Examples of Homomorphic Compression Schemes

1 EXAMPLES OF HC SCHEMES

Based on the homomorphic compression theory disccussed in our paper, we introduce three homomorphic compression schemes implemented in HOCO. We select these schemes from two categories: FHC and PHC. LHC is not exemplified because its features are contained in PHC. And as all compression algorithms that must be decompressed before operating belong to UNHC, their implementation of homomorphic operations is nothing more than decompression, processing the uncompressed text, and compression. So we do not make special notes on UNHC either. Before diving into each schemes, we explain the four basic homomorphic operations introduced in Section 3.1. `extract(offset_start, len)` locates `offset_start` and returns an uncompressed sequence of a given length `len`. `insert(offset_start, str)` locates `offset_start`, inserts new text `str`, and returns the evaluated compressed text. `delete(offset_start, len)` locates `offset_start`, deletes the sequence of given length `len`, and returns the evaluated compressed text. `symbol_compare(s_1, s_2)` compares symbols `s_1` and `s_2`, which usually comes after `extract`. Note that the parameter `len` refers to the size of the uncompressed text.

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. E.g., to execute `extract(5,3)` on the compressed text of Figure 1 (a), the *Eval* step sets a read pointer at the start of the compressed text, and shifts 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. E.g., to perform `inset(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. E.g., 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.

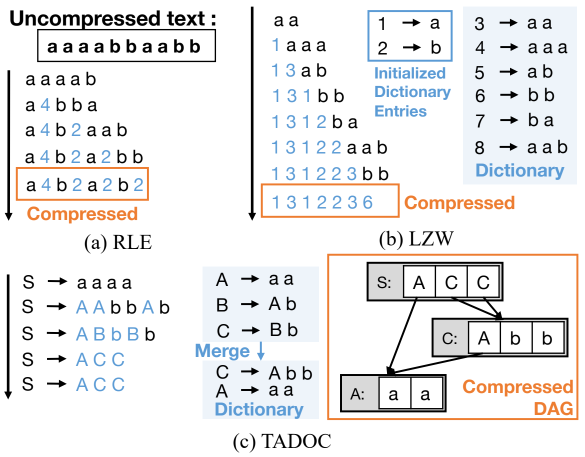

Figure 1: Compression algorithms of four homomorphic compression schemes.

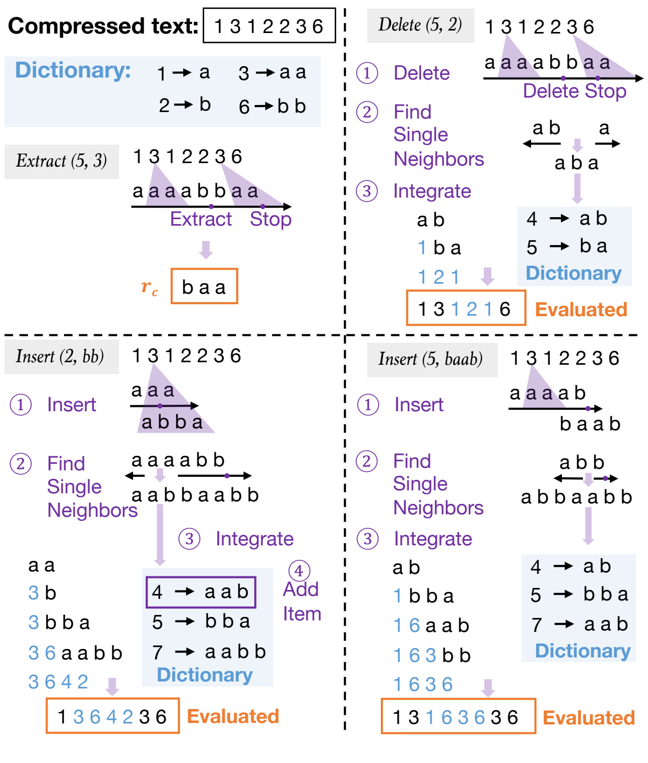

Figure 2: Homomorphic operations of the LZW scheme.

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

LZW is a dictionary-based compression, which mainly extracts the smallest unseen substring from the residual uncompressed symbol string. We use the example in Figure 1 (b) for illustration. First, in the initial stage, a dictionary is initialized to contain all possible single symbols. As in Figure 1 (b), the dictionary is initialized with $1 \longrightarrow a$ and $2 \longrightarrow b$. Second, the algorithm works by scanning through the input string until it finds an unseen substring, e.g., *aa*. Once such a string is found, the index of the string without the last character ($1 \longrightarrow a$) is retrieved from the dictionary and sent to the output, then a new item ($3 \longrightarrow aa$) is added to the dictionary. The last character (a) is then used as a starting symbol for the next scan. Third, dictionary entries are discarded except for the single symbols. Decompression reconstructs the dictionary by performing an inverse process of compression. Based on LZW, we construct a homomorphic compression scheme in which the *Comp* and *Decomp* algorithms are inherited from LZW, but store the valid parts of the dictionary. For the example in Figure 1 (b), the final dictionary of the LZW scheme retains only four items: $1 \longrightarrow a$,$2 \longrightarrow b$,$3 \longrightarrow aa$,and $6 \longrightarrow bb$, because only these substrings recur. We provide an illustration of the main homomorphic operations in Figure 2 and give the following description.

- Extract. This operation scans the compressed text with the aid of dictionary, and accumulates the current offset until it reaches `offset_start`. E.g., to execute `extract(5, 3)`, it scans the compressed text until offset 5 is reached at the end of 1312. Then it starts fetching until the given `len` is met when it reads $3 \longrightarrow aa$. At this point it stops and returns *baa*.
- Insert. This operation, after locating the target offset, first determines whether `offset_start` is inside a dictionary entry or between two entries. If it is within an entry, it expands this entry and inserts `str`, e.g. `insert(2, bb)` inserts *bb* in $3 \longrightarrow aa$ to get *abba*. While if `offset_start` is between two entries, it inserts `str` directly, e.g. `insert(5, baab)` inserts *baab* directly between $2 \longrightarrow b$ and $3 \longrightarrow aa$. Next, the insert operation detects single-symbol neighbors on both sides, starting from the modification point `offset_start`, then compresses and integrates them locally. Finally, useless dictionary entries are removed.
- Delete. This operation, after locating the target offset `offset_start`, continues scanning and deleting until `len` is reached. Both start and end offsets can occur within or between dictionary entries. But in either case, as in the `insert` operation, we detect single-symbol neighbors from the modification point to both sides and integrate them locally. Finally the useless dictionary entries are removed.
- 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 is a technique for operating on compressed data which is built on top of a grammar-based compression algorithm called sequitur. Sequitur works by scanning the symbol string while recording all adjacent symbol pairs (including the new symbols generated by the replacement). Once a repetitive symbol pair appears, it is replaced directly if the pair is already in the dictionary, otherwise a new symbol is generated to replace the current match. After all replacements are completed, the algorithm reverts the symbols that are used less frequently to avoid additional space consumption. As in Figure 1 (c), symbol B appears only once in the final result (i.e., $C \longrightarrow AB$), so it is reverted. 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 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 (e.g., in Figure 3, the length of rule A is recorded after being processed). So when A is subsequently accessed in rule C, it gets A's length directly from the length table and skips the internal traversal of A.
- 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. On the one hand, if the insertion point is not the root node, it creates a new rule with the newly inserted text `str` (i.e., $C' \longrightarrow Abbaabb$ in Figure 3), and this new rule is further integrated locally ($C' \longrightarrow CC$). Then it updates the dictionary and replaces the node at the corresponding position in the parent rule with the new rule ($S \longrightarrow AC'C$). On the other hand, if the insertion point is the root node, it performs a local integration of `str` and its neighbor characters (excluding the rule nodes for efficiency), then updates the dictionary.
- 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 insertion operation creates a new rule C' from the remaining part of the rule C where the starting deletion point is located, and then performs a local integration of C', followed by changing the

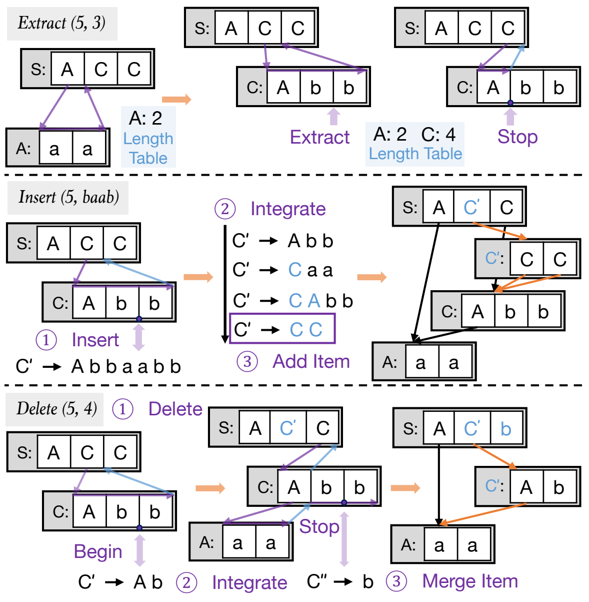

Figure 3: Homomorphic operations of the TADOC scheme.

parent node to $S \longrightarrow AC'C$. It continues to traverse until reaching the end point. A new rule $C'' \longrightarrow b$ is constructed and integrated. Finally the parent node is updated to $S \longrightarrow AC'b$. 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 provides a comparison of characters and rules, where the rules need to be compared by means of DFS to the symbols strings they contain.

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**.

2 WORD COUNT EXAMPLE USING HOCO DSL

Considering the HOCO code example of word count, shown in Listing 1, it first indicates the input file path, text access granularity (`WORD`), and a total of one code segment. Next, a global dictionary structure is initialized to hold word counts. Within the code segment, the required local data structures are first initialized, including setting the current offset to the starting position of the file (`FILE_START`), then the end condition is set to access to the end of the file (`FILE_END`). Inside the loop body, it calls the `EXTRACT` API iteratively to fetch a word (the length parameter is set to 1, same as what the extraction granularity set to `WORD` earlier), and then maintains the global word count dictionary. During compilation, HOCO translates this part into an operation done directly on the compressed text.

Listing 1: Word Count using HOCO DSL.

```
TEXT_DIR = "Input.dic"
ELEMENT = WORD
CODE_SEGMENT_NUM = 1
GLOBAL_STRUCTURE_INIT = {
    map<string, int> wordCount;
}
SEGMENT 1:
    LOCAL_STRUCTURE_INIT = {
        string word;
        ulong current_offset = FILE_START;
    }
    END_CONDITION = { current_offset == FILE_END; }
    while( !END_CONDITION ) {
        word = EXTRACT(current_offset++, 1);
        if( wordCount.find(word) == wordCount.end())
            wordCount[word] = 1;
        else    wordCount[word] ++;
    }
```