**Technical Report**

# iZENElib Data Compression

Ian Yang

March 31, 2009

*iZENEsoft*(Shanghai) Co., Ltd

**Abstract**

This is a report for the sub project data compression in *iZENElib*. Project data compression focuses on the search of large dictionaries if the data contained can be compressed.

# Contents

# 1 Introduction

The world is drowning in data. Classic algorithms are greedy in terms of their space usage and often cannot perform computations on all of the data. Nowadays, moderate scale information retrieval system must handle huge scale date. The entire dataset can be compressed in such occasion, but many problems still require the compressed dataset to be queried quickly.

The paper *An algorithmic framework for compression and text indexing*[1] presented a unified algorithmic framework to obtain nearly optimal space bounds for text compression and compressed text indexing, apart from lower-order terms. The authors provide a new tight analysis of text compression based on the Burrows-Wheeler transform (BWT). They also provide a new implementation of compressed text indexing based on the compressed suffix array (CSA). A key point of the unified approach is the use of the finite set model instead of the empirical probability model. This new perspective reveals an alternative way to model an arbitrary partition of the bwt. The encoding adopts an algorithm that stores t items out of a universe of size n in the information theoretic minimum space $\lg \binom{n}{t}$ bits (since there are $\binom{n}{t}$ subsets of t items out of n).

This project intends to provide an implementation of the unified framework, as well as the applications to compressed suffix array and text indexing. Chapter 2 briefly introduces the algorithms and data structures used in the framework. Chapter 3 represents the design outline.

# 2 Algorithm and Data Structures

## 2.1 Suffix Array

Suffix array is a data structure designed for efficient searching of a large text. The data structure is simply an array containing all the pointers to the text suffixes sorted in lexicographical (alphabetical) order. Each suffix is a string starting at a certain poinsition in the text and ending at the end of the text. Searching a text can be performed by binary search using the suffix array.

| m | i | s | s | i | s | s | i | p | p | i | # |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

Table 2.1: Input String $T$

The table 2.1 numbers the input text $T$. A sentinel letter # is append to the end. It appears only once and larger than any other letter in the string. The text $T$ has twelve suffixes: "`mississippi#`", "`ississippi#`", "`ssissippi#`", and so on down to "`i#`" and "`#`" that can be sorted into lexicographical order as in table 2.2.

If the original string is available, each suffix can be completely specified by the index of its first character. The suffix array is the array of the indices of suffixes sorted in lexicographical order. For the string "`mississippi#`", using one-based indexing, the suffix array is 8, 5, 2, 11, 1, 10, 9, 7, 4, 6, 3, 12, because the suffix "`ippi#`" begins at position 8, "`issippi#`" begins at position 5, "`ississippi#`" begins at position 2, and so forth.

The suffix array of a string can be used as an index to quickly locate every occurrence of a substring within the string. Finding every occurrence of the substring is equivalent to finding every suffix that begins with the substring. Because of the lexicographical ordering, these suffixes will be grouped together in the suffix array, and can be found efficiently with a binary search. If implemented straightforwardly, this binary search takes $O(m \log n)$ time, where $m$ is the length of the substring.

## 2.2 The BWT

Loosely speaking, the BWT produces a permutation $\texttt{BWT}(T)$ of the input text $T$ such that from $\texttt{BWT}(T)$ we can retrieve $T$ but at the same time $\texttt{BWT}(T)$ is much easier to compress.

| | |
|---|---|
| 1 | `ippi#` |
| 2 | `issippi#` |
| 3 | `ississippi#` |
| 4 | `i#` |
| 5 | `mississippi#` |
| 6 | `pi#` |
| 7 | `ppi#` |
| 8 | `sippi#` |
| 9 | `sissippi#` |
| 10 | `ssippi#` |
| 11 | `ssissippi#` |
| 12 | `#` |

Table 2.2: Sorted Suffix

The BWT is a very powerful tool and even the simplest algorithms that use it have surprisingly good performances. More advanced BWT-based compressors, such as `bzip` and `szip`, are among the best compressors currently available.

Another remarkable property of the BWT is that it can be used to build a data structure which is a sort of compressed suffix array for the input text $T$. Such data structure consists of a compressed version of $\texttt{BWT}(T)$ plus $o(|T|)$ bits of auxiliary information. This data structure can be used to compute the number of occurrences of an arbitrary pattern $p$ in $T$ in $O(|p|)$ time, and compute the position in $T$ of each one of such occurrences in $O(\log^{\epsilon}|T|)$ time.

The BWT consists of a reversible transformation of the input string $T$. The transformed string denoted by $\texttt{BWT}(T)$, contains the same characters as $T$ but it is usually easier to compress. The string $\texttt{BWT}(T)$ is obtained as follows (see Table 2.3). First, a special character # larger than any other character is appended to end of $T$. Then, the cyclic shifts of $T\#$ are sorted in lexical order. The output $\texttt{BWT}(T)$ is the last column of the sorted matrix. In Table 2.3, $\texttt{BWT}(T) = \texttt{ssmp\#pissiii}$.

Clearly, the BWT is related to suffix sorting, since the comparison of any two circular shifts must stop when the end marker # is encountered. The corresponding suffix array is a simple way to store the sorted suffix (see column '$SuffixArray$' in Table 2.3).

*TODO*: How to compress BWT. Context based partition and wavelet tree

| *Original* | *Sorted* | | | *Suffix Array* |
|---|---|---|---|---|
| $Q$ | $F$ | | $L$ | |
| mississippi# | i | ppi#missis | s | ippi# |
| #mississippi | i | ssippi#mis | s | issippi# |
| i#mississipp | i | ssissippi# | m | ississippi# |
| pi#mississip | i | #mississip | p | i# |
| ppi#mississi | m | ississippi | # | mississippi# |
| ippi#mississ | p | i#mississi | p | pi# |
| sippi#missis | p | pi#mississ | i | ppi# |
| ssippi#missi | s | ippi#missi | s | sippi# |
| issippi#miss | s | issippi#mi | s | sissippi# |
| sissippi#mis | s | sippi#miss | i | ssippi# |
| ssissippi#mi | s | sissippi#m | i | ssissippi# |
| ississippi#m | # | mississipp | i | # |

Table 2.3: Matrix $Q$ for the BWT containing the cyclic shifts of text $T = \texttt{mississippi}\#$ (column '*Original*'). Sorting of the rows of $Q$, in which the first ($F$) and last ($L$) symbols in each row are separated (column '*Sorted*'). Suffix array $SA$ for $T$ (column '$SuffixArray$').

# 3 Design

The data compression framework is decomposed into layers as illustrated as Figure 3.1. Following sections describe each layer in detail.

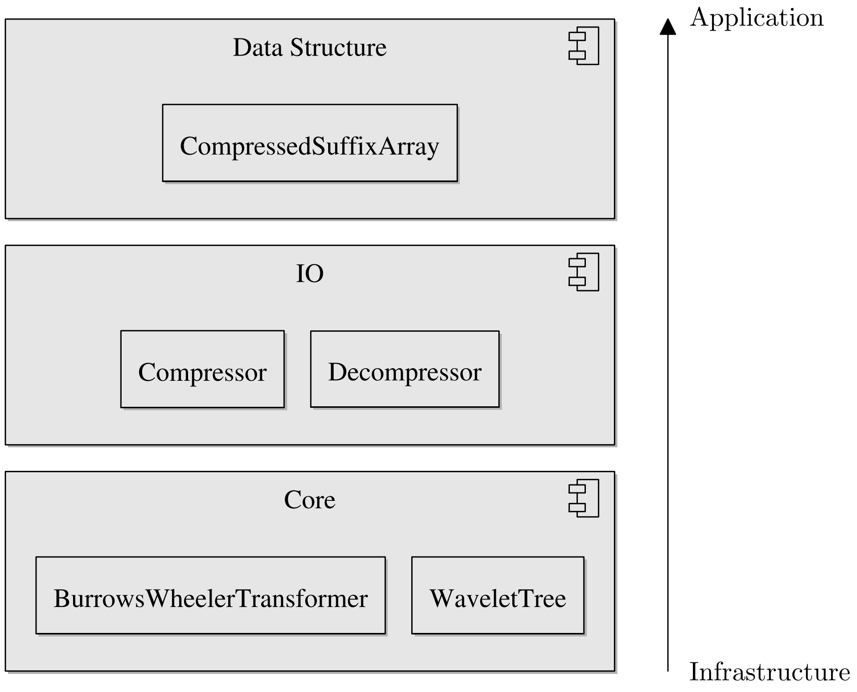

Figure 3.1: Architecture. Framework is decomposed into three layers, which are “Core”, “IO”, “Data Structure” from infrastructure to concrete applications.

## 3.1 Core

“Core” is the infrastructure layer of compression and decompression, which includes following facilities. In the unified framework mentioned in the preceding chapters, “Core” is responsible for BWT, and compressing BWT using high order empirical entropy model.

*TODO*: Core detailed design. How to partition BWT; How to compress BWT; How to encode empirical entropy model; How to support indexing.

## 3.2 IO

Concept stream is widely used in IO operations. A stream is an abstraction that represents a device on which input and ouput operations are performed. A stream can basically be represented as a source or destination of characters of indefinite length. Clients may read data from stream (input stream), or/and write data into stream (output stream).

The compression and decompression can be viewed as stream `Filters`. `Filters` are used to modified character sequences. For example, somebody might use a filter to replace all instances of one word with another, to convert all alphabetic characters to lower case or to compress, decompress a document.

`Filters` can compose a chain (or pipeline), and data are transferred from one to another. There exists two basic categories of `Filters`, which are named `InputFilters` and `OutputFilters`. `InputFilters` represent a "pull" model of filtering: a source of unfiltered data is provided, and the `Filter` is expected to generate a certain number characters of the filtered sequence (see Figure 3.2). The filtered sequence is generated incrementally, meaning that to filter a given character sequence the Filter typically must be invoked several times. `OutputFilters` represent a "push" model of filtering: a sequence of unfiltered characters and a sink are provided, and the `Filter` is expected to filter the characters and write them to the sink (see Figure 3.3). Like `InputFilters`, `OutputFilters` also process data incrementally.

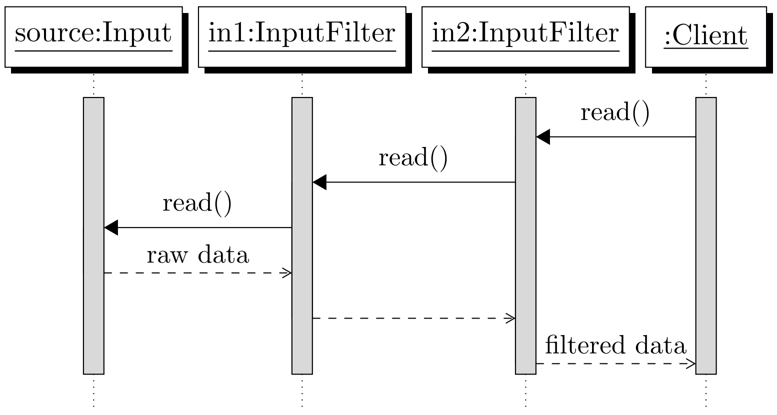

Figure 3.2: `InputFilter` Sequence Diagram

Based on the layer "Core", compression and decompression can implement the interface of `Filter` as in figure 3.4. Client just reads from an instance of `Input` and writes to an instance of `Output`, and don't cares the details of compression or decompression.

## 3.3 Data Structure

The layer builds data structure based on the compression and decompression framework. Because of the relationship between BWT and Suffix Array, Compressed Suffix Array is

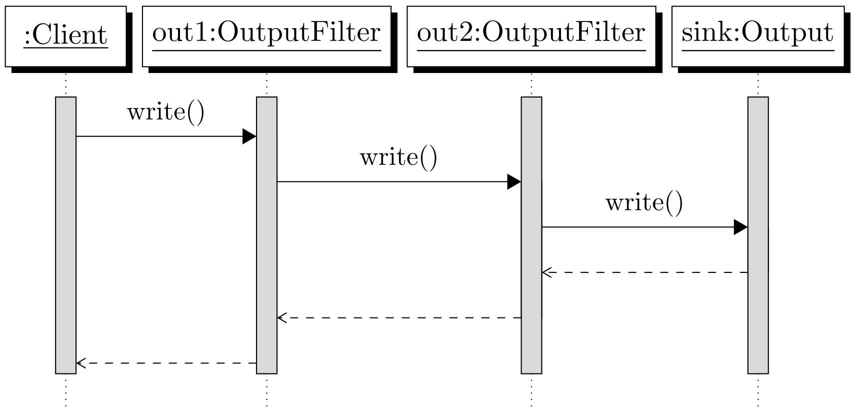

Figure 3.3: `OutputFilter` Sequence Diagram

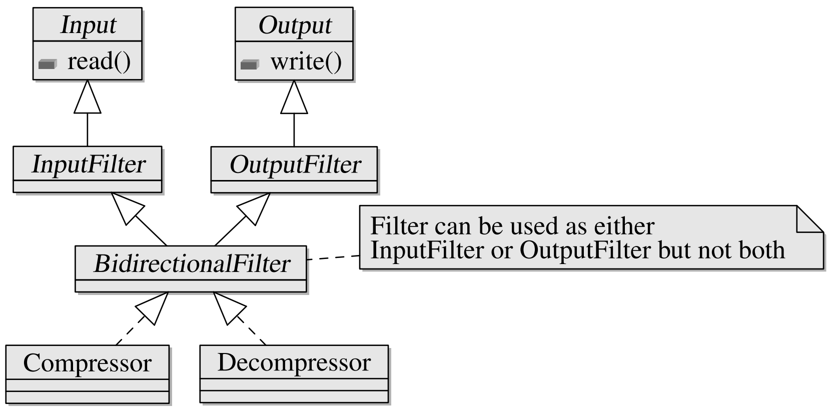

Figure 3.4: IO Layer Class Diagram

one of the strait forward application.

Compressed Suffix Array manipulates compressed text using the `Filter Compressor` and `Decompressor`. Client can still use the Compressed Suffix Array to search substring efficiently.

# Appendix

# A Schedule

Table A.1: *iZENElib* Data Compression Schedule

| Milestone | Start | End | In Charge | Description | Status |
|---|---|---|---|---|---|
| 1 Read the existing tr of *iZENElib* | 2009-03- 05 | 2009-03- 05 | Ian | Read tr to understand the objectives of *iZENElib* and the architecture, interface conventions, and *etc.* | Finished |
| 2 Go though existing source codes of *iZENElib* | 2009-03- 06 | 2009-03- 06 | Ian | Go though codes to get familiar with the coding style and the layout of the whole project. | Finished |
| 3 Requirements Analysis | 2009-03- 09 | 2009-03- 10 | Ian | Background research and confirm the objectives and outcomes of this project. | Finished |
| 4 Research | 2009-03- 11 | 2009-03- 17 | Ian | Survey existing solutions, papers as reference and have a brainstorm. | Finished |
| 5 Design | 2009-03- 18 | 2009-03- 24 | Ian | Architecture design. Detailed design for core components. Describe the design in TR. | Finished |
| 6 Implementation | 2009-03- 25 | 2009-04- 01 | Ian | Implement all functionalities. | Ongoing |
| 7 Test | 2009-04- 02 | 2009-04- 03 | Ian | Unit test and integrated test. | Not Started |
| 8 Report | 2009-04- 07 | 2009-04- 08 | Ian | Make conclusion and refine TR. | Not Started |

# Bibliography

[1] R. Grossi, A. Gupta, and J.S. Vitter. An algorithmic framework for compression and text indexing. *submitted for publication*, 2007.

# Index