Optimal Partitioning of Data Chunks in Deduplication Systems*

M. Hirsch[a], A. Ish-Shalom[a], S.T. Klein[b]

[a] *IBM – Diligent, Tel Aviv, Israel*
{hirschm,arielish}@il.ibm.com

[b] *Department of Computer Science, Bar Ilan University, Ramat Gan, Israel*
tomi@cs.biu.ac.il

Abstract

Deduplication is a special case of data compression in which repeated chunks of data are stored only once. For very large chunks, this process may be applied even if the chunks are similar and not necessarily identical, and then the encoding of duplicate data consists of a sequence of pointers to matching parts. However, not all the pointers are worth being kept, as they incur some storage overhead. A linear, sub-optimal solution of this partition problem is presented, followed by an optimal solution with cubic time complexity and requiring quadratic space.

Keywords: Deduplication, partitioning data chunks, dynamic programming

1. Introduction and Background

Lossless Data compression deals with techniques to recode large amounts of digital information into a more compact form, under the constraint that the original may be restored without alterations. The compression gain depends on the compressibility of the data at hand, and some files, like pseudo-random sequences, may not be compressed at all. A special case of input data are large backup and storage systems, which need to process ever increasing amounts of information, and standard lossless data compression methods may not be able to cope with it. On the other hand, the use of classical compression may be an overkill, since backup data has generally

*This is an extended version of a paper that has been presented at the Prague Stringology Conference (PSC'13) in 2013, and appeared in its Proceedings, 128–141.

the property that only a small fraction of it is changed between consecutive backup generations. The data is therefore highly repetitive, which calls for a special form of data compression, known as *deduplication*: trying to store duplicates only once. The challenge is, of course, to locate as much of the duplicated data as possible.

A standard deduplication system achieves its goal in the following way. Partition the input database, which is often called the *repository*, into fixed or variable sized blocks, called *chunks*, apply a cryptographically strong hash function on each of these input chunks, and store the different hash values, along with the address of the corresponding chunk, in a fast to access data structure, like a hash table or a B-Tree [6, 7]. When a fresh copy of the data is given, e.g., for a weekly or even daily backup, the new data, often called a *version*, is also partitioned into similar chunks, and a chunk is only kept if the corresponding hash value is not stored yet. Otherwise it is replaced by a pointer to the already stored copy.

A major dilemma is to decide what the (average) chunk size should be, as if it is too small, the number of chunks and the accompanying overhead might be too large; on the other hand, the larger the chunks, the lower is the probability of finding identical ones, reducing the potential deduplication benefits. Note that systems based on using hashing functions are generally only able to detect *identical* chunks, because most hashing functions are designed with the specific aim that even small changes in the argument should imply substantive changes in the hashed values. This lead to the idea of devising deduplication systems based on *similarity* rather than identity, thereby allowing the use of considerably larger chunks, as in the IBM ProtecTIER product, described in [1]. An extension of this similarity based deduplication system to an environment using small sized chunks appears in [2]. We focus here on systems using very large chunks, and shall deal with the following problem implied by it.

While a single pointer is sufficient for the compression of an identical chunk, the case of similar chunks is more involved. Similarity might imply that most of the data of the version chunk can be copied from the repository, but the data to be copied is not necessarily contiguous and might appear in various chunks; moreover, even if several pointers refer to the same repository chunk, they could point to locations that are scattered throughout it. Consider, for example, a repository including Lewis Carroll's famous quote:

> "Begin-at$\boxed{\text{-}}$the-begin$\boxed{\text{n}}$ing,"-the$\boxed{\text{-}}$King-said$\boxed{\text{,}}$-very-gra$\boxed{\text{v}}$ely,"-and$\boxed{\text{-}}$go-on-til$\boxed{\text{l}}$-you-come$\boxed{\text{-}}$to-the-en$\boxed{\text{d}}$:-then-st$\boxed{\text{o}}$p."

where spaces have been visualized by dashes, and every tenth character is boxed to facilitate their enumeration. Imagine now a new version of a similar phrase:

> "Start-at-the-beginning,"-said-the-King,-very-gravely,-"and-continue-until-you-reach-the-end:-then-finish."

The phrase can be encoded as a sequence of characters and (address, length) pairs as in Ziv and Lempel's compression scheme known as LZ77 [8], where address refers to the starting location of a matching string from the version within the repository, and length is the number of its characters. In our example, the encoded form would be:

> "Start (7,19) (35,5) (26,9) (40,21) continue-un (67,3) (71,5) reach (83,15) finish (102,2)

The encoding of a compressed chunk will thus be a sequence of various copy items, interspersed with stretches of new data. If one considers quite long chunks, say, of the order of 16MB, and adds to this the fact that the new data can be as short as a single byte, the conclusion is that the number of elements in the encoding of a single chunk may be large.

This situation is aggravated in a typical scenario of a backup system, which stores several consecutive *generations* of almost the same data. There might only be minor changes between adjacent generations, but these changes have a cumulative effect, leading to chunks that are increasingly fragmented into smaller and smaller copy and non-copy items. However, storing the data needed to reconstruct a highly fragmented chunk may itself create a compression problem.

In the next section, we define the specific problem dealt with herein, namely finding an optimal partition of a chunk into matching and non-matching parts. Section 3 then suggests a sub-optimal, yet linear, algorithm, and Section 4 an optimal one, requiring cubic time. Section 5 brings a few improvements.

Papers presenting new compression schemes usually contain experimental sections reporting on tests of the suggested algorithms. But while there are well established test cases which have been agreed upon in the compression community, like the Calgary or the Canterbury [3] corpora, there is no equivalent for deduplication tests. The reason is mainly that the performance does not depend on the nature of the files, but rather on the their repetitiveness. Thus even a file containing random data, which cannot be compressed, may still profit from deduplication if it appears more than once in the repository.

There is therefore no possibility to find data that could be deemed to be representative, which is why we have preferred to leave this article on the theoretic level, suggesting only a theoretical framework, with no experimental section. Even if we had performed some tests, the experimental results could be presented as examples only, without claiming that one could extrapolate from them information on the performance in general.

2. Definition of the problem

We assume that given a new version, the deduplication system has already located all its matching parts in the repository, and we consider now applying a filtering stage which should eliminate those parts of the compressed data that will ultimately not be worth being kept, because the required overhead might be larger than the compression gain. The input to this part of the process is a chunk of data and a *list of matches*, each consisting of a pair of pointers, one to the given version chunk, one into the repository, and the size of the matching substring. The expected output is a partition of the given chunk into a sequence of mismatching and matching blocks. The compressed form of the chunk will then consist of a copy of the mismatching parts, and of pointers describing where the matching parts can be found.

A simplistic solution would of course be to build the output by just copying the input, that is, accept exactly the partition found by listing all the matches. But this would ignore the fact that at least a part of the matches are not worth being kept, as they might cause a too high degree of fragmentation. Referring to the example quote in the introduction, the starting quotation marks " of the version could have been substituted by the pair (1,1), pointing to their counterpart at the beginning of the repository, but this is obviously more expensive than repeating the " itself; similarly, the (102,2) pointer at the end refers to the string ." (dot, quotation mark), and might better be omitted. The challenge is therefore to decide which matches should be kept, and which should be ignored.

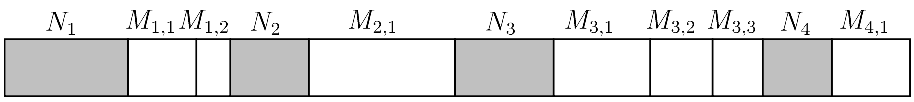

Figure 1: *Schematic representation of the partition of a data chunk*

Figure 1 shows a possible partition of a data chunk into alternating areas of non-matches and matches. The non-matches, represented by the grey rectangles, contain **N**ew data and are indexed $N_1, N_2, \ldots, N_k$. The **M**atches, drawn as the white rectangles, contain data that has previously appeared in the repository, and will be stored by means of pointers of the form ($\mathsf{address, length}$); the matching parts between the non-matching blocks N_i and N_{i+1} are indexed $M_{i,1}, M_{i,2}, \ldots M_{i,j_i}$. Non-matching parts cannot be consecutive — this is new data, and any stretch of such new characters is considered a single new part. The matching parts, on the other hand, may consist of several different sub-parts that are located in different places on the disk; each sub-part needs therefore a pointer of its own.

We consider two functions defined on these matching and non-matching parts. A cost function $c()$ giving the price we incur for storing the pointers in the meta-data; typically, but not necessarily, all pointers are of fixed length E (in our implementation, $E = 24$ bytes), that is $c(N_i) = c(M_{\ell,j}) = 24$ for all indexes, so that actually, the cost for the meta-data depends only on the number of parts, which is $k + \sum_{t=1}^{k} j_t$. In other implementations, the pointers may undergo another layer of compression, e.g., Huffman coding, resulting in variable length elements.

The second function $s()$ measures, for each part, the size of the data on the disk. So we have that $s(N_i)$ will be just the number of bytes of the non-matching part, as these new bytes have to be stored physically somewhere, and $s(M_{\ell,j}) = 0$, since no new data is written to the disk for a matching part. However, we shall define $s(M_{\ell,j}) = \mathsf{length}$ for a block $M_{\ell,j}$ that is stored by means of a pointer ($\mathsf{address, length}$), which means that the size will be defined as the number of bytes written to the disk in case we decide to ignore the fact that $M_{\ell,j}$ has occurred earlier and thus has a matching part already in the repository.

The compressed data consists of the items written to the disk plus the pointers in the meta-data, but these cannot necessarily be traded one to one, as storage space for the meta-data will generally be more expensive. We shall assume that there exists a multiplicative factor F such that, in our calculations, we can count one byte of meta-data as equivalent to F bytes of data written to the disk. This factor need not be constant and may dynamically depend on several run-time parameters. Practically, F will be stored in a variable and may be updated when necessary, but we shall use it in the sequel as if it were a constant.

Given the above notations, the size of the compressed file is then

$$F \cdot \left[\sum_{i=1}^{k} \left(c(N_i) + \sum_{t=1}^{j_i} c(M_{i,t}) \right) \right] + \sum_{i=1}^{k} s(N_i),$$

and in the particular case of fixed length pointers of size E, which we shall assume below, for simplicity:

$$F \cdot E \cdot \left(k + \sum_{t=1}^{k} j_t \right) + \sum_{i=1}^{k} s(N_i), \tag{1}$$

whereas the uncompressed file has size

$$\sum_{i=1}^{k} \left(s(N_i) + \sum_{t=1}^{j_i} s(M_{i,t}) \right).$$

The optimization problem we consider is based on the fact that the partition we obtain as input may be altered. The non-matching parts N_i can obviously not be touched, so the only degree of freedom we have is to decide, for each of the matching parts $M_{i,j}$, whether the corresponding pointer should be kept, or whether we opt to ignore the match and treat this part as if it were non-matching. There is a priori nothing to be gained from such a decision: the pointer in the meta-data is changed from matching to non-matching, but incurs the same cost, and some data has been added to the disk, so there will always be a loss.

The following example shows that nevertheless, there can also be a gain in certain cases. Consider the block $M_{1,2}$ in Figure 1. If we decide to ignore its matching counterpart, the data of $M_{1,2}$ has to be written to the disk, but it is contiguous with the data of N_2. The two parts may therefore be fusioned, which reduces the number of meta-data entries by one. This will result in a gain if

$$s(M_{1,2}) < F \cdot E.$$

Moreover, if indeed we decide to consider $M_{1,2}$ as a non-matching block, this will leave $M_{1,1}$ as a single match between two non-matches. In this case, ignoring the match may allow to unify the three blocks $N_1, M_{1,1}, N_2$, reducing the number of meta-data entries by two. This will be worthwhile even if

$$s(M_{1,1}) < 2\ F \cdot E.$$

More generally, any extremal matching blocks (those touching on at least one of their sides with a non-match) may be candidates for such a fusion, which can trigger even further unifications like in the example. But these are not the only cases: even non-extremal blocks may profit from unification. This is not true for a single matching blocks, whose both neighbors are also matching, like $M_{3,2}$ in Figure 1, because we add data to the disk, but do not remove any meta-data, just change one of the entries. But there might be a stretch of several (at least two) matching blocks that can profit from unification.

It should be noted that devising a new partition is not only a matter of trading a byte of meta-data versus F bytes of disk data. Reducing the number of entries in the meta-data has also an effect on the time complexity, since each entry requires an additional read operation. Many compression algorithms have to deal with such time/space tradeoffs, and for our purpose, we shall assume that the factor F already takes also the time complexity into account, that is, F reflects our estimation of how many bytes of disk space we are ready to pay in order to save one byte of meta-data, considering all aspects, including space, CPU and I/O.

The challenge is therefore to come up with an efficient, and if possible, optimal way to select an appropriate subset of the input partition which minimizes the size of the compressed file as measured by equation (1).

3. Linear sub-optimal algorithm

The following algorithm is a first solution attempt. The partition it produces is not necessarily optimal, but the complexity is linear in the number of elements N_i and $M_{i,j}$. The algorithm uses as main data structure a doubly linked list L, the elements of which represent the matching or non-matching data blocks defined above, so their initial number is $n = k + \sum_{t=1}^{k} j_t$. Each element p of the list L has the following fields:

- **status(p)** – indicating whether the element p is pointing to is matching (M), non-matching (NM), or a sentinel element (S) for smoother programming
- **prev(p)** – pointing to the predecessor of p
- **succ(p)** – pointing to the successor of p
- **size(p)** – if status(p) = NM, this is the number of non-matching bytes; if status(p) = M, this is the length of the element to be copied; if status(p) = S, size(p) is not defined.
- **data(p)** – defined only if status(p) = NM, in which case it contains the new data not found in the repository; if status(p) = M, nothing will be stored in data(p), but we shall refer by DATA(p) to the bytes pointed to by the (address, length) pointer.

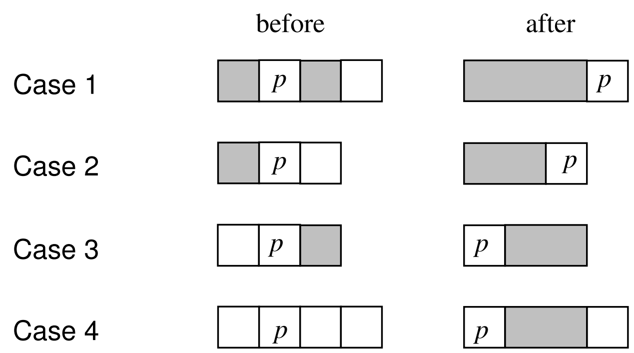

FIGURE 2: *Different cases dealt by the algorithm*

We first add sentinel elements, TOP and REAR at the beginning and end of the list, respectively, which avoids the necessity to check at each step whether successors and predecessors exist. The main idea is then to scan the list of items with a pointer p and perform local substitutions according to the contexts, if possible. If the current item is of type NM, it is skipped. If it is a matching item, we consider 5 disjoint cases.

1. Case 1: The item pointed to by p is surrounded by NM items. In this case, all 3 elements can be merged into one, if appropriate, that is, if **size(**p**)**$< 2F\,E$.
2. Case 2: The item pointed to by p is preceded by an NM item (and since it is a disjoint case from Case 1, there is no need to check that the element is followed by an M item); it can then be merged into the preceding item, if appropriate. Note that if several consecutive items can be merged, this is dealt with in the following iterations.
3. Case 3: The item pointed to by p is followed by an NM item (and therefore preceded by an M item); this case is symmetric to Case 2 .
4. Case 4: The item pointed to by p is surrounded by M items. We then check whether two M items can be merged into one NM item. Longer chains of M items are considered in the following iterations, though then in Case 3.
5. Case 5: No substitution is possible, just advance p to its successor.

The four first cases are schematically represented in Figure 2, where as before, NM items appear in grey and M items in white. As part of the actions to be performed in each case, the pointer p has to be repositioned. In the first 2 cases, p will point to the item following the newly merged block,

```
p ⟵ succ(TOP)
while succ(p) ≠ REAR
    if status(p) ≠ M then
        p ⟵ succ(p)
    else

        if status(prev(p)) = NM and status(succ(p)) = NM
                and size(p) < 2 F E then
                    // Case 1
            q ⟵ prev(p)
            size(q) ⟵ size(q) + size(p) + size(succ(p))
            data(q) ⟵ data(q) ‖ DATA(p) ‖ data(succ(p))
            q ⟵ succ(succ(p))
            delete (p) and succ(p) from L
            p ⟵ q

        else if status(prev(p)) = NM and size(p) < F E then
                    // Case 2
            q ⟵ prev(p)
            size(q) ⟵ size(q) + size(p)
            data(q) ⟵ data(q) ‖ DATA(p)
            q ⟵ succ(p)
            delete (p) from L
            p ⟵ q

        else if status(succ(p)) = NM and size(p) < F E then
                    // Case 3
            q ⟵ succ(p)
            size(q) ⟵ size(q) + size(p)
            data(q) ⟵ DATA(p) ‖ data(q)
            q ⟵ prev(p)
            delete (p) from L
            p ⟵ q

        else if status(prev(p)) ≠ NM and status(succ(succ(p))) ≠ NM
                and size(p) + size(succ(p)) < F E then
                    // Case 4
            status(p) ⟵ NM
            size(p) ⟵ size(p) + size(succ(p))
            data(p) ⟵ DATA(p) ‖ data(succ(p))
            delete succ(p) from L
            p ⟵ prev(p)

        else
            p ⟵ succ(p)
```

FIGURE 3: *Linear sub-optimal algorithm*

so the next iteration will take us to Case 2, and in the last 2 cases, p will point to the item preceding the newly merged block, so the next iteration will take us to Case 3.

It therefore follows that the main pointer of the procedure may also move backwards, which could result in an unbounded number of iterations. But in each iteration, either the pointer is advanced by one step, or the overall number of items is reduced by one, which bounds the global complexity to be at most $2n$ iterations, each requiring $O(1)$ commands. Note, however, that this solution is not necessarily optimal, as sequences of consecutive blocks are substituted greedily by pairs. It may happen that 3 consecutive M items could be merged, but considered as two pairs, none of them will result in a substitution. The formal algorithm is given in Figure 3. The operator $\|$ denotes concatenation.

At the end, the linked list contains all the necessary information on the partition. In particular, the original data can be reconstructed by the following sequential scan:

```
p ⟵ succ(TOP)
while (p) ≠ REAR
    if status(p) = NM then output data(p)
    else                   output DATA(p)
```

4. Optimal solution of the partition problem

We now turn to an optimal solution of the partition problem. The solution will be applied individually on each sequence of consecutive M-items, surrounded on both ends by NM-items, since these cannot be altered, and the only possible transformation is to declare matching blocks as if they were non-matching. Therefore the originally given NM-items will appear also in the final optimal solution, so we can concentrate on each sub-part on its own. Consider then the (matching) elements as indexed $1, 2, \ldots, n$, and the non-matching delimiters as indexed 0 and $n + 1$.

Notation: we shall return the required partition in the form of a bit-string of length n, with the bit in position i being set to 1 if the i-th element should be of type NM, and set to 0 if the i-th element should be of type M. This notation implies immediately that the number of possible solutions is 2^n, so that an exhaustive search of this exponential number of alternatives is ruled out.

The basis for a non-exponential solution is the fact that the optimal partition can be split into sub-parts, each of which has to be optimal for the corresponding subranges. We can thus get the solution for a given range by trying all the possible splits into, say, two sub-parts. Such recursive definitions call for resolving them by means of dynamic programming [4]. The tricky part here is that the optimal solution for the range (i, j), might depend on whether its bordering elements, indexed $i - 1$ and $j + 1$, are of type matching or non-matching, so the optimal solution for range (i, j) might depend on the optimal solution on the neighboring ranges.

The optimal partition will thus be built by means of a two-dimensional dynamic programming table $C[i, j]$, and the optimal partition will be stored in a similar table PS, so that $PS[i, j]$ holds the optimal partition for the given parameters, which is a bit-string of length $j - i + 1$. For $1 \leq i \leq j \leq n$, we define $C[i, j]$ as the global cost of the optimal partition of the sub-sequence of elements $i, i+1, \ldots, j-1, j$, when the surrounding elements $i - 1$ and $j + 1$ are of type NM. This cost will be given in bytes and reflects the size of the data on disk for NM-items, plus the size of the meta-data for all the elements, using the equivalence factor explained above, that is, each meta-data entry incurs a cost of FE bytes. Once the table is filled up, the cost of the optimal solution we seek is stored in $C[1, n]$ and the corresponding partition is given in $PS[1, n]$.

The basis of the calculation will be the individual items themselves stored in the main diagonal of the matrix, $C[i, i]$ for $1 \leq i \leq n$, as well as the elements just below the diagonal, $C[i, i - 1]$. The following iterations will then be ordered by increasing difference between i and j. We shall thus first deal with all sequences of two adjacent elements, then 3, etc. When calculating the optimal solution for a sequence of ℓ adjacent elements, we can use our knowledge of the optimal solutions for all shorter sub-sequences. If fact, for a sequence of length $\ell = j - i + 1$, we only need to check the sum of the costs of all possible partitions of this range into two subranges, that is the cost for $(i, k - 1)$ plus that of $(k + 1, j)$ for $i < k < j$. We initialize the cost for each subrange by the possibility of leaving all the n elements of type matching.

More specifically, the formal algorithm is given in Figure 4 and the line numbers below refer to this figure. Lines 1 and 3 initialize the table for ranges of size 0, that is, of type $[i + 1, i]$, giving them a cost 0. The corresponding bit-strings are Λ, which denotes the empty string. Lines 4–7 deal with singletons of type $[i, i]$. Since we assume that the surrounding elements are both of type NM, we have to compare the size $s(i)$ of the matching element with the cost of defining it as non-matching, and letting it be absorbed by the neighboring NM items. In that case, two elements of the meta-data can be saved, which is checked in line 4.

```
C[n+1, n] ⟵ 0    PS[n+1, n] ⟵ Λ
for i ⟵ 1 to n
   C[i, i−1] ⟵ 0    PS[i, i−1] ⟵ Λ
   if s(i) − FE < FE then
      C[i, i] ⟵ s(i) − FE    PS[i, i] ⟵ '1'
   else
      C[i, i] ⟵ FE    PS[i, i] ⟵ '0'
end for i

for diff ⟵ 1 to n − 1
   for i ⟵ 1 to n − diff
      j ⟵ i + diff
      C[i, j] ⟵ (diff + 1)FE
      PS[i, j] ⟵ '000···0'    //(length diff + 1)

      OK ⟵ 0
      for k ⟵ i to j
         if k = j then L ⟵ 1 else L ⟵ left(PS[k+1, j])
         if k = i then R ⟵ 1 else R ⟵ right(PS[i, k−1])

         newcost ⟵ C[i, k−1] + C[k+1, j] + s(k) + (1 − L − R)FE
         if newcost < C[i, j]
            C[i, j] ⟵ newcost
            OK ⟵ k
      end for k

      if OK > 0 then
            PS[i, j] ⟵ PS[i, OK−1] || '1' || PS[OK+1, j]
   end for i
end for diff
```

FIGURE 4: *Optimal algorithm*

The main loop starts then on line 9. The table is filled primarily by diagonals, each corresponding to a constant difference $\mathit{diff} = j - i$, and within each diagonal, by increasing i. Line 11 redefines j just for notational convenience.

In lines 12–13, the table entries are given default values, corresponding to the extreme case of all $\mathit{diff}+1$ elements in the range between and including i and j remaining matching as initially given in the input. This corresponds to a bitstring of $\mathit{diff}+1$ zeroes '000···0' in PS. As to the cost of the default partition, we have to store $\mathit{diff}+1$ meta data blocks, at the total price of $(\mathit{diff}+1)FE$.

After having initialized the table, the loop starting in line 15 tries to partition the range (i, j) into two sub-pieces. The idea is to consider two possibilities for the optimal partition of the range $[i, j]$: either all the $\mathit{diff}+1$ elements should remain matching, as we assume in the default setting initializing the $C[i, j]$ value in line 12, or there is at least one element k, with $i \leq k \leq j$, which in the optimal partition should be turned into an NM-element. The optimal solution is then obtained by solving the problem recursively on the remaining sub-ranges $(i, k-1)$ and $(k+1, j)$. The advantage of this definition is that the surrounding elements of the sub-ranges, $i-1$ and k for $(i, k-1)$, and k and $j+1$ for $(i, k-1)$, are again both of type NM, so the same table C can be used.

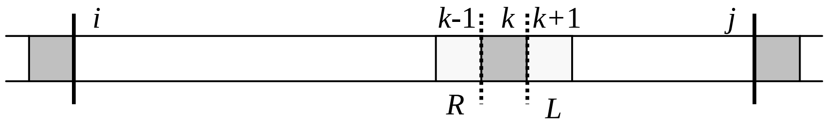

FIGURE 5: *Schematic representation of a partition of a sub-range*

However, to combine the optimal solutions of the sub-ranges into an optimal solution for the entire range, one needs to know whether the elements adjacent to the separating element indexed k are of type M or NM. For if one or both of them are NM, they can be merged with the separating element itself, so the meta-data decreases by one or two elements, reducing the price by FE or $2FE$. Let L denote type, 0 or 1, corresponding to M or NM, of the leftmost element of the right range $[k+1, j]$, and R the type of the rightmost element of the left range $[i, k-1]$. These values are assigned in lines 16–17, including extremal values. The functions left(B) and right(B) return, respectively, the leftmost and rightmost bit of a given bitstring B. The general case is depicted in Figure 5. We thus need a function $f(L, R)$, giving the number of additional meta-data elements needed as function of the type of the bordering elements, L and R. This function should give values according to Table 1. A possible function is thus $f(L, R) = 1 - L - R$, which explains the definition of the *newcost* in line 18.

L	R	$f(L,R)$
1	1	-1
0	1	0
1	0	0
0	0	1

TABLE 1: *Values for* $f(L, R)$

We check the sum of the costs of the optimal solutions of the subproblems plus the cost of the separating element, and keep the smallest such sum, over all the possible partition points k, in the table entry $C[i, j]$. In other words,

$$C[i,j] \leftarrow \min \begin{cases} (\mathit{diff}+1)FE, \\ \min\limits_{i \le k \le j} \left(C[i,k-1] + C[k+1,j] + s(k) + (1-L-R)FE \right). \end{cases}$$

OK stores the value of k for which the optimal partition has been found, i.e., that with minimum cost. If the default value has been changed, the optimal solution, expressed as a bitstring of length $\mathit{diff} + 1$, is obtained in line 24 by concatenating the bitstrings corresponding to the optimal solutions of the subranges and between them the string $'1'$ corresponding to the element indexed k.

The complexity of evaluating the table is dominated by the loops starting at line 9. There are three nested loops, and the loop on k goes from i to $j - 1 = i + \mathit{diff} - 1$, so it is executed *diff* times for each possible value of *diff* and i. The total number of iterations is therefore

$$\sum_{i=1}^{n-1} i(n-i) = \left[n\frac{n(n-1)}{2} - \frac{(n-1)n(2n-1)}{6} \right] = \frac{1}{6}(n^3 - n).$$

Such a cubic number of iterations might be prohibitive, even though the coefficient of n^3 is at most 0.17. Recall that n, the input parameter of the number of consecutive blocks dealt with in each call to the program for the optimal partition, is the number of consecutive matching items between two non-matching ones. In terms of our bit-string notation: the result of applying the deduplication algorithm of a large input chunk is a sequence of matching or non-matching items, which we denoted by a bit-string of the form, e.g., 10010001011100000001 00.... The optimal partition algorithm is then invoked for each of the 0-bit runs, which, on the given example, are of lengths 2, 3, 1, 0, 0, 7, etc. There is of course no need to call the procedure when $n = 0$.

5. Improvements

5.1. Reducing the time complexity

If certain values of n are too large, one may try to reduce the complexity a priori by applying a preliminary filtering heuristic that will not impair the optimal solution. For example, one could consider the maximal possible gain from declaring a matching item (0) to be non-matching (1). This happens if the two adjacent blocks are non-matching themselves, and then all 3 items could be merged into a single one. The savings would then be equivalent to $2FE$ bytes, which have to be counterbalanced by the loss of $s(i)$ bytes that are not referenced anymore, so have to be stored explicitly. Thus, if $s(i) > 2FE$, the ith M-element will surely not be transformed into an NM-element. It follows that $s(i) > 2FE$ is a sufficient condition for keeping the value of the ith bit in the optimal partition as 0.

The heuristic will then scan all the input items and check this condition for each 0-item. If the condition holds, the element can be declared to remain of type 0, which partitions the rest of the elements into two parts. For example, if the middle element of n is thereby declared as keeping its 0-status, we have split the n elements into two parts of size $n/2$ each, so the complexity is reduced from $\frac{1}{6}n^3$ to $2\frac{1}{6}\left(\frac{n}{2}\right)^3 = \frac{1}{24}n^3$. Returning to the example bit-string above 10**0**10**0**010111000**00**00100..., if the boldfaced elements are those fixed by the heuristic in their 0-status, the algorithm will be invoked with lengths 1, 1, 1, 1, 3, 2, etc. Theoretically, the worst case didn't change, even after applying this heuristic, but in practice, the largest values of n might be much smaller.

There remains a technical problem: the optimal partition evaluated in $C[i, j]$ is based on the assumption that the surrounding elements $i - 1$ and $j + 1$ were of type 1, and if the above heuristic is applied, this assumption is not necessarily true. Two approaches are possible to confront this problem. We could use the value of $C[i, j]$ and the corresponding partition in $PS[i, j]$ and adapt it locally to the cases if one of the surrounding elements is 0. For example, if the rightmost bit in $PS[i, j]$ is 0, and bit $j + 1$ is also 0, then no adaptation is needed; but if the rightmost bit in $PS[i, j]$ is 1, and bit $j + 1$ is

0, then the optimal value $C[i, j]$ took into account that elements j and $j+1$ were merged, which is not true in our case, so the value of $C[i, j]$ has to be increased by one meta-data element, that is by FE. A similar adaptation is needed for the left extremity, element $i-1$. Such an adaptation is not necessary optimal, since it might be possible that, had we known that the surrounding elements are not both 1, an altogether different solution will be optimal.

As a second approach, we could extend the definitions of the $C[i, j]$ and $PS[i, j]$ tables to be 4-dimensional, with $C[i, j, L, R]$ being the cost of the optimal partition of the elements $i, i+1, \ldots, j$, under the assumption that the bordering elements $i-1$ and $j+1$ are of type L and R, respectively, where $L, R \in \{0, 1\}$. Similarly, $PS[i, j, L, R]$ will hold the optimal partition for the given parameters. There are only four possibilities for L and R: $LR \in \{00, 01, 10, 11\}$, and the total size of each table is therefore only $2n^2$.

As above, one tries to partition the range (i, j) into two pieces, just without a separating element as before. The ranges will be (i, k) and $(k+1, j)$, for some $i \leq k < j$. L and R still denote the elements to the left of i and to the right of j, respectively, but we also need the bordering elements of the subranges, which again can be of type M or NM, denoted by 0 or 1, respectively. We therefore need to iterate on the possible internal left and right values IL and IR. It might be easiest to understand the notation by referring to the schema in Figure 7. The left subrange, (i, k), is delimited on its left by L and on its right by IL, whereas the right subrange, $(k+1, j)$, is delimited on its left by IR and on its right by R. The notation thus refers each bordering element to the position of the corresponding subrange, rather than to its own position, which is why IL appears in the figure to the right of IR.

Iterating on the four possibilities for (IL, IR), we have to check for consistency. Suppose, for example, that we consider $IL = 0$. That means that we are looking for the optimal partition of the left range (i, k), under the condition that the bordering elements are L and $IL = 0$. But we have also to check that the complementing optimal solution of the right range $(k+1, j)$ is such that its leftmost bit is indeed 0. A similar consistency check verifies that the optimal solution for the right range $(k+1, j)$ is taken for the given value of IR and that indeed, the rightmost bit of the string corresponding to the left range (i, k) is consistent with this IR value. If there is consistency, we check the sum of the costs of the optimal solutions of the sub-problems, and keep the smallest such sum, over all the possible partition points k. If there is no consistency for any k, the default value of keeping all bits as 0 is chosen. We omit here the formal algorithm and the

```
for LR ∈ {00, 01, 10, 11}
   C[n + 1, n, L, R]  ⟵  0      PS[n + 1, n, L, R]  ⟵  Λ
for i  ⟵  1 to n
   for LR ∈ {00, 01, 10, 11}
      C[i, i − 1, L, R]  ⟵  0      PS[i, i − 1, L, R]  ⟵  Λ
   C[i, i, 0, 0]  ⟵  FE      PS[i, i, 0, 0]  ⟵  '0'
   if s(i) < FE then
      C[i, i, 0, 1]  ⟵  s(i)      PS[i, i, 0, 1]  ⟵  '1'
      C[i, i, 1, 0]  ⟵  s(i)      PS[i, i, 1, 0]  ⟵  '1'
   else
      C[i, i, 0, 1]  ⟵  FE      PS[i, i, 0, 1]  ⟵  '0'
      C[i, i, 1, 0]  ⟵  FE      PS[i, i, 1, 0]  ⟵  '0'
   if s(i) − FE < FE then
      C[i, i, 1, 1]  ⟵  s(i) − FE      PS[i, i, 1, 1]  ⟵  '1'
   else
      C[i, i, 1, 1]  ⟵  FE      PS[i, i, 1, 1]  ⟵  '0'
end for i

for diff  ⟵  1 to n − 1
   for i  ⟵  1 to n − diff
      j  ⟵  i + diff
      for LR ∈ {00, 01, 10, 11}
         C[i, j, L, R]  ⟵  (diff + 1)FE
         PS[i, j, L, R]  ⟵  '000···0'    //(length diff + 1)
         OK  ⟵  0
         for k  ⟵  i to j − 1
            for IL IR ∈ {00, 01, 10, 11}
               if  left(PS[k + 1, j, IR, R]) = IL  and  right(PS[i, k, L, IL]) = IR
                  newcost  ⟵  C[i, k, L, IL] + C[k + 1, j, IR, R] − (IL × IR)FE
                  if newcost < C[i, j, L, R]
                     C[i, j, L, R]  ⟵  newcost
                     OK  ⟵  k    OL  ⟵  IL    OR  ⟵  IR
            end for IL IR
         end for k
         if OK > 0 then
            PS[i, j, L, R]  ⟵  PS[i, OK, L, OL]  ||  PS[OK + 1, j, OR, R]
      end for LR
   end for i
end for diff
```

Figure 4a: *New Optimal algorithm*

```
C[n+1, n] ⟵ 0    LT[n+1, n] ⟵ 1    RT[n+1, n] ⟵ 1
for i ⟵ 1 to n
   C[i, i−1] ⟵ 0    LT[i, i−1] ⟵ 1    RT[i, i−1] ⟵ 1
   if s(i) − FE < FE then
      C[i, i] ⟵ s(i) − FE    S[i, i] ⟵ i
      LT[i, i] ⟵ 1    RT[i, i] ⟵ 1
   else
      C[i, i] ⟵ FE
      LT[i, i] ⟵ 0    RT[i, i] ⟵ 0
end for i

for diff ⟵ 1 to n − 1
   for i ⟵ 1 to n − diff
      j ⟵ i + diff
      C[i, j] ⟵ (diff + 1)FE
      LT[i, j] ⟵ 0    RT[i, j] ⟵ 0
      OK ⟵ 0
      for k ⟵ i to j
         L ⟵ LT[k+1, j]
         R ⟵ RT[i, k−1]
         newcost ⟵ C[i, k−1] + C[k+1, j] + s(k) + (1 − L − R)FE
         if newcost < C[i, j]
            C[i, j] ⟵ newcost
            OK ⟵ k
      end for k
      S[i, j] ⟵ OK
      if OK > 0 then
         LT[i, j] ⟵ LT[i, OK − 1]    RT[i, j] ⟵ RT[OK + 1, j]
   end for i
end for diff
```

Figure 6: *Optimal algorithm with reduced space complexity*

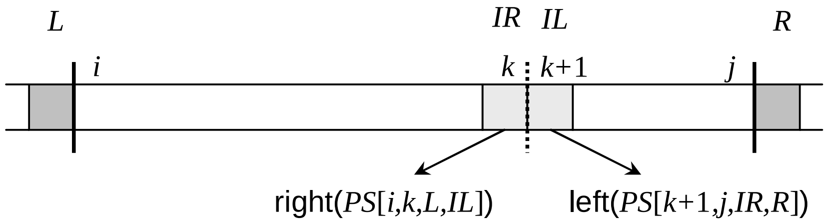

FIGURE 7: *Schematic representation of an alternative partition of a sub-range*

details.

5.2. Reducing the space complexity

While the time complexity is $\theta(n^3)$, the $C[i,j]$ table needs only n^2 space. But the strings stored in the $PS[i,j]$ table are of length $j-i+1$, so that the space for $PS[i,j]$ is also $\theta(n^3)$. We can reduce this and store only $O(1)$ for each entry at the cost of not giving the optimal partition explicitly, but providing enough information for the optimal partition to be built in linear time, similarly to what has been done in [5].

The key to this reduction is storing in $PS[i,j]$ (which we call now $S[i,j]$ to avoid confusions) not the string itself, but the value OK at which the range $[i,j]$ has been split in an optimal way (line 27), or leaving it undefined, if no such value OK exists. Since the string $PS[i,j]$ served also to provide information on its extremal elements (left and right in lines 16 and 17 of the algorithm in Figure 4), these elements have now to be saved in tables LT and RT on their own. The updated algorithm is given in Figure 6.

To build the optimal solution, we initialize a vector A with n zeros, and then change selected values according to the values in the $S[i,j]$ matrix, using the recursive procedure Fill_Sol, given in Figure 8. It will be invoked by Fill_Sol$(A,1,n)$. The total running time of the recursion is clearly bounded by n.

6. Conclusion

References

[1] ARONOVICH L., ASHER R., BACHMAT E., BITNER H., HIRSCH M., KLEIN S.T., The Design of a Similarity Based Deduplication System, *Proc. SYSTOR'09*, Haifa, (2009) 1–14.

```
Fill_Sol(A, i, j)
   if j ≥ i AND S[i, j] is defined
      k  ⟵  S[i, j]
      A[k]  ⟵  1
      Fill_Sol(A, i, k − 1)
      Fill_Sol(A, k + 1, j)
```

FIGURE 8: *Construction of the optimal solution*

[2] ARONOVICH L., ASHER R., HARNIK D., HIRSCH M., KLEIN S.T., TOAFF Y., Similarity based Deduplication with small data chunks, *Proc. Prague Stringology Conference PSC–2012*, Prague, (2012) 3–17.

[3] http://corpus.canterbury.ac.nz/

[4] CORMEN T.H., LEISERSON C.E., RIVEST R.L., *Introduction to Algorithms*, MIT Press, 1990.

[5] KLEIN S.T., On the Use of Negation in Boolean IR Queries, *Information Processing & Management* **45** (2009) 298–311.

[6] QUINLAN S., DORWARD S., Venti: A New Approach to Archival Storage, *Proceedings of FAST'02, the 1st USENIX Conference on File and Storage Technologies* , Monterey, CA (2002) 89–101.

[7] ZHU B., LI K., PATTERSON H., Avoiding the Disk Bottleneck in the Data Domain Deduplication File System, *Proceedings of FAST'08, the 6th USENIX Conference on File and Storage Technologies*, San Jose, CA (2008) 279–292.

[8] ZIV J., LEMPEL A., A Universal Algorithm for sequential data compression, *IEEE Trans. on Information Theory* **23** (1977) 337–343.