Supplementary for Homomorphic Compression

1 EXAMPLES OF HC SCHEMES

Drawing upon the theoretical foundations of homomorphic compression elucidated in our paper, this supplementary material elaborates on three homomorphic compression schemes, namely RLE, LZW, and TADOC, which have been implemented in HOCO. These schemes are selected from two categories: FHC and PHC. LHC is not exemplified as its features are encompassed within PHC. Moreover, UNHC schemes, which require decompression prior to performing operations, are not highlighted as their implementation of homomorphic operations is nothing more than decompression, processing the uncompressed text, and compression. So we do not provide specific notes for UNHC either.

R2.RE

1.1 Basic homomorphic operations

Before diving into each schemes, we explain the four basic homomorphic operations as introduced in Section 3.1.

- `extract(offset_start, len)`. This operation is characterized by two positive integer parameters. The first parameter, `offset_start`, denotes the initial offset at which the user intends to perform the operation on the uncompressed text. The second parameter, `len`, represents the total number of symbols contained in the uncompressed text. The main utility of this operation is to locate the specified `offset_start` and retrieve an uncompressed sequence of a given length `len` as the output.
- `insert(offset_start, str)`. This operation is characterized by two parameters. The first parameter, `offset_start`, is identical to the one used in the extract operation. The second parameter, `str`, represents an uncompressed symbol sequence that needs to be inserted. The primary purpose of this operation is to locate the specified `offset_start`, insert the new text represented by `str`, and return the resulting compressed text.
- `delete(offset_start, len)`. This operation shares the same parameters as the extract operation. Its purpose is to locate the specified `offset_start`, delete the sequence of symbols with the given length `len`, and return the resulting compressed text.
- `symbol_compare(s_1, s_2)`. This operation involves two symbol parameters, `s_1` and `s_2`, and performs a comparison between them.

The `extract`, `insert`, and `delete` operations share identical forms in both uncompressed and compressed contexts, with the main distinction lying in their internal implementations. In the uncompressed context, the implementations are straightforward. For example, consider the execution of `extract(5, 3)` on an uncompressed text *aaaabbaa*. This operation directly locates the fifth character in the text and extracts the subsequent three characters *baa* as the output. However, in the compressed context, the primary objective of homomorphic compression is to isolate the compressed text stored in the underlying storage from the upper-layer applications. To achieve this, homomorphic operations act as intermediaries, receiving processing requests based on uncompressed text and transforming them to be executed on the compressed text. The outputs are then provided in the form of uncompressed text.

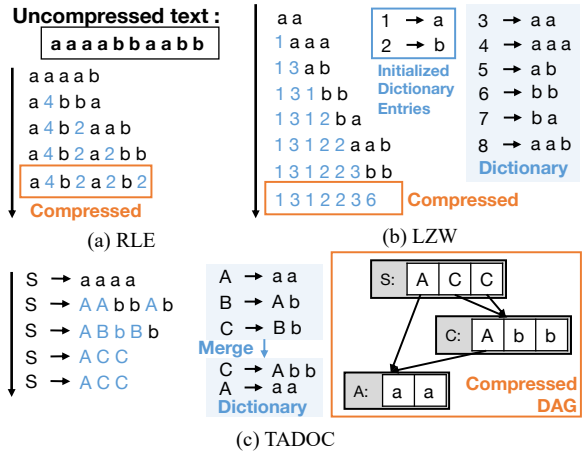


Figure 1: Compression algorithms of four homomorphic compression schemes.

The `symbol_compare` operation demonstrates consistent behavior in both uncompressed and compressed contexts. It compares two arguments, `s_1` and `s_2`, which typically represent individual characters. The underlying implementation of this operation remains consistent, given that compressed text is essentially a string. However, the definition of `symbol_compare` does not exclude the possibility of expanding and comparing two compressed symbols such as rules or dictionary entries, ensuring the completeness of the operation set. It is important to note that in practical applications, users interact with the system from the standpoint of uncompressed text, rendering compressed symbols invisible. Therefore, direct requests to compare two compressed symbols do not exist. In most cases, comparing the characters of the compressed symbols themselves is sufficient to determine their equality.

R1.O1 R2.RE

1.2 Converting Compression Algorithms to Homomorphic Compression Schemes

In this section, we delve into the intricate details of the three HC schemes implemented in HOCO through illustrative examples. The processing flows of the original compression algorithms are shown in Figure 1.

RLE algorithm to RLE HC scheme. RLE is a modest algorithm that compresses data by counting the consecutive occurrences of characters, as shown in Figure 1 (a). It is simple and fast, and is well suited for offset mapping. On top of RLE, we construct a homomorphic compression scheme where the *Comp* and *Decomp* steps follow the original algorithms of RLE. While for its operations, we give a general description as follows:

- Extract. This operation locates `offset_start` by scanning and accumulating the repetitive lengths of the characters, and continues until `len` is achieved. For example, to execute `extract(5,3)` on the compressed text obtained in Figure 1 (a), the *Eval* step sets a read pointer at the start of the compressed text, and shifts

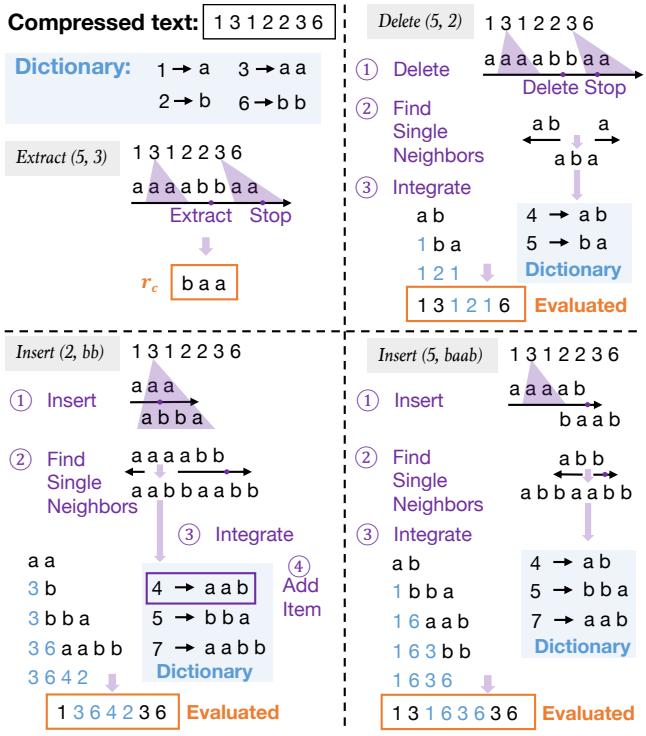


Figure 2: Homomorphic operations of the LZW scheme.

the pointer sequentially to read the characters along with their repetition lengths. When the character b and length 2 are read, `offset_start=5` is reached, it triggers the extraction and finally outputs *baa*.

- Insert. This operation follows the same offset mapping procedure as `extract`. Then two cases arise: (1) if `offset_start` is inside a *run*, it first expands this run and inserts `str` into it; (2) if `offset_start` lies between two adjacent runs, it expands both runs and inserts `str`. Finally, local integration is performed on the modified part. For example, to perform `insert(5, baab)`, it locates $(b, 2)$ and finds that 5 is within this run. Then it expands $(b, 2)$ to *bb* and inserts `str` to form a new symbol sequence *bbaabb*. Finally, it performs a local integration to get $(b, 2)(a, 2)(b, 2)$ and replaces the original $(b, 2)$.
- Delete. This operation is similar to the `extract` operation, except that it removes the target symbol sequence and integrates the runs at the start and end offsets. For example, to execute `delete(5,3)`, it locates $(b, 2)$ and removes *baa*. Then the remaining *b*s of the starting run and $(b, 2)$ of the ending run are integrated into $(b, 3)$.
- Symbol_compare. This operation provides a comparison of two types of symbols: characters and runs. Specifically, the principle for comparing runs is: comparing characters if they are different, otherwise comparing their repetitive lengths.

All four homomorphic operations of the RLE scheme satisfy directness and strong homomorphism. Thus the RLE scheme belongs to **FHC**.

LZW algorithm to LZW HC scheme. LZW is a dictionary-based compression, which mainly extracts the smallest unseen substring from the residual uncompressed symbol string. We refer to the example depicted in Figure 1 (b) for illustration. Initially, a dictionary is created to contain all possible single symbols. In the given example, the dictionary is initialized with $1 \rightarrow a$ and $2 \rightarrow b$. Then, the algorithm operates by scanning through the input string until it encounters an unseen substring, e.g., *aa*. Upon identifying such a string, the algorithm retrieves the index of the substring without the last character, which in this case is $1 \rightarrow a$, from the dictionary. The last character, denoted as *a*, is subsequently employed as the starting symbol for the next scanning phase. Based on LZW, we construct a homomorphic compression scheme in which the *Comp* and *Decomp* algorithms are inherited from LZW. We provide an illustration of the main homomorphic operations in Figure 2 and give the following description. M.3 R2.O1 R2.O3

- Extract. This operation scans the compressed text with the aid of dictionary, and accumulates the current offset (e.g., `offset_cnt`) until the specified `offset_start` is reached. For example, when executing `extract(5, 3)`, it scans the compressed text and counts the number of characters contained in the dictionary entries until offset 5 is reached at the end of 1312. In detail, during the scanning process, the algorithm encounters the first character *a* (represented by the dictionary entry $1 \rightarrow a$), and increments the `offset_cnt` by 1. Next, it expands the entry 3 to get *aa* according to the dictionary, and adds 2 to `offset_cnt`. The subsequent indexes 12 corresponds to the substring *ab*. At this point, the offset 5 is reached, and the extraction begins. The algorithm continues fetching characters until the given `len` is met. In this case, it reads $3 \rightarrow aa$, and then stops, returning the extracted sequence *baa*. M.3 R2.O1 R2.O3 R2.RE
- Insert. This operation, upon locating `offset_start`, examines its position relative to dictionary entries. If `offset_start` falls within an entry, the operation expands the entry and inserts the new string. If `offset_start` is located between two entries, the operation inserts the substring directly. It then identifies single-symbol neighbors starting from the modification point, integrates them locally, and removes unnecessary dictionary entries. As illustrated in Figure 2, `insert(2, bb)` first locates within the entry 3. Then it adds *bb* into $3 \rightarrow aa$ to get *abba*. It then extends from the modification point in both directions to encompass neighboring entries with single symbols, yielding the uncompressed sequence *aabbaabb*. This sequence undergoes local integration using the LZW algorithm, generating the compressed text 3642. Finally, the dirctionary is updated accordingly. Similarly, `insert(5, baab)` inserts *baab* directly between two adjacent entries $2 \rightarrow b$. By combining the neighboring single symbols, it obtains the uncompressed sequence *abbaabb*. This sequence is further locally integrated to generate the compressed text 1636. Finally, useless dictionary entries are removed. M.3 R2.O1 R2.O3
- Delete. This operation, after locating `offset_start`, continues scanning and deleting until `len` is reached. Both the start and end offsets can be within or between dictionary entries. Similar to the `insert` operation, it identifies single-symbol neighbors from the modification point to both sides and integrates them locally. Then it removes unnecessary dictionary entries. Consider `delete(5, 2)` as an illustration. The start point is situated M.3 R2.O1 R2.O3

between two dictionary entries, while the end point falls within the entry $3 \rightarrow aa$. This necessitates the expansion of the entry 3 at the end point. Upon removing the substring ba, it explores adjacent entries with single symbols, leading to the sequence aba. As no further compression can be applied on this sequence, the compressed text 121 is generated.

- Symbol_compare. This operation provides a comparison of characters and dictionary entries, where for entries the comparison is between their corresponding symbol strings.

All four homomorphic operations of the LZW scheme satisfy directness. `extract` and `symbol_compare` do not change the compressed text, and can be performed an unlimited number of times. The local integration steps of `insert` and `delete` lead to inconsistencies in their evaluated and fresh compressed text, as well as the inability to fully exploit the redundancy of the global text. Nonetheless, we still prefer the local integration operation, as it exhibits better efficiency and a tolerable loss of compression ratio than iterating over all subsequent texts for expansion and integration, which is tantamount to performing re-compression. To sump up, the LZW scheme belongs to **PHC**.

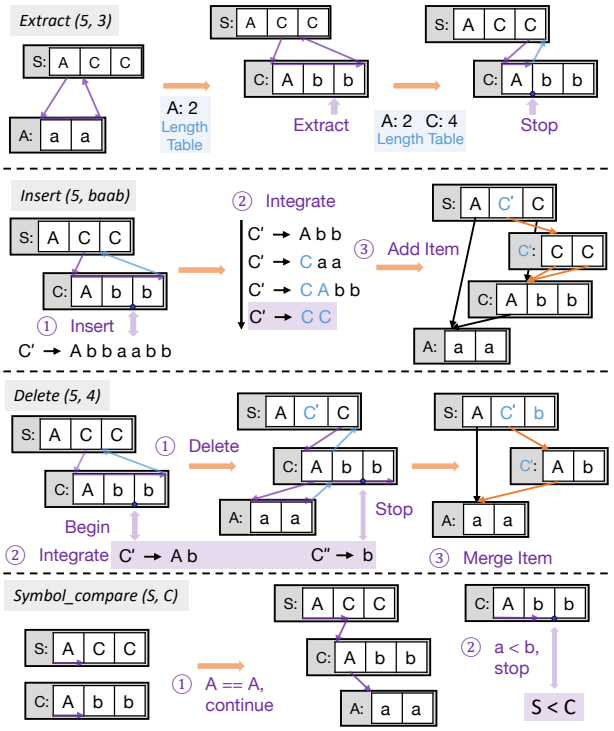
TADOC scheme to TADOC HC scheme. TADOC is a compressed data processing technique built upon the grammar-based compression algorithm called sequitur. Sequitur operates by scanning the symbol string and recording adjacent symbol pairs, including the newly generated symbols resulting from replacements. Once a repetitive symbol pair appears, it is replaced directly if the pair already exists in the dictionary. Otherwise, a new symbol is generated to substitute the current match. After all replacements are completed, symbols that are less frequently used are reverted to avoid additional space consumption. In the provided example in Figure 1 (c), as the algorithm processes the input sequence $aaaa$, it identifies the repetitive symbol pair aa and replaces it with a newly generated symbol A, resulting in new sequence AA. Then, when encountering the sequence $AAbbAb$, the algorithm recognizes the repetitive symbol pair Ab and replaces it with the symbol B. Similarly, the repetitive symbol pair Bb is substituted with the symbol C. Finally, due to the fact that the symbol B appears only once, it is reverted back to its original form. M.3 R2.O1 R2.O3

Sequitur treats a dictionary entry as a non-terminator or *rule*. And the final grammar can be represented as a directed acyclic graph (DAG). Therefore, we construct a homomorphic compression scheme in which the *Comp* and *Decomp* steps follow Sequitur.

TADOC achieves efficient `extract` and `insert` operations through a delicate design of auxiliary data structures. However, there are three aspects of TADOC that deserve rethinking: first, its auxiliary structures are complex; second, its implementation of insertion does not satisfy the directness property; third, it does not support deletion. However, we still refer to our constructed homomorphic compression scheme as the TADOC scheme, since we will draw on the efficient text analysis implementation of TADOC in the following sections. Now along with the examples in Figure 3, we describe the homomorphic operations of the TADOC scheme.

- Extract. This operation works in a DFS fashion. It traverses the compressed DAG starting from the root rule and tracks the current offset until the string of `len` from `offset_start` is read. In particular, it records the length of the accessed rules to avoid repeated recursive visits. For example, in Figure 3, the operation first explores the rule A and knows that it contains two characters. It then records the length of rule A to the length table. Next, when it continues to traverse the rule C, the rule A appears again. Now the operation obtain the length of rule A directly from the length table, bypassing the need for useless internal traversal. M.3 R2.O1 R2.O3
- Insert. This operation accesses the target offset in the same way as `extract`. There are different strategies depending on whether the insertion point is in the root rule or not, since the root rule has no parent node and thus does not require any modification to its upper nodes. If the insertion point is not in the root node, it creates a new rule with the newly inserted text `str`. As in the case of the `insert(5, baab)` in Figure 3 where the starting offset of 5 resides within the subrule C, it creates a new rule with the newly inserted text `str` ($C' \rightarrow Abbaabb$). and this new rule is further integrated locally ($C' \rightarrow CC$). Then it updates the dictionary and replaces the node at the corresponding position in the parent rule with the new rule ($S \rightarrow AC'C$). If the insertion point is in the root node, it performs a local integration of `str` and its neighbor characters (excluding the rule nodes for efficiency), then updates the dictionary. M.3 R2.O1 R2.O3
- Delete. This operation also differs depending on whether the start and end modification points are within the root node or not. If the modification point is not in the root node, it locally integrates the remaining part of the rule that the modification point belongs to, and replace the node at the corresponding position in the higher-level rule. As in Figure 3, the `delete` operation creates a new rule C' from the remaining part of the rule C, and then performs a local integration of C', followed by changing the parent node to $S \rightarrow AC'C$. It continues to traverse until reaching the end point. A new rule $C'' \rightarrow b$ is constructed and integrated. Finally, the parent node is updated to $S \rightarrow AC'b$. While if a modification point is in the root node, the left and right character neighbors of the modification point are integrated locally.
- Symbol_compare. This operation compares characters and rules, where the rules need to be compared by means of DFS to the symbol strings they represent. As in Figure 3, when comparing rules S and C, it first skips the common rule A, and then initiates a graph traversal from the point of divergence to compare their corresponding uncompressed characters. Since the characters at offset 2 appears $a < b$, the operation returns $S < C$. M.3 R1.O1 R2.O1, R2.O3

All four homomorphic operations of TADOC satisfy directness. In particular, `extract` and `symbol_compare` meet strong homomorphism. While the local integration steps of `insert` and `delete` result in the differences between their evaluated and fresh compressed texts. For example, the new rule C's generated by the `insert` and `delete` operations in Figure 3 appear only in the root rules. It is redundant to set up such a separate grammar C' compared to expanding and replacing it. However, consider a case where the grammar is complex and the DAG is deep. Recursively integrating and replacing in a bottom-up manner after a minor modification to one of the rules would incur huge overhead. Therefore, we prefer locally integrated solutions for its efficiency. In summary, the TADOC scheme is a **PHC**.



R1.O1 **Figure 3: Homomorphic operations of the TADOC scheme.**

2 WORD COUNT EXAMPLE USING HOCO DSL

R2.O2 In this section, we present a concrete example for the HOCO DSL template outlined in our paper, providing a clearer exposition of HOCO's evaluation module. Specifically, we focus on a `word count` task, which entails determining the frequency of occurrence for each word in a given text. By completing the HOCO DSL template based on the processing logic for uncompressed text, users can leverage the HOCO evaluation module to perform code transformation and optimization specifically designed for processing compressed text directly.

Listing 1 showcases a simple DSL implementation that demonstrates the `word count` task. According to the DSL writing rules, the storage path for the text is specified, and the processing granularity is set at the word level. As word counting is a simple and repetitive task, a single code segment is employed to encapsulate the core functionality. Additionally, a global map structure is initialized to serve as the word counter.

Within `SEGMENT 1`, the essential functionality for the `word count` task is implemented. The implementation of this segment closely resembles that of a function, requiring the initialization of local variables or data structures specific to the segment. In the provided example, the variable `current_offset` is utilized to track the current position in the text, denoting the word number since the processing granularity is predefined as `WORD`. Its initial value is set as the keyword `FILE_START`. The termination condition is defined by checking whether `current_offset` is equal to `FILE_END`, indicating the completion of processing all words in the text. Within the key action loop, the code invokes the `EXTRACT` API to read one word during each iteration, and updates the global counter accordingly.

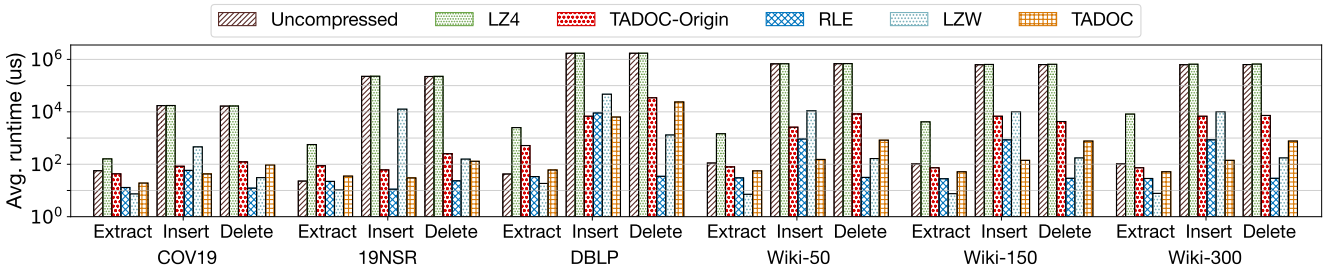
Notably, HOCO incorporates special handling for the keywords `FILE_START` and `FILE_END`. By overloading associated operators such as "=" and "==", HOCO enables seamless adaptation to the start and end conditions specific to compressed text.

Listing 1: Word Count using HOCO DSL.

```
1  TEXT_DIR = Input/
2  ELEMENT = WORD
3  CODE_SEGMENT_NUM = 1
4  GLOBAL_STRUCTURE_INIT = {
5      map<string, int> wordCount;
6  }
7  SEGMENT 1:
8      LOCAL_STRUCTURE_INIT = {
9          string word;
10         Offset current_offset = FILE_START;
11     }
12     END_CONDITION = { current_offset == FILE_END; }
13     while( !END_CONDITION ) {
14         word = EXTRACT(current_offset++, 1);
15         if( wordCount.find(word) == wordCount.end())
16             wordCount[word] = 1;
17         else     wordCount[word] ++;
18     }
```

3 RUNTIME EVALUATION OF BASIC OPERATIONS

In our experimental setup, the throughput, defined as the number of operations per unit of time, is nearly identical to the average runtime. While it is recognized that read operations tend to occur more frequently than write operations in real-world workloads, accurately capturing precise data distributions under real-world scenarios is challenging. Therefore, in order to provide an estimate of maximum achievable efficiency, we quantify throughput by executing all operations for half a million times consecutively. Nevertheless, we concur that runtime metrics offer a more intuitive approach for evaluation. Therefore, we include the runtime metric in this context to provide a more comprehensive assessment of the performance and efficiency of each scheme, as depicted in Figure 4. The average runtime metric is intuitive and remains independent of the specific number of operations performed, making it a more intuitive metric for evaluation. R4.O2



R4.O2 **Figure 4: Average runtime of basic operations.** – RLE, LZW, and TADOC are HOCO's three HC schemes.