# Indexing and Compression

When storing large amounts of data, such as indexed web pages for search engines, compressing the data and accessing it efficiently on disk is often more important than sheer runtime complexity. This section considers two simple examples that minimize the amount of storage required for certain types of data, while maintaining efficient access times.

## Variable Length Encoding

Suppose you have a collection of 50,000,000 positive integers to store on disk and you can guarantee that every integer will fit into a 32-bit `int` variable. A naïve solution would be to store 50,000,000 32-bit integers on disk, for a total of 200,000,000 bytes. We seek a better alternative that would use significantly less disk space. (One reason for compressing data on disk might be to load it more quickly into memory.)

*Variable length encoding* is a compression technique suitable for number sequences that contain many small values. Before we consider how it works, we need to guarantee that there *will* be many small values in the sequence—which currently does not seem to be the case. If the 50,000,000 integers are uniformly distributed in the range [0, $2^{32}$], then more than 99% will not fit in 3 bytes and require a full 4 bytes of storage. However, we can sort the numbers prior to storing them on disk, and, instead of storing the numbers, store the gaps. This trick is called *gap compression* and likely makes the numbers much smaller, while still allowing us to reconstruct the original data.

For example, the series (38, 14, 77, 5, 90) is first sorted to (5, 14, 38, 77, 90) and then encoded using gap compression to (5, 9, 24, 39, 13). Note that the numbers are much smaller when using gap compression, and the average number of bits required to store them has gone down significantly. In our case, if the 50,000,000 integers were uniformly distributed in the range [0, $2^{32}$], then many gaps would very likely fit in a single byte, which can contain values in the range [0, 256].

Next, we turn to the heart of *variable byte length encoding*, which is just one of a large number of methods in information theory that can compress data. The idea is to use the most significant bit of every byte to indicate whether it is the last byte that encodes a single integer or not. If the bit is off, go on to the next byte to reconstruct the number; if the bit is on, stop and use the bytes read so far to decode the value.

For example, the number 13 is encoded as `10001101`—the high bit is on, so this byte contains an entire integer and the rest of it is simply the number 13 in binary. Next, the number 132 is encoded as ``00000001`10000100``. The first byte has its high bit off, so remember the seven bits `0000001`, and the second byte has its high bit on, so append the seven bits `0000000` and obtain `10000100`, which is the number 132 in binary. In this example, one of the numbers was stored using just 1 byte, and the other using 2 bytes. Storing the gaps obtained in the previous step using this technique is likely to compress the original data almost four-fold. (You can experiment with randomly generated integers to establish this result.)

## Index Compression

To store an index of words that appear on web pages in an efficient way—which is the foundation of a crude search engine—we need to store, for each word, the page numbers (or URLs) in which it appears, compressing the data, while maintaining efficient access to it. In typical settings, the page numbers in which a word appears do not fit in main memory, but the dictionary of words just might.

Storing on disk the page numbers in which dictionary words appear is a task best left to variable length encoding—which we just considered. However, storing the dictionary itself is somewhat more complex. Ideally, the dictionary is a simple array of entries that contain the word itself and a disk offset to the page numbers in which the words appears. To enable efficient access to this data, it should be sorted—this guarantees $O(\log n)$ access times.

Suppose each entry is the in-memory representation of the following C# value type, and the entire dictionary is an array of them:

```
struct DictionaryEntry {
  public string Word;
  public ulong DiskOffset;
}
DictionaryEntry[] dictionary = . . .;
```

As Chapter 3 illustrates, an array of value types consists solely of the value type instances. However, each value type contains a reference to a string; for *n* entries, these references, along with the disk offsets, occupy 16*n* bytes on a 64-bit system. Additionally, the dictionary words themselves take up valuable space, and each dictionary word—stored as a separate string— has an extra 24 bytes of overhead (16 bytes of object overhead + 4 bytes to store the string length + 4 bytes to store the number of characters in the character buffer used internally by the string).

We can considerably decrease the amount of space required to store the dictionary entries by concatenating all dictionary words to a single long string and storing offsets into the string in the `DictionaryEntry` structure (see Figure 9-5). These concatenated dictionary strings are rarely longer than $2^{24}$ bytes = 16MB, meaning the index field can be a 3-byte integer instead of an 8-byte memory address:

```
[StructLayout(LayoutKind.Sequential, Pack = 1, Size = 3)]
struct ThreeByteInteger {
  private byte a, b, c;
  public ThreeByteInteger() {}
  public ThreeByteInteger(uint integer) . . .
  public static implicit operator int(ThreeByteInteger tbi) . . .
}
struct DictionaryEntry {
  public ThreeByteInteger LongStringOffset;
  public ulong    DiskOffset;
}
class Dictionary {
  public DictionaryEntry[] Entries = . . .;
  public string    LongString;
}
```

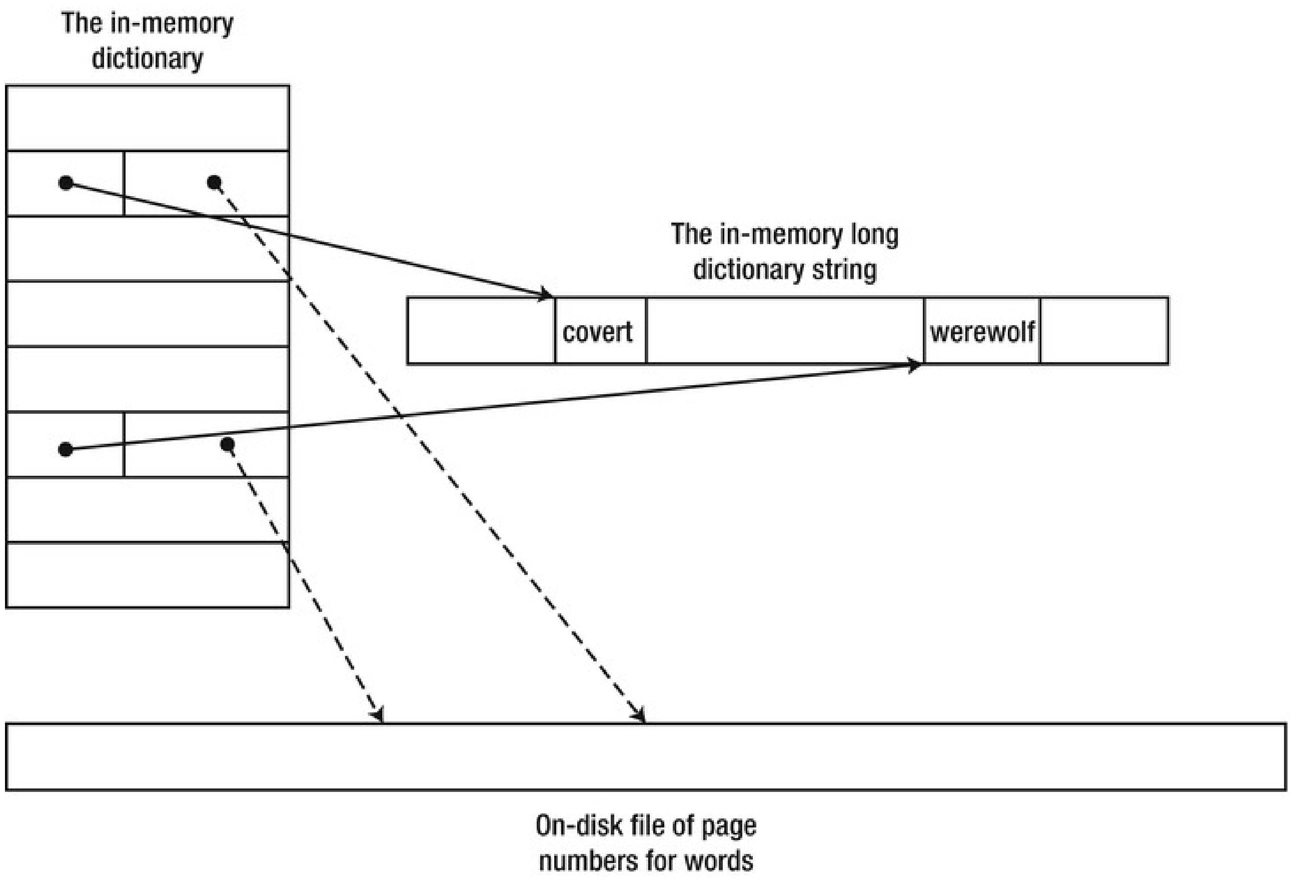
The in-memory dictionary
The in-memory long dictionary string
covert
werewolf
On-disk file of page numbers for words

*Figure 9-5 . General structure of the in-memory dictionary and string directory*

The result is that we maintain binary search semantics over the array—because the entries have uniform size—but the amount of data stored is significantly smaller. We saved almost 24*n* bytes for the string objects in memory (there is now just one long string) and another 5*n* bytes for the string references replaced by offset pointers.