Technical Report

iZENElib Tolerant Retrieval

Ian Yang

May 25, 2009

iZENEsoft(Shanghai) Co., Ltd

Abstract

String processing and searching tasks are at the core of modern web search, information retrieval and data mining applications. Many of these tasks may be implemented by basic algorithmic primitives which involve a large dictionary of strings having variable length. Such string dictionary problem is also known as the Tolerant Retrieval problem in the research literature.

This report first surveys existing Tolerant Retrieval solutions. The compressed permuterm index solves the problem in optimal query time, i.e. time proportional to the length of the searched pattern, and space close to the k-th order empirical entropy of the indexed dictionary. The index supports several types of wild-card queries over the string dictionary by performing a backward search.

The compressed permuterm index is implemeted in the project. This report presents the design outline and implementation details. To validate the performance, several experiments are designed to evaluate the algorithm.

Contents

1 Introduction 4

2 Background 6

3 Compressed Permuterm Index 7

A Schedule 9

Bibliography 10

Index 11

1 Introduction

String processing and searching tasks are at the core of modern web search, IR and data mining applications. Most of such tasks boil down to some basic algorithmic primitives which involve a large dictionary of strings having variable length. Typical examples include: pattern matching (exact, approximate, with wild-cards, ...), the ranking of a string in a sorted dictionary, or the selection of the i-th string from it. Such string dictionary problem is also known as the Tolerant Retrieval problem in the research literature[3].

The problem is defined in [1] as followers.

Let $\mathcal{D}$ be a sorted dictionary of m strings having total length n and drawn from an arbitrary alphabet Σ. The tolerant retrieval problem consists of preprocessing $\mathcal{D}$ in order to efficiently support the following WildCard(P) query operation: search for the strings in D which match the pattern $P \in (\Sigma \cup \{*\})^+$. Symbol $*$ is the so called wild-card symbol, and matches any substring of Σ . In principle, the pattern P might contain several occurrences of ; however, for practical reasons, it is common to restrict the attention to the following significant cases:

- Membership query determines whether a pattern $P \in \Sigma^+$ occurs in $\mathcal{D}$. Here P does not include wild-cards.
- Prefix query determines all strings in $\mathcal{D}$ which are prefixed by string α. Here $P = \alpha*$ with $\alpha \in \Sigma^+$.
- Suffix query determines all strings in $\mathcal{D}$ which are suffixed by string β. Here $P = *\beta$ with $\beta \in \Sigma^+$.
- Substring query determines all strings in D which have γ as a substring. Here $P = *\gamma*$ with $\gamma \in \Sigma^+$.
- PrefixSuffix query is the most sophisticated one and asks for all strings in D that are prefixed by α and suffixed by β. Here $P = \alpha * \beta$ with $\alpha, \beta \in \Sigma^+$.

There are two classical approaches to string searching: Hashing and Tries. As analyzed in [1], Hashing supports only the exact Membership query. Tries are more powerful in searching than hashing, but they fail to provide an efficient solution to the PrefixSuffix query. Although it can be resolved by building two tries, one storing the strings of $\mathcal{D}$ and the other storing their reversals. But this solution is space consuming.

The *Permuterm index* is a time-efficient and elegant solution to the tolerant retrieval problem above. It is presented in [2]. The idea is to take every string $s \in \mathcal{D}$, append a special symbol \$, and then consider all the cyclic rotations of $s\$$. The dictionary of all rotated strings is called the *permuterm dictionary*, and is then indexed via data structure that supports prefix-searches, *eg.* , the trie. The key to solve the PrefixSuffix query is to rotate the query string $\alpha * \beta\$$ so that the wild-card symbol appears at the end, namely $\beta\$\alpha*$. Finally, it suffices to perform a prefix-query for $\beta\$\alpha$ over the permuterm dictionary. As a result, the Permuterm index allows to *reduce any query of the Tolerant Retrieval problem on the dictionary $\mathcal{D}$ to a prefix query over its permuterm dictionary.*

The limitation of this elegant approach relies in its space occupancy, as "its dictionary becomes quite large, including as it does all rotations of each term."[3]. In practice, one memory word per rotated string (and thus 4 bytes per character) is needed to index it, for a total of $\Omega(n \log n)$[1] bits.

Ferragina and Venturini presents a *compressed* permuterm index in their paper [1]. It solves the tolerant retrieval problem in optimal query time, *i.e.*, time proportional to the length of the queried string, and space close to the k-th order empirical entropy of the dictionary $\mathcal{D}$. The latter is an information-theoretic lower bound to the output size of any compressor that encodes each symbol of a string with a code that depends on the symbol itself and on the k immediately preceding symbols. Known effective compressors are `gzip`[2] and `bzip2`[3].

The compressed permuterm index is based on a variant of the Burrows-Wheeler Transform here extended to work on a dictionary of strings of variable length. They have experimented their solution over various large dictionaries of URLs, hosts and terms, and compare it against some classical approaches to the Tolerant Retrieval problem such as tries, front-coded dictionaries, and `ZGrep`[4]. Experiments show that tries are much space consuming, and `ZGrep` is too slow to be used in any applicative scenario. Front-coding is the best known approach in terms of time/space trade-off. The compressed permuterm index improves the space occupancy of front-coding by more than 50% in absolute space occupancy, resulting close to `gzip` and `bzip2`. The query time is comparable to front-coding, taking few μ-secs per searched character. The compressed permuterm index is also flexible, it is possible to trade query time for space occupancy, thus offering a plethora of solutions for the Tolerant Retrieval problem which may well adapt to different applicative scenarios.

[1]Throughout this report all logarithms are taken to the base 2, whenever not explicitly indicated, and it is assumed that $0 \log 0 = 0$.
[2]Available at http://www.gzip.org
[3]Available at http://www.bzip.org
[4]A `grep` over `gzip`-ed files

2 Background

3 Compressed Permuterm Index

Appendix

A Schedule

Table A.1: *iZENElib* Data Compression Schedule

	Milestone	**Start**	**End**	**In Charge**	**Description**	**Status**
1	Requirements Analysis	2009-05- 19	2009-05- 19	Ian	Confirm the requirements of the project.	Finished
2	Research	2009-05- 20	2009-05- 22	Ian	Understand the requirements and survey related works and essential techniques for the project. Make decision of the solutions used in project.	Ongoing
3	Design	2009-05- 25	2009-05- 27	Ian	Program design for the solution.	Not Started
4	Implementation	2009-05- 31	2009-06- 2	Ian	Implement the data structure and integrate into izenelib.	Not Started
5	Test	2009-06- 03	2009-06- 04	Ian	Unit test, smoke test and integrated test	Not Started
6	Report	2009-06- 08	2009-06- 10	Ian	Evaluate the data structure & update TR	Not Started

Bibliography

[1] Paolo Ferragina and Rossano Venturini. Compressed permuterm index. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 535–542, New York, NY, USA, 2007. ACM.

[2] Eugene Garfield. The permuterm subject index: An autobiographic review. *JASIS*, 27(5-6):288–291, 1976.

[3] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

Index

burrows-wheeler transform, *see* BWT
BWT, 5

compressed permuterm index, 5

hash, 4

permuterm dictionay, 4
permuterm index, 4

tolerant retrieval, 4
trie, 4

wild-card, 4