pubmed</td>
<td>100000</td>
<td>8000</td>
<td>76.23%</td>
<td>1535.506</td>
<td>864.594</td>
<td>450.890</td>
<td>378.227</td>
</tr>
<tr>
<td></td>
<td>151.6</td>
<td>790.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>70.3</td>
<td>100.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>717.487</td>
<td>5.643</td>
<td></td>
<td>198.6</td>
<td>4.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>73.08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>29520.5</td>
<td>4474.8</td>
<td></td>
<td>5662.2</td>
<td>5.662</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>900.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>211.2</td>
<td>10.8</td>
<td></td>
<td>53.1</td>
<td>4.7</td>
<td></td>
<td>0.031</td>
</tr>
<tr>
<td></td>
<td>1535.506</td>
<td>864.594</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>912.3</td>
</tr>
<tr>
<td></td>
<td>29688.662</td>
<td></td>
<td></td>
<td>14934.780</td>
<td>725.446</td>
<td></td>
<td>151.6</td>
</tr>
<tr>
<td></td>
<td>6,409,707</td>
<td>198.6</td>
<td></td>
<td>147,273,421</td>
<td>8000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>100000</td>
<td>8000</td>
<td>73.08%</td>
<td>52198.713</td>
<td>29688.662</td>
<td></td>
<td>151.6</td>
</tr>
<tr>
<td></td>
<td>717.487</td>
<td>5.643</td>
<td></td>
<td>198.6</td>
<td>4.5</td>
<td></td>
<td>0.031</td>
</tr>
<tr>
<td></td>
<td>73.08</td>
<td></td>
<td></td>
<td>53.1</td>
<td>4.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>29520.5</td>
<td>4474.8</td>
<td></td>
<td>5662.2</td>
<td>5.662</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>900.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>211.2</td>
<td>10.8</td>
<td></td>
<td>53.1</td>
<td>4.7</td>
<td></td>
<td>0.031</td>
</tr>
<tr>
<td></td>
<td>1535.506</td>
<td>864.594</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>912.3</td>
</tr>
<tr>
<td></td>
<td>29688.662</td>
<td></td>
<td></td>
<td>14934.780</td>
<td>725.446</td>
<td></td>
<td>151.6</td>
</tr>
<tr>
<td></td>
<td>6,409,707</td>
<td>198.6</td>
<td></td>
<td>147,273,421</td>
<td>8000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We created columns in zipf distribution with factor $s = 1.5$ and the domain size is 100. Table 6 reports the decompression time and compression quality for Huffman and BCA with different number of fragments and different size of fragments. As shown, Huffman achieves the highest compression quality and has comparable decompression performance against BCA. Compared with the results in Table 5, we noticed that the decompression performance of Huffman is significantly improved. This is because the domain size is small (say 100) here, which can be fitted into L1 cache. This results also indicate that Huffman is perfect for measure attributes. Finally, we observed that multiple threads improve the decompression performance for all the encoding methods. For Huffman, the performance is continuously be improved with the increasing of threads, while the improvements of bitmap and dictionary become slower with more threads. This is because that Huffman is more CPU bounded than memory bounded.

10. CONCLUSION

Column databases provide more opportunities for compression than row databases. This research exam surveyed different compression techniques on both row-stores and column-stores and discussed how to integrating compression and query execution. We also theoretically analyzed the compression quality of different compression methods. In addition, we proposed a new data organization to answer relationship queries efficiently by allowing both light-weighted and heavy-weighted compressions.

11. REFERENCES

Table 6: Space cost and decompression time of Dictionary and Huffman. Domain size is 100, data follows zipf distribution with factor $s = 1.5$. Fragments contains duplicates, which simulates fragments in measure attributes.

<table>
<thead>
<tr>
<th></th>
<th># elements</th>
<th># columns</th>
<th>compression ratio</th>
<th>1 thread</th>
<th>2 threads</th>
<th>4 threads</th>
<th>8 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dictionary</td>
<td>100</td>
<td>8000000</td>
<td>21.88%</td>
<td>1381.167</td>
<td>801.313</td>
<td>410.039</td>
<td>348.422</td>
</tr>
<tr>
<td></td>
<td>1000000</td>
<td>80000</td>
<td>21.88%</td>
<td>1397.182</td>
<td>798.372</td>
<td>386.327</td>
<td>317.879</td>
</tr>
<tr>
<td></td>
<td>10000000</td>
<td>80</td>
<td>21.88%</td>
<td>1286.579</td>
<td>652.020</td>
<td>333.645</td>
<td>283.507</td>
</tr>
<tr>
<td>Huffman</td>
<td>100</td>
<td>8000000</td>
<td>15.38%</td>
<td>5055.162</td>
<td>2543.277</td>
<td>1280.826</td>
<td>668.050</td>
</tr>
<tr>
<td></td>
<td>1000000</td>
<td>80000</td>
<td>11.41%</td>
<td>4635.229</td>
<td>2332.007</td>
<td>1174.428</td>
<td>612.555</td>
</tr>
<tr>
<td></td>
<td>10000000</td>
<td>80</td>
<td>11.39%</td>
<td>4374.838</td>
<td>2201.003</td>
<td>1108.435</td>
<td>578.143</td>
</tr>
</tbody>
</table>


