Chapter 7
Compression

As discussed in Chap. 5, SanssouciDB is a database architecture designed to run transactional and analytical workloads in enterprise computing. The underlying data set can easily reach a size of several terabytes in large companies. Although memory capacities of commodity servers are growing, it is still expensive to process those huge data sets entirely in main memory. Therefore, SanssouciDB and most modern in-memory storage engines use compression techniques on top of the initial dictionary encoding to decrease the total memory requirements. The columnar storage of data, as applied in SanssouciDB, is well suited for compression techniques, as data of the same type and domain is stored consecutively.

Another advantage of compression is that it reduces the amount of data that needs to be shipped between main memory and CPUs, thereby increasing the performance of query execution. We discuss this in more detail in Chap. 16 on materialization strategies.

This chapter introduces several lightweight compression techniques, which provide a good trade-off between compression rate and additional CPU-cycles needed for encoding and decoding. There are also a large number of so-called heavyweight compression techniques. They achieve much higher compression rates, but encoding and decoding is prohibitively expensive for their usage in our context. An in-depth discussion of many compression techniques can be found in [AMF06].

7.1 Prefix Encoding

In real-world databases, we often find the case that a column contains one predominant value and the remaining values have low redundancy. In this case, we would store the same value very often in an uncompressed format. Prefix encoding is the simplest way to handle this case more efficiently. To apply prefix encoding, the data sets need to be sorted by the column with the predominant value and the attribute vector has to start with the predominant value.

H. Plattner, *A Course in In-Memory Data Management*,
DOI: 10.1007/978-3-642-36524-9_7, © Springer-Verlag Berlin Heidelberg 2013

To compress the column, the predominant value should not be stored explicitly every time it occurs. This is achieved by saving the number of occurrences of the predominant value and one instance of the value itself in the attribute vector. Thus, a prefix-encoded attribute vector contains the following information:

- number of occurrences of the predominant value
- valueID of the predominant value from the dictionary
- valueIDs of the remaining values.

7.1.1 Example

Given is the attribute vector of the country column from the world population table, which is sorted by population of countries in descending order. Thus, the 1.4 billion Chinese citizens are listed at first, then Indian citizens and so on. The valueID for China, which is situated at position 37 in the dictionary (see Fig. 7.1a), is stored 1.4 billion times at the beginning of the attribute vector in uncompressed format. In compressed format, the valueID 37 will be written only once, followed by the remaining valueIDs for the other countries as before. The number of occurrences "1.4 billion" for China will be stored explicitly. Figure 7.1b depicts examples of the uncompressed and compressed attribute vectors.

The following calculation illustrates the compression rate. First of all the number of bits required to store all 200 countries is calculated as $log_2(200)$ which results in 8 bit.

Without compression the attribute vector stores the 8 bit for each valueID 8 billion times:

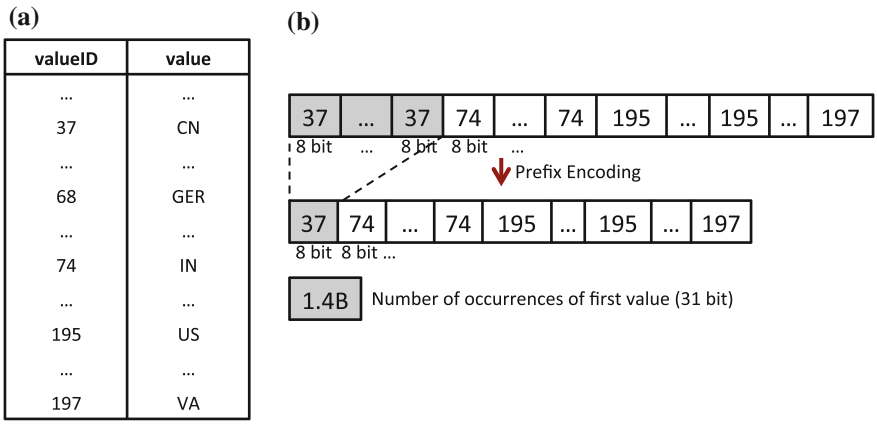

Fig. 7.1 Prefix encoding example. (**a**) Dictionary. (**b**) Dictionary-encoded attribute vector (*top*) and prefix-encoded dictionary-encoded attribute vector (*bottom*)

$$8\,\text{billion} \cdot 8\,\text{bit} = 8\,\text{billion}\,\text{Byte} = 7.45\,\text{GB}$$

If the country column is prefix-encoded, the valueID for China is stored only once in 8 bit instead of 1.4 billion times 8 bit. An additional 31 bit field is added to store the number of occurrences ($\lceil log_2\ (1.4\,\text{billion})\ \rceil = 31\,\text{bit}$). Consequently, instead of storing 1.4 billion times 8 bit, only $31\,\text{bit} + 8\,\text{bit} = 39\,\text{bit}$ are really necessary. The complete storage space for the compressed attribute vector is now:

$$(8\,\text{billion} - 1.4\,\text{billion}) \cdot 8\,\text{bit} + 31\,\text{bit} + 8\,\text{bit} = 6.15\,\text{GB}$$

Thus, 1.3 GB, i.e., 17 % of storage space is saved. Another advantage of prefix encoding is direct access with row number calculation. For example, to find all male Chinese the database engine can determine that only tuples with row numbers from 1 until 1.4 billion should be considered and then filtered by the gender value.

Although we see that we have reduced the required amount of main memory, it is evident that we still store much redundant information for all other countries. Therefore, we introduce run-length encoding in the next section.

7.2 Run-Length Encoding

Run-length encoding is a compression technique that works best if the attribute vector consists of few distinct values with a large number of occurrences. For maximum compression rates, the column needs to be sorted, so that all the same values are located together. In run-length encoding, value sequences with the same value are replaced with a single instance of the value and

(a) either its number of occurrences or
(b) its starting position as offsets.

Figure 7.2 provides an example of run-length encoding using the starting positions as offsets. Storing the starting position speeds up access. The address of a specific value can be read in the column directly instead of computing it from the beginning of the column, thus, providing direct access.

7.2.1 Example

Applied to our example of the country column sorted by population, instead of storing all 8 billion values (7.45 GB), we store two vectors:

- one with all distinct values: 200 times 8 bit

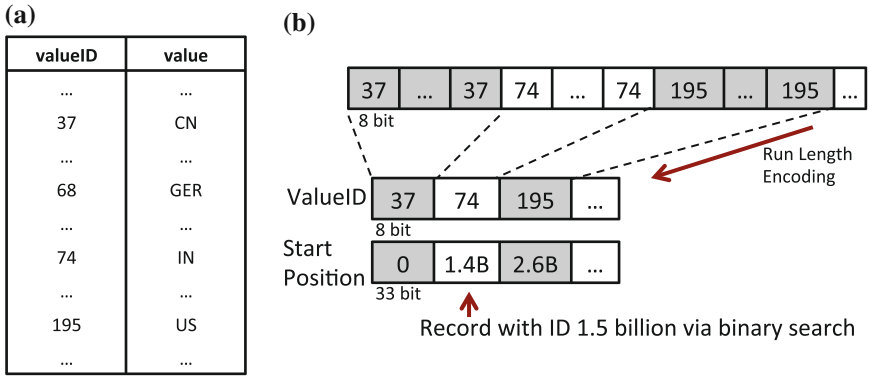

Fig. 7.2 Run-length encoding example. (**a**) Dictionary. (**b**) Dictionary-encoded attribute vector (*top*) and compressed dictionary-encoded attribute vector (*bottom*)

- the other with starting positions: 200 times 33 bit with 33 bit necessary to store the offsets up to 8 billion ($\lceil log_2\ (8\,\text{billion}) \rceil = 33\,\text{bit}$). An additional 33 bit field at the end of this vector stores the number of occurrences for the last value.

Hence, the size of the attribute vector can be significantly reduced to approximately 1 KB without any loss of information:

$$200 \cdot (33\,\text{bit} + 8\,\text{bit}) + 33\,\text{bit} \approx 1\,\text{KB}$$

If the number of occurrences is stored in the second vector, one field of 33 bit can be saved with the disadvantage of losing the direct access possibility via binary search. Losing direct access results in longer response times, which is no option for enterprise data management.

7.3 Cluster Encoding

Cluster encoding works on equal-sized blocks of a column. The attribute vector is partitioned into N blocks of fixed size (typically 1024 elements). If a cluster contains only a single value, it is replaced by a single occurrence of this value. Otherwise, the cluster remains uncompressed. An additional bit vector of length N indicates which blocks have been replaced by a single value (1 if replaced, 0 otherwise). For a given row, the index of the corresponding block is calculated by integer division of the row number and the block size N. Figure 7.3 depicts an example for cluster encoding with the uncompressed attribute vector on the top and the compressed attribute vector on the bottom. Here, the blocks only contain four elements for simplicity.

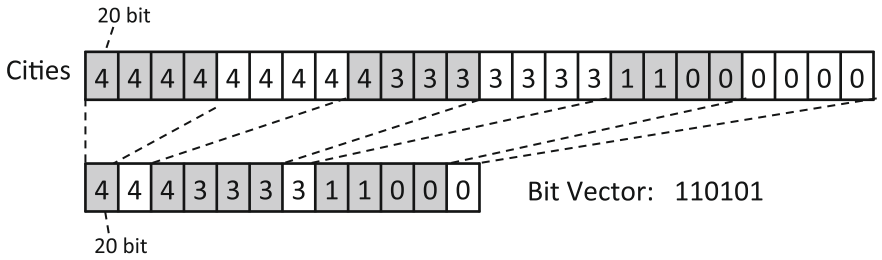

Fig. 7.3 Cluster encoding example

7.3.1 Example

Given is the city column (1 million different cities) from the world population table. The whole table is sorted by country and city. Hence, cities, which belong to the same country, are stored next to each other. Consequently, the occurrences of the same city values are stored next to each other, as well. 20 bit are needed to represent 1 million city valueIDs ($\lceil log_2\ (1\ \text{million})\ \rceil = 20$ bit). Without compression, the city attribute vector requires 18.6 GB (8 billion times 20 bit).

Now, we compute the size of the compressed attribute vector illustrated in Fig. 7.3. With a cluster size of 1024 elements the number of blocks is 7.8 million ($\frac{8\ billion\ rows}{1024\ elements\ per\ block}$). In the worst case every city has 1 incompressible block. Thus, the size of the compressed attribute vector is computed from the following sizes:

$$
\begin{aligned}
&\textit{incompressible blocks} + \textit{compressible blocks} + \text{bit } \textit{vector} \\
&= 1\,\text{million} \cdot 1024 \cdot 20\,\text{bit} + (7.8 - 1)\,\text{million} \cdot 20\,\text{bit} + 7.8\,\text{million} \cdot 1\,\text{bit} \\
&= 2441\,\text{MB} + 16\,\text{MB} + 1\,\text{MB} \\
&\approx 2.4\,\text{GB}
\end{aligned}
$$

With a resulting size of 2.4 GB, a compression rate of 87 % (16.2 GB less space required) can be achieved.

Cluster encoding does not support direct access to records. The position of a record needs to be computed via the bit vector. As an example, consider the query that counts how many men and women live in Berlin (for simplicity, we assume that only one city with the name "Berlin" exists and the table is sorted by city):

```
SELECT gender, COUNT(gender)
FROM world_population
WHERE city = 'Berlin'
GROUP BY gender;
```

To find the recordIDs for the result set, we look up the valueID for "Berlin" in the dictionary. In our example, illustrated in Fig. 7.4, this valueID is 3. Then, we scan the cluster-encoded city attribute vector for the first appearance of valueID 3.

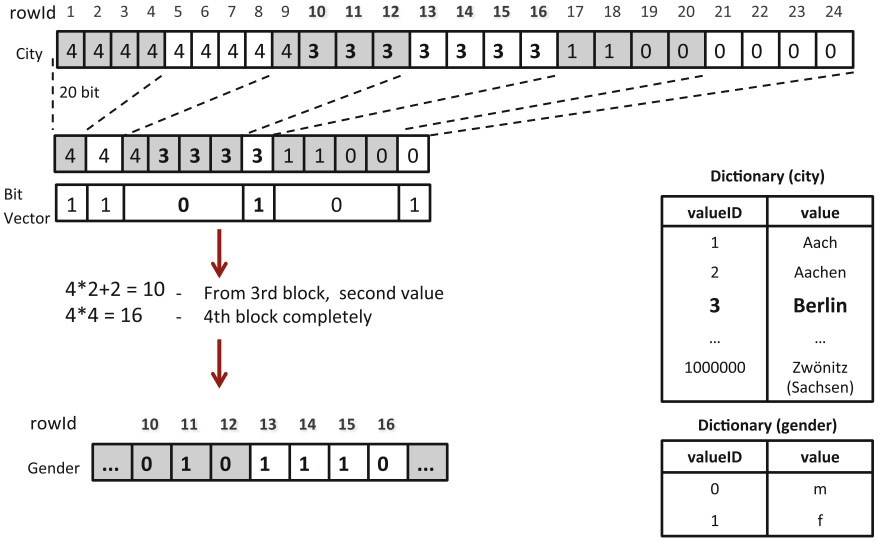

Fig. 7.4 Cluster encoding example: no direct access possible

While scanning the cluster-encoded vector, we need to maintain the corresponding position in the bit vector, as each position in the vector is mapped to either one value (if the cluster is compressed) or four values (if the cluster is uncompressed) of the cluster-encoded city attribute vector. In Fig. 7.4, this is illustrated by stretching the bit vector to the corresponding value or values of the cluster-encoded attribute vector. After the position is found, a bit vector lookup is needed to check whether the block(s) containing this valueID are compressed or not to determine the recordID range containing the value "Berlin". In our example, the first block containing "Berlin" is uncompressed and the second one is compressed. Thus, we need to analyze the first uncompressed block to find the first occurrence of valueID 3, which is the second position, and can calculate the range of recordIDs with valueID 3, in our example 10 to 16. Having determined the recordIDs that match the city predicate, we can use these recordID to access the corresponding gender records and aggregate according to the gender values.

7.4 Indirect Encoding

Similar to cluster encoding, indirect encoding operates on blocks of data with N elements (typically 1024). Indirect Encoding can be applied efficiently if data blocks hold a few numbers of distinct values. It is often the case if a table is sorted by another column and a correlation between these two columns exists (e.g., name column if table is sorted by countries).

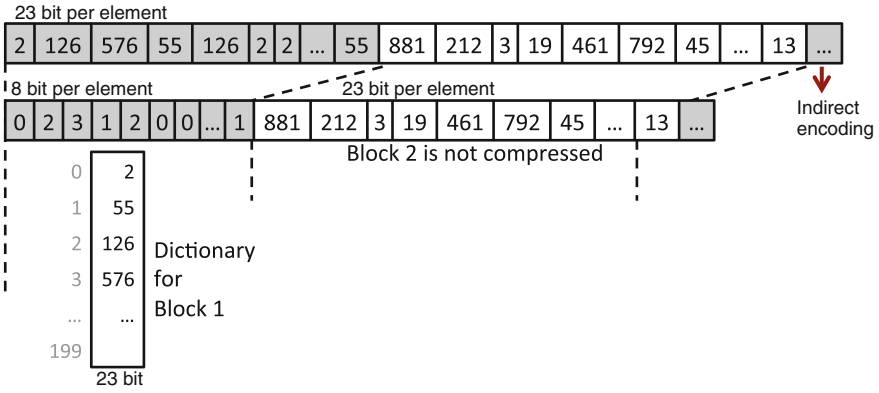

Fig. 7.5 Indirect encoding example

Besides a global dictionary used by dictionary encoding in general, additional local dictionaries are introduced for those blocks that contain only a few distinct values. A local dictionary for a block contains all (and only those) distinct values that appear in this specific block. Thus, mapping to even smaller valueIDs can save space. Direct access is still possible, however, an indirection is introduced because of the local dictionary. Figure 7.5 depicts an example for indirect encoding with a block size of 1024 elements. The upper part shows the dictionary-encoded attribute vector, the lower part shows the compressed vector. The first block contains only 200 distinct values and is compressed. The second block is not compressed.

7.4.1 Example

Given is the dictionary-encoded attribute vector for the first name column (5 million distinct values) of the world population table that is sorted by country. The number of bits required to store 5 million distinct values is 23 bit ($\lceil log_2\ (5\,\text{million})\ \rceil = 23\,\text{bit}$). Thus, the size of this vector without additional compression is 21.4 GB ($8\,\text{billion} \cdot 23\,\text{bit}$).

Now we split up the attribute vector into blocks of 1024 elements resulting in 7.8 million blocks ($\frac{8\,billion\,rows}{1024\,elements}$). For our calculation and for simplicity, we assume that each set of 1024 people of the same country contains on average 200 different first names and all blocks will be compressed. The number of bits required to represent 200 different values is 8 bit ($\lceil log_2(200) \rceil = 8\,\text{bit}$). As a result, the elements in the compressed attribute vector need only 8 bit instead of 23 bit when using local dictionaries.

Dictionary sizes can be calculated from the (average) number of distinct values in a block (200) multiplied by the size of the corresponding old valueID (23 bit) being the value in the local dictionary. For the reconstruction of a certain row, a pointer to the local dictionary for the corresponding block is stored (64 bit). Thus, the runtime for accessing a row is constant. The total amount of memory necessary for the compressed attribute vector is calculated as follows:

$$
\begin{aligned}
&\textit{local dictionaries} + \textit{compressed attribute vector} \\
&= (200 \cdot 23\,\text{bit} + 64\,\text{bit}) \cdot 7.8\,\text{million}\ \textit{blocks} + 8\,\text{billion} \cdot 8\,\text{bit} \\
&= 4.2\,\text{GB} + 7.6\,\text{GB} \\
&\approx 11.8\,\text{GB}
\end{aligned}
$$

Compared to the 21.4 GB for the dictionary-encoded attribute vector, a saving of 9.6 GB (44 %) can be achieved. The following example query that selects the birthdays of all people named "John" in the "USA" shows that indirect encoding allows for direct access:

```
SELECT birthday
FROM world_population
WHERE first_name = 'John' AND country = 'USA'
```

Listing 7.4.1: Birthdays for all residents of the USA with first name John

As the table is sorted by country, we can easily identify the recordIDs of the records with country="USA", and determine the corresponding blocks to scan the "first_name" column by dividing the first and last recordID by the cluster size.

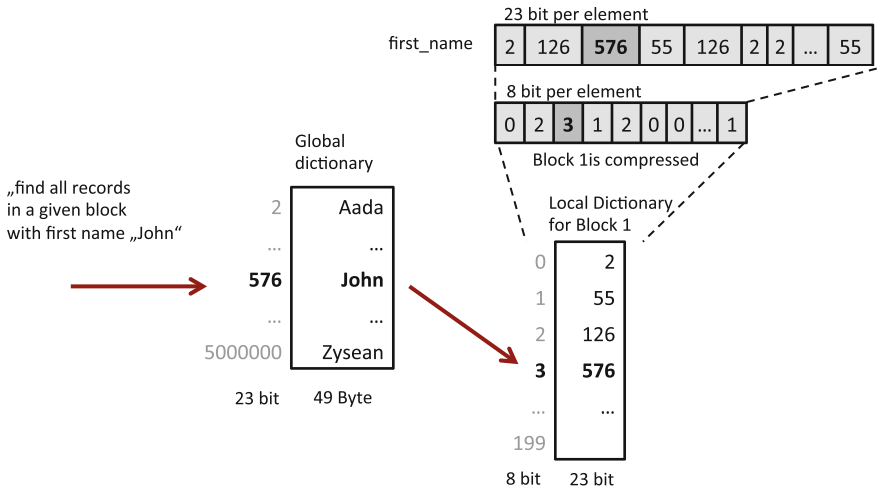

Fig. 7.6 Indirect encoding example: direct access

Then, the valueID for "John" is retrieved from the global dictionary and, for each block, the global valueID is translated into the local valueID by looking it up in the local dictionary. This is illustrated in Fig. 7.6 for a single block. Then, the block is scanned for the local valueID and corresponding recordIDs are returned for the birthday projection. In most cases, the starting and ending recordID will not match the beginning and the end of a block. In this case, we only consider the elements between the first above found recordID in the starting block up to the last found recordID for the value "USA" in the ending block.

7.5 Delta Encoding

The compression techniques covered so far reduce the size of the attribute vector. There are also some compression techniques to reduce the data amount in the dictionary as well. Let us assume that the data in the dictionary is sorted alpha-numerically and we often encounter a large number of values with the same prefixes. Delta encoding exploits this fact and stores common prefixes only once.

Delta encoding uses a block-wise compression like in previous sections with typically 16 strings per block. At the beginning of each block, the length of the first string, followed by the string itself, is stored. For each following value, the number of characters used from the previous prefix, the number of characters added to this prefix and the characters added are stored. Thus, each following string can be composed of the characters shared with the previous string and its remaining part. Figure 7.7 shows an example of a compressed dictionary. The dictionary itself is shown in Fig. 7.7a. Its compressed counterpart is provided in Fig. 7.7b.

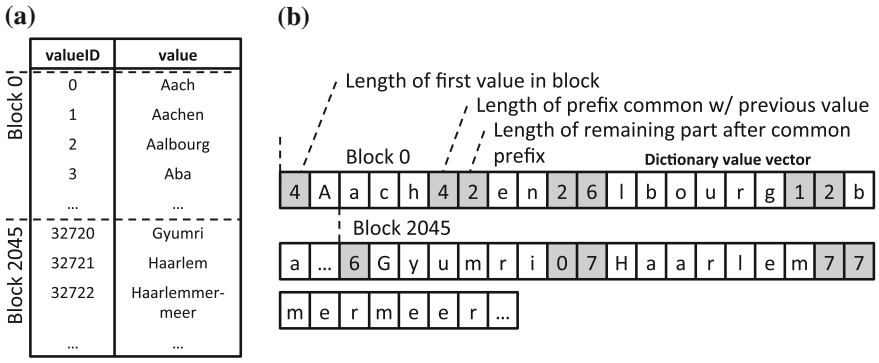

Fig. 7.7 Delta encoding example. **a** Dictionary. **b** Compressed dictionary

7.5.1 Example

Given is a dictionary for the city column sorted alpha-numerically. The size of the uncompressed dictionary with 1 million cities, each value using 49 Byte (we assume the longest city name has 49 letters), is 46.7 MB.

For compression purposes, the dictionary is separated into blocks of 16 values. Thus, the number of blocks is 62,500 ($\frac{1\,million\,cities}{16}$). Furthermore, we assume the following data characteristics to calculate the required size in memory:

- average length of city names is 7
- average overlap of 3 letters
- the longest city name is 49 letters ($\lceil log_2(49) \rceil = 6\,\text{bit}$).

The size of the compressed dictionary is now calculated as follows:

$$
\begin{aligned}
&\textit{block size} \cdot \textit{number of blocks} \\
&= \textit{encoding lengths} + 1\textit{st city} + 15\ \textit{other cities} \cdot \textit{number of blocks} \\
&= ((1 + 15 \cdot 2) \cdot 6\,\text{bit} + 7 \cdot 1\,\text{Byte} + 15 \cdot (7 - 3) \cdot 1\,\text{Byte}) \cdot 62,500 \\
&\approx 5.4\,\text{MB}
\end{aligned}
$$

Compared to the 46.7 MB without compression the saving is 42.2 MB (90 %).

7.6 Limitations

What has to be kept in mind is that most compression techniques require sorted sets to tap their full potential, but a database table can only be sorted by one column or cascading. Furthermore, some compression techniques do not allow direct access. This has to be carefully considered with regard to response time requirements of queries.

7.7 Self Test Questions

1. **Sorting Compressed Tables**
   Which of the following statements is correct?
   (a) If you sort a table by the amount of data for a row, you achieve faster read access
   (b) Sorting has no effect on possible compression algorithms
   (c) You can sort a table by multiple columns at the same time
   (d) You can sort a table only by one column.

2. **Compression and OLAP / OLTP**
What do you have to keep in mind if you want to bring OLAP and OLTP together?

   (a) You should not use any compression techniques because they increase CPU load
   (b) You should not use compression techniques with direct access, because they cause major security concerns
   (c) Legal issues may prohibit to bring certain OLTP and OLAP datasets together, so all entries have to be reviewed
   (d) You should use compression techniques that give you direct positional access, since indirect access is too slow.

3. **Compression Techniques for Dictionaries**
Which of the following compression techniques can be used to decrease the size of a sorted dictionary?

   (a) Cluster Encoding
   (b) Prefix Encoding
   (c) Run-Length Encoding
   (d) Delta Encoding.

4. **Indirect Access Compression Techniques**
Which of the explained compression techniques does not support direct access?

   (a) Run-Length Encoding
   (b) Prefix Encoding
   (c) Cluster Encoding
   (d) Indirect Encoding.

5. **Compression Example Prefix Encoding**
Suppose there is a table where all 80 million inhabitants of Germany are assigned to their cities. Germany consists of about 12,200 cities, so the valueID is represented in the dictionary via 14 bit. The outcome of this is that the attribute vector for the cities has a size of 140 MB. We compress this attribute vector with Prefix Encoding and use Berlin, which has nearly 4 million inhabitants, as the prefix value. What is the size of the compressed attribute vector? Assume that the needed space to store the amount of prefix values and the prefix value itself is neglectable, because the prefix value only consumes 22 bit to represent the number of citizens in Berlin and additional 14 bit to store the key for Berlin once. Further assume the following conversions: $1\,\text{MB} = 1000\,\text{kB},\ 1\,\text{kB} = 1000\,\text{B}$

   (a) 0.1 MB
   (b) 133 MB
   (c) 63 MB
   (d) 90 MB

6. **Compression Example Run-Length Encoding Germany**
Suppose there is a table where all 80 million inhabitants of Germany are assigned to their cities. The table is sorted by city. Germany consists of about 12,200 cities (represented by 14 bit). Using Run-Length Encoding with a start position vector, what is the size of the compressed city vector? Always use the minimal number of bits required for any of the values you have to choose. Further assume the following conversions: $1\,\text{MB} = 1000\,\text{kB},\ 1\,\text{kB} = 1000\,\text{B}$

   (a) 1.2 MB
   (b) 127 MB
   (c) 5.2 KB
   (d) 62.5 kB

7. **Compression Example Cluster Encoding**
Assume the world population table with 8 billion entries. This table is sorted by countries. There are about 200 countries in the world. What is the size of the attribute vector for countries if you use Cluster Encoding with 1,024 elements per block assuming one block per country can not be compressed? Use the minimum required count of bits for the values. Further assume the following conversions: $1\,\text{MB} = 1000\,\text{kB},\ 1\,\text{kB} = 1000\,\text{B}$

   (a) $\approx 9\,\text{MB}$
   (b) $\approx 4\,\text{MB}$
   (c) $\approx 0.5\,\text{MB}$
   (d) $\approx 110\,\text{MB}$

8. **Best Compression Technique for Example Table**
Find the best compression technique for the name column in the following table. The table lists the names of all inhabitants of Germany and their cities, i.e. there are two columns: first_name and city. Germany has about 80 million inhabitants and 12,200 cities. The table is sorted by the city column. Assume that any subset of 1,024 citizens contains at most 200 different first names.

   (a) Run-Length Encoding
   (b) Indirect Encoding
   (c) Prefix Encoding
   (d) Cluster Encoding.

Reference

[AMF06] D. Abadi, S. Madden, M. Ferreira, Integrating compression and execution in column-oriented database systems, in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD '06* (ACM, New York, 2006), pp. 671–682