# FAST CALCULATION OF N-GRAM-BASED PHRASE SIMILARITY

by

Zichu Ai

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
December 2017

© Copyright by Zichu Ai, 2017

*To dear professors who helped me with this project.*

*To my friends who provided help when I needed.*

*To my parents who are always supportive.*

# Table of Contents

**List of Tables** . . . v

**List of Figures** . . . vi

**Abstract** . . . vii

**Acknowledgements** . . . viii

**Chapter 1 Introduction** . . . 1

1.1 Problem Statement . . . 1

1.2 Phrase Similarity Calculation . . . 2

1.3 Contributions . . . 5

1.4 Thesis Document Structure . . . 5

**Chapter 2 Related Work** . . . 6

2.1 Word and Phrase Relatedness Using Google N-Gram Corpus . . . 6

2.2 N-gram indexing . . . 6

2.3 Efficient Integer Sequence Compression . . . 10

**Chapter 3 High Performance Computational Framework for Phrase Relatedness based on TrWP** . . . 12

3.1 Overview . . . 12

3.2 Pre-processing . . . 14

3.3 Mapping N-Grams to Phrase Indexes . . . 15

3.4 Mapping Target Phrase Indexes to Frequencies . . . 17

3.5 Mapping Target Phrase Indexes to Phrase Contexts . . . 18

3.6 Intersection . . . 32

3.7 Overall design . . . 38

**Chapter 4 Evaluation** . . . . . . . . . . **39**

4.1 Experiment Setup . . . . . . . . . . 39

4.2 Evaluation of the *gram-indexer* . . . . . . . . . . 39

4.3 Evaluation of the *gram-frequency-indexer* . . . . . . . . . . 41

4.4 Evaluation of the *gram-context-indexer* . . . . . . . . . . 42

4.5 Evaluation of Overall Performance in Speed . . . . . . . . . . 44

**Chapter 5 Conclusion** . . . . . . . . . . **49**

**Bibliography** . . . . . . . . . . **51**

**Appendix A Framework Description** . . . . . . . . . . **55**

## List of Tables


1.1 Examples of target phrase and phrase context . . . . . . . . . . 3

1.2 Example of binary vector . . . . . . . . . . . . . . . . . . . . . 4

2.1 Existing N-gram indexing solutions . . . . . . . . . . . . . . . . 7

2.2 Baseline of encoding approaches classified by data alignment . 10

3.1 Components of key collection, with the number of $n$-grams and size of file . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

3.2 Number of context arrays and file size represented as integer sequences . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

3.3 Word-aligned codecs . . . . . . . . . . . . . . . . . . . . . . . 30

4.1 Comparison of different indexing approaches in the aspect of construction time, retrieval time per query and memory cost in GB . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

4.2 Comparison of retrieval times and size of different *gram-frequency-indexer* implementation. . . . . . . . . . . . . . . . . . . . . . . 41

4.3 Comparison of different bit-aligned methods and differential coding approaches applied on *gram-context-indexer* . . . . . . . . . 42

4.4 Comparison of different differential coding approaches with most compact encoding method applied on *gram-context-indexer* . . 42

4.5 Comparison of different byte-aligned methods and differential coding approaches applied on *gram-context-indexer* . . . . . . . 43

4.6 Comparison of different byte-aligned methods and differential coding approaches applied on *gram-context-indexer* . . . . . . . 45

4.7 Comparison of different intersection methods with uncompressed context arrays. . . . . . . . . . . . . . . . . . . . . . . . . . . 45

4.8 Comparisons of framework with most compact storage and fastest calculation speed. . . . . . . . . . . . . . . . . . . . . . . . . 46

## List of Figures


| | | |
|---|---|---|
| 3.1 | Overview of structure and workflow | 13 |
| 3.2 | Pre-processing for generating required data set | 14 |
| 3.3 | Tiered *gram-indexer* | 16 |
| 3.4 | Expected memory cost of *gram-frquency-indexer* | 18 |
| 3.5 | Expected 32-bit aligned tiered *gram-frequency-indexer* | 18 |
| 3.6 | Structure of *gram-context-indexer* | 20 |
| 3.7 | Coding scheme of using SIMD operation *pshufb* (*Variant-G8IU*) | 26 |
| 3.8 | Coding scheme of Variant-Byte | 28 |
| 3.9 | Coding Scheme of *Variant-GB* | 28 |
| 3.10 | Coding Scheme of *Variant-G8IU* | 29 |
| 3.11 | Binary Packing with 6-bit frames and 8-bit frames | 30 |
| 3.12 | Patched Frame of Reference | 31 |
| 3.13 | Scalar intersection of monotonic integer sequence | 33 |
| 3.14 | Overall structure of phrase relatedness calculation framework | 38 |
| 4.1 | Tiered gram-indexer with sequential secondary key | 40 |
| 4.2 | Distribution of context lengths | 43 |
| 4.3 | Length distribution of evaluated context arrays | 45 |
| 4.4 | Break-down in time with different combinations of each module | 47 |
| A.1 | Framework UML | 56 |
| A.2 | Calling & Invoking Sequence | 57 |

## Abstract

*Text Relatedness Using Word and Phrase Relatedness Method* (TrWP) is a text relatedness measure that computes semantic similarity between words and phrases utilizing aggregated statistics from the Google Web-1T corpus. The phrase similarity computation in TrWP has significant overhead in time and memory cost, making TrWP inefficient for practical scenario with massive queries. This thesis presents an in-memory computational framework for TrWP, which optimizes the calculation process by efficient indexing and compact storage using perfect hashing, parallelism, quantization and variable length encoding. Using the Google Web 1T 5-gram corpus, we demonstrate that the fastest computational speed of our framework reaches 4098 queries per second.

# Acknowledgements

First, I'd really like to show special thanks to my supervisors: Dr. Norbert Zeh and Dr. Abidalrahaman Mohammad for offering me the chance of knowing what research is and what attitude should I have for challenges, and most importantly, patience. And I'd like to show thanks to Dr. Evangelos Milios for his useful comments and Dr. Vlado Keselj for being my reader. And I'd like to show special thanks to Jie Mei, without whose help I would not have been able to learn how to write a good paper and manage large projects. I'd like to show thanks to my friend Xiang Zhang, Yang Wang and Sitong Chen for their inputs when I needed.

Thanks to my mom and dad for offering me the chance of having this kind of unique experience.

# Chapter 1

# Introduction

## 1.1 Problem Statement

Text relatedness is an important research topic in the field of Natural Language Processing (NLP) that is useful for many tasks, including plagiarism detection [31], document classification [7], and machine translation [24]. Text relatedness algorithms work by capturing semantic information from text corpora. The availability of large corpora, like Google-Web 1T [6], benefits text relatedness algorithm by providing more semantic information. The *Google Trigram Method* (GTM) takes the advantage of the Google Web-1T corpus, and it has been demonstrated as an effective algorithm for calculating word relatedness and text relatedness [20]. *Text relatedness using word and phrase relatedness* (TrWP) [28] improves GTM by recognizing that semantic units that capture related concepts may be represented by phrases composed of two or more words. Phrases (of one or two words) sharing similar semantic information are recognized by their occurrences in the same contexts. The introduction of phrase contexts improves the accuracy of phrase relatedness calculations expressed by the Pearson correlation, but it requires a large amount of memory to manipulate and analyze those phrase contexts, so storage space is a problem. A document similarity calculation may trigger hundreds of phrase similarity queries, so the query speed is also crucial. Plenty of work has been done based on efficient file-system based approaches [18, 8]. However, disk I/O overhead restricts the retrieval speed of these methods, thus making TrWP impractical for calculating the text relatedness for large text data sets and answering queries on the fly.

In recent years, in-memory indexing has been demonstrated to be an effective approach for large text corpora [25, 16, 17]. However, in-memory indexing leads to conflicting objectives: storing large text corpora in memory requires them to be represented compactly while fast queries require the support of efficient data structures, which incur a certain space overhead. In the case of TrWP, the high memory cost of

the context array is the main challenge to be addressed. Thus our primary objective is to design efficient data structures to support fast calculation and a very important secondary objective is to make the data structure as compact as possible.

### 1.2 Phrase Similarity Calculation

TrWP extends GTM and shows improvement in accuracy of calculating phrase relatedness. In general, TrWP achieves the improvement by introducing bigram-bigram relatedness, and unigram-bigram relatedness during phrase similarity calculation and this similarity introduction are performed by using the notion of phrase context, which is fully discussed in this section later.

Phrase similarity is calculated based on the phrase context generated from Google Web-1T 5-gram corpus. Examples of n-gram contexts are shown in Table 1. Target phrase could be either one-word or two-word. And we have three kinds of phrase contexts: a *left-context* consists of two unigrams immediately before a target phrase, a *right-context* consists of two unigrams right after target phrase, while a *left-right context* consists of a unigram immediately before target phase and another unigram right after target phrase. If a pair of phrase contexts share the same content but they are correlated with different target phrase, we say we find a common phrase context. Difference in type and frequency is acceptable. In order to clearly formalize the working flow of TrWP, we declare a collection of abbreviations and symbols as follows: assume we have a pair of target phrases $P_1$ and $P_2$. For each target phrase, we may find a sequence of correlated phrase contexts $A_{P_1} = \{C_1^1, C_1^2, \dots C_1^n\}$ with $n$ elements and $A_{P_2} = \{C_2^1, C_2^2, \dots C_2^m\}$ with $m$ elements for $P_1$ and $P_2$ respectively from either trigrams or fourgrams, where $C$ represents phrase context. We need to extract common phrase contexts $AC = \{CC_1, CC_2 \dots CC_k\}$ by finding intersection of $A_{P_1}$ and $A_{P_2}$, where $CC$ stands for common context. For example: for the trigram "*nice* bachelor *person*" and the fourgram "behaved as *nice person*", we have target phrases "bachelor" and "behaved as". They both have context with same content "*nice person*" but of different type. Since the contents of context are the same, "*nice person*" is still defined as common context. We use $F_1()$ to query the frequency of contexts $AC_{P_1}$, $F_2()$ to query the frequency of contexts in $AC_{P_2}$ and $F()$ to query the frequency of target phase.

| **Context Category** | **Contexts in Google n-gram** |
|---|---|
| *rightcontext* | bachelor *lives alone* |
| *left-right context* | *nice* bachelor *person* |
| *left context* | *good looking* bachelor |
| *right context* | large number *of files* |
| *left-right context* | *very* large number *generator* |
| *left context* | *show someone* large number |

Table 1.1: Examples of target phrase and phrase context

Notion of Phrase Context. The table shows three context types, based on the position of the target phrase in the n-gram. Words in italic font represent the context while words in normal font represent target phrase.

First, we need to extract $A_{P_1}$ for $P_1$ and $A_{P_2}$ for $P_2$. Second, we perform lexical pruning on $A_{P_1}$ and $A_{P_2}$. Lexical pruning involves removal of some of the contexts that satisfy certain conditions. One condition is related to the presence of stopwords in a context: for *left-contexts*, those with stopword or punctuation at the left-most position should be removed, *right-contexts* with stopword or punctuation at the right-most position should be removed, and *left-right contexts* with stop words at both ends should also be removed. Third, we detect common contexts $AC$ from $A_{P_1}$ and $A_{P_2}$. Fourth, we perform statistical pruning on the common contexts. This means the following: (a) For each context of $AC_{P_1}$ and each context of $AC_{P_2}$, we collect their frequencies; (b) Compile the frequencies of all common contexts into two arrays, one for $P_1$ and one for $P_2$; (c) For each array, we calculate the strength of each common context pair based on the following sum-ratio strategy: assume the frequency of $P_1$ is $c_1$, the frequency of $P_2$ is $c_2$, the frequency of $CC$ correlated with $P_1$ is $cc_1$ and the frequency of $CC$ correlated with $P_2$ is $cc_2$, then we calculate the strength of $CC$ as $\frac{min(\frac{cc_1}{c_1},\frac{cc_2}{c_2})}{max(\frac{cc_1}{c_1},\frac{cc_2}{c_2})} \times (cc_1 + cc_2)$; (d) we calculate the average value $\mu$ and standard deviation $sd$ of all the strengths, common context pairs with strength value exceeding $\mu \pm sd$ should be removed.

Phrase relatedness strength is obtained by combining relatedness strengths calculated from identical contexts and all contexts of the phrase pairs, which is described as follows:

- **Step 1:** For each identical context pair, we calculate the ratio (minimum / maximum) between their frequencies, and multiply it with the sum of their

frequencies. Then we sum the results of all $k$ statistically pruned identical context pairs, to obtain the relatedness strength $RS$, in Eq. 1:

$$RS(P_1, P_2) = \sum_{i=1}^{k} \left[ \frac{min(F_1(CC_i), F_2(CC_i))}{max(F_1(CC_i), F_2(CC_i))} \cdot [F_1(CC_i) + F_2(CC_i)] \right] \quad (1.1)$$

where $F_1(CC_i)$ retrieves the frequency value of $CC_i$ whose target phrase is $P_1$ and $F_2(CC_i)$ retrieves the frequency value of $CC_i$ whose target phrase is $P_2$.

- **Step 2:** We compute the cosine similarity between all the non-pruned(identical and non-identical) phrase contexts of $P_1$ and $P_2$ using two boolean vectors $AB_{P_1}$ and $AB_{P_2}$. If one context appears in $A_{P_1}$ but not in $A_{P_2}$, corresponding element in $AB_{P_1}$ will be assigned 1 while in $AB_{P_2}$ it will be assigned 0:

$$cosSim(P_1, P_2) = \frac{AB_{P_1} \cdot AB_{P_2}}{||AB_{P_1}|| \cdot ||AB_{P_2}||} \quad (1.2)$$

For example: if we have target phrases $P_1$ "bachelor" with trigrams "former bachelor acquaintances","degree of bachelor", "number of bachelor" and $P_2$ "single man" with "former single man acquaintances", "property of single man", "number of single man". Then $AB_{P_1}$ and $AB_{P_2}$ will be described as follows:

| | former .. acquaintances | degree of | number of | property of |
|---|---|---|---|---|
| $AB_{P_1}$ | 1 | 1 | 1 | 0 |
| $AB_{P_2}$ | 1 | 0 | 1 | 1 |

Table 1.2: Example of binary vector

- **Step 3:** We combine the relatedness strength and cosine similarity generated from the context arrays, defined as $RSCOS$:

$$RSCOS(P_1, P_2) = RS(P_1, P_2) \cdot cosSim(P_1, P_2) \quad (1.3)$$

- **Step 4:** We define the normalized phrase relatedness ranging from 0 to 1 as the NGD [9] normalization of $RSCOS$, where $N$ stands for the total number of web documents used in Google-n-gram corpus:

$$NGD(RSCOS(P_1, P_2)) = e^{\{-2 \cdot \frac{max(\log F(P_1), \log F(P_2)) - \log(RSCOS(P_1, P_2))}{\log N - min(\log F(P_1), \log F(P_2))}\}} \tag{1.4}$$

where *F()* retrieves the frequency of queried target phrase.

## 1.3 Contributions

- We demonstrate that an in-memory representation of unigram and bigram contexts can be used to speed up the computation of phrase similarities.
- We implement a computational framework with careful engineering, integrated with perfect hashing, fast differential coding, variable length encoding of different schemes. And we provide a space-efficient representation of context arrays to guarantee that our framework fits in memory.
- We evaluate parallel methods based on Single instruction, multiple data (SIMD) instructions available on modern microprocessors to speed up the calculation of phrase relatedness.

## 1.4 Thesis Document Structure

The remainder of this thesis is organized as follows:

In Chapter 2, we review the relevant literature.

In Chapter 3, we introduce algorithms applied on our high-performance framework, and we provide a high-level description of it. Then we present the detailed description of each component including an analysis of bottlenecks and a study of indexing approaches such as perfect hashing, variable-length coding, and SIMD acceleration.

In Chapter 4, we evaluate each step of our computational framework and compare its performance with other open-source language model tool kits, like SRILM [33]. We compare different approaches for accelerating each step and we identify the most efficient combination.

In Chapter 5, we discuss future work and offer conclusion.

# Chapter 2

# Related Work

In this chapter, we review the background of phrase relatedness calculation based on n-grams. Besides, we find our work highly correlates with some well-studied topics in information retrieval, including n-gram indexing and optimization on inverted index. Thus we review works related to the two topics in this chapter.

## 2.1 Word and Phrase Relatedness Using Google N-Gram Corpus

$$RT(\omega_1,\omega_2)=\begin{cases}\frac{\log\frac{\mu_T(\omega_1,\omega_2)C_{max}^2}{C(\omega_1)C(\omega_2)\min(C(\omega_1),C(\omega_2))}}{-2\times\log\frac{\min(C(\omega_1),C(\omega_2))}{C_{max}}} & if \quad \log\frac{\mu_T(\omega_1,\omega_2)C_{max}^2}{C(\omega_1)C(\omega_2)\min(C(\omega_1),C(\omega_2))}>1\\ \frac{\log 1.01}{-2\times\log\frac{\min(C(\omega_1),C(\omega_2))}{C_{max}}} & if \quad \log\frac{\mu_T(\omega_1,\omega_2)C_{max}^2}{C(\omega_1)C(\omega_2)\min(C(\omega_1),C(\omega_2))}\leq 1\\ 0 & if \quad \mu_T(\omega_1,\omega_2)=0\end{cases} \tag{2.1}$$

GTM [19] is an unsupervised text relatedness algorithm. It uses unigrams and trigrams from Google's Web 1T n-gram corpus [6] for word relatedness calculation. The relatedness is calculated by Equation 2.1, where $C(\omega)$ stands for count of unigram $\omega$, $\mu_T(\omega_1,\omega_2)$ stands for mean frequency of trigrams either start with $\omega_1$ and end with $\omega_2$ or start with $\omega_2$ and end with $\omega_2$, $C_{max}$ represents maximum frequency among all unigrams.

Text relatedness based on Words and Phrases (TrWP) [28] extends GTM. It introduces word-phrase relatedness and phrase-phrase relatedness by taking advantage of phrase contexts, again extracted from the Google Web 1T n-gram corpus, which produce more accurate results but lead to high memory cost due to large data representing phrase contexts.

## 2.2 N-gram indexing

Generally speaking, we have three kinds of starting points that can be used for n-gram indexing: (a) file-based indexing, (b) relational databases, and (c) specialized systems

| Name | Year | Purpose | Core DataStructure | Is Code Released |
|---|---|---|---|---|
| SRILM | 2002 | Frequency Query | Trie | www.speech.sri.com/projects/srilm/ |
| RandLM | 2007 | Machine Translation | BloomFilter | sourceforge.net/projects/randlm |
| LOUDS | 2009 | Language Model Compression | Trie(BitVector) | github.com/tarowatanabe/expgram |
| MPHR | 2010 | Frequency Query | Perfect Hashing + BitVector | No |
| BerkeleyLM | 2011 | Machine Translation | Trie(SortedArray/HashTable) | github.com/adampauls/berkeleylm |
| KenLM | 2011 | Machine Translation | Trie(SortedArray/HashTable) | kheaeld.com/code/kenlm |
| Marisa | 2011 | General String Dictionary | Patricia Trie | github.com/s-yata/marisa-trie |
| Elias-Fano Trie | 2017 | Machine Translation | Trie(BitVector) | github.com/jermp/tongrams |

Table 2.1: Existing N-gram indexing solutions

[14]. Our baseline implementation uses file-based indexing, which is relatively slow. And relational databases are mostly based on file-indexing as well. In memory, two kinds of data structures are commonly used for the indexing of large text corpora, namely hash tables and trie [25].

We do a brief review of works related to efficient in-memory n-gram indexing in Table 2.1. In-memory indexing of n-grams for fast retrieval was first used in SRILM [33], an n-gram language model toolkit. SRILM uses a naive word-level trie with a hash table in each layer. Each hash table contains successive unigrams of a common prefix as key and corresponding frequency values as satellite data. *MPHR* [16] indexes the whole Google Web1T corpus based on perfect hashing and n-grams represented as fingerprints. Fingerprints are generated from the lower $m$ bits of hash value generated from Murmurhash[1]. Although fingerprints are kept to avoid false positives caused by perfect hashing, it cannot avoid false positives caused by Murmurhash. That is, different key may result in a same lower $m$ bits in their hash value. The extension on fingerprint length may reduce false positive ratio but will lead to more memory cost. MPHR also realize the values of satellite data are rather scalar, so memory can be saved by representing satellite data using *rank*, which is the offset of corresponding value in an auxiliary array, named *unique value array*, which keeps unique values of satellite data. The resulting framework achieves compact storage with around 3 bytes per n-gram and 1.97 microseconds per query. *BerekelyLM* [25] introduced a novel indexing structure based on a trie-based structure. Since a pointer-based trie leads to high memory cost because of pointers, BerkeleyLM uses array offsets to indicate the ids of n-grams. Vocabularies are kept in string while n-grams containing more than one word are kept in numerical representation. N-grams sharing the same suffix are kept in blocks of sorted arrays. Ids of unigrams are retrieved from the offset in the

[1]https://sites.google.com/site/murmurhash/

vocabulary array while the prefixes are retrieved from the array offset of *n-1* grams. Layers of the trie-based structure are represented using two kinds of alternatives: sorted array, in which loop-up operation is performed by binary search, and hash map, in which loop-up operation is performed by hash table look-up. BerkeleyLM firstly introduce variable length encoding in solutions listed in Table 2.1 for compact storage. In addition, BerkeleyLM notice queries are often highly repetitive, so it introduces a key-value cache based on *direct-mapping* (each key has exactly one address), so that repetitive queries can be answered by the cache rather than the trie structure. Fastest retrieval speed by BerkeleyLM achieves 0.139 microseconds per query. *KenLM* [17] further extend BerkeleyLM, introducing optimizations in *searching* and *bitpacking*. KenLM has been demonstrated to be more efficient than BerkeleyLM and SRILM. It stores keys as their hash values using Murmurhash, guaranteeing an even distribution of hash values like BerkeleyLM. But it performs *interpolation search* [26] instead of binary search to look for corresponding unigram in each layer, thus reducing time complexity of searching corresponding element from $O(\log n)$ to $O(\log \log n)$. However, *interpolation search* is effective only for long sequences, it will be less competitive when dealing with short sequences. Because task of machine translation requires fast comparison of probabilities, KenLM use *binning method* [13] to quantize sorted floating probability values into bins of equal size and it uses mean value of each bin as representative, thus introducing scarification of accuracy since probability values lies in the same bin cannot be properly compared. KenLM is fast with 0.32 microsecond per query and 14.77 byte per n-gram. *LOUDS trie* [35] takes advantage of succinct coding method *LOUDS* [21], which represents a trie with $m$ nodes using $2m + 1$ bits. It supports layer navigation by efficiently perform a select operation on a bit vector. Associated values are compressed using variant-byte encoding method with an auxiliary bit vector. The bit vector indicates the boundaries of byte values by setting 1 to the $i_{th}$ position associated with the last byte in a variant-byte represented integer, thus guaranteeing random access of each associated value by retrieving bytes between $Select_i + 1$ and $Select_{i+1} + 1$. The comprehensibility of LOUDS reaches 2.40 bits for each n-gram, however, the paper does not contain evaluation on query speed. Recently, Giulio compared all available n-gram indexing frameworks, coming up with a framework named Elias-Fano Trie with excellent speed-memory trade-off.

*Elias-Fano Trie* [27] implements a similar trie structure like BerkeleyLM and KenLM but with different coding approach, which takes advantage of *Elias-Fano Coding* [12] by recognizing the availability of random access on compressed data with auxiliary data structures for efficiently answering $Select_1(i)$. It takes advantage of many optimizations introduced in other works mentioned in this section, like *unique value array* [16]. It reaches 1.35 microseconds per query but only requires 1.97 bytes per n-gram.

However, we notice the difference between those works and our objectives by recognizing the following properties:

- Most existing works are evaluated under a scenario different from ours. For example, BerkeleyLM and KenLM keep probabilities of n-gram as satellite information for machine translation scenarios. LOUDS Trie supports layer navigation for the purpose of finding successors and predecessors of query. However, key information, in our task, has different satellite information according to its role. For example, in our implementation of TrWP, we use n-gram indexing to unigrams, bigrams and phrase contexts, in which unigrams and bigrams are treated as target phrases but part of bigrams can be treated as phrase contexts as well. As a result, depending on the different interpretation of a bigram, we associate different satellite data. Thus we do not include satellite data in the n-gram indexing step.

- Trie structure is commonly chosen because in machine translation scenario, we require probabilities of contexts as feature to support the decision on hypotheses. However, in our case, we only care about correctly retrieving indexes. Additional overhead brought by layer navigation and satellite data could be ignored.

- We show no tolerance to false positives for the correctness of TrWP. Since we expect massive queries, a small probability of false positive may result in non-negligible false positives. Despite of [16, 25], other works mentioned in this section treat n-grams as digital tokens by default, however, we have to consider invalid queries. This rules out algorithms based on Bloom Filter or perfect hashing without additional steps to catch false positives.

- Succinct data structures show high compressibility, but they are not competitive

| Coding Family | Coding Name | Highlight |
| --- | --- | --- |
| Bit Aligned | Elias Family | • Treat data as bit stream.<br>• High compressibility.<br>• Hard to parallelize. |
| | Golomb/Rice | |
| | Block Coding | |
| Byte Aligned | Variant Byte | • Treat data as byte stream.<br>• Relatively lower compressibility.<br>• Easy to parallelize. |
| | Variant Group Byte | |
| | Variant G8IU | |
| | Variant G8CU | |
| Word Aligned | Simple-9 | • Try to store as many values as possible inside one word.<br>• Relatively higher speed because of data alignment.<br>• Available when integer sequence satisfy certain distribution. |
| | Simple-16 | |
| | Simple-8b | |
| Frame Aligned | Binary Packing | • Store data in sequences of fixed-length bit frames. |
| | PFor | |
| | AFor | |
| Succinct | Elias-Fano | • Random access on compressed data. |

Table 2.2: Baseline of encoding approaches classified by data alignment

in speed due to frequent bit manipulations, which conflicts with our objective of guaranteeing high retrieval speed. To use succinct data structures in our implementation, we would need a method that supports much faster *Select* operations on a bit vector than what are currently known.

## 2.3 Efficient Integer Sequence Compression

As we will see in this Chapter 3.5, managing phrase contexts is quite similar to the implementation of inverted indexes. Efficient implementation of both context arrays and inverted indexes require compact storage and support for fast intersections of integer sequences. Given that integer sequences are usually compactly stored as bit arrays, we review works related on architecture-dependent compression approaches and use of data-level parallelism to implement bitmap compression and inverted indexes. These approaches often exploit SIMD, which is a collection of instructions available on modern microprocessors. Plenty works have been done but the majority of them are optimized based on algorithms listed in Table 2.2.

Generally speaking, the compressed representation of integer sequence consists two components: *descriptor* (or *selector*, *control pattern*), which is used to describe the length of the natural representation of integers in bits, bytes, etc, and *payload* (or *data*), which carries the actual information of original integers. Lemire introduced S4-BP128-D4 [23], which is a compression scheme on 32-bit integer sequences. "D4" stands for differential coding, which computes deltas of every four integers. "BP"

stands for bit packing, which compresses integers into a sequence of fix-length bit frames, "128" represents the number of integers in each block and "S4" stands for 4-integer SIMD acceleration. It achieves 0.7 CPU cycles for the decoding process of each 32-bit integer. Another work of Lemire [22] focuses on engineering compression of long sequence of monotonically increasing 32-bit values, reaching a performance of 1600 million integer compressions and 2600 million integer decompressions per second. Byte-aligned coding [32] is a collection of compression schemes. Stepanov introduced a general approach for fast decoding of grouped byte-aligned compression algorithms (*Variant-GB, Variant-G8IU* and *Variant-G8CU*) based on the SIMD shuffling operation *pshufb* and the pre-construction of shuffle sequences. Trotman's work [34] optimizes word-aligned codings, also known as simple codings, by recognizing left-greedy strategy is not optimal dealing with some counter-examples. That is, the integer sequence is compiled in sequence of machine words successively, so some integers might be forced to be compiled in next machine word if it overflows remaining space in current one. However, same data sequences with different orders show different compressibility. In this work, *Simple-9* is optimized by performing dynamic programming on integer sequence to find the best order. However, the paper do not mention how to recover the original sequence if the order need to be considered. Schlegel [30] apply SIMD acceleration to *Elias* $\gamma$ coding by aggregating integers into blocks and using *shared* length information so SIMD can be applied on data blocks.

We also find differences between works discussed above and our work: The SIMD accelerated approaches for speeding up compression and decompression of monotonically increasing 32-bit integer sequences focus on very long sequences. However, sequence lengths in our framework vary significantly between 2-3 elements and up to 80,000 elements. The majority of context arrays in our work contains less than 5 elements, which makes SIMD based approaches not applicable or degraded. This makes it difficult to apply SIMD techniques to speed up compression and decompression in our framework. *Simple-9* and *Simple-16* [34] can only handle integer sequences with each element not exceeding $2^{28}$, but the codec is expected to handle larger integers since many frequency values in Google n-gram corpus exceed $2^{28}$.

# Chapter 3

# High Performance Computational Framework for Phrase Relatedness based on TrWP

Our primary objective is to build on the approach of TrWP to calculate phrase relatedness and engineer a framework supporting fast queries so that TrWP can be practical to massive queries. We do not change the fundamental method used by TrWP to compute phrase relatedness but ensure the method is implemented using efficient algorithms and data structure to achieve high throughput. These data structures are large, having to store information about the entire Google Web -1T corpus for unigrams, bigrams, trigrams and fourgrams. To be able to fit these data structures in memory, an important secondary objective is to make them as compact as possible.

## 3.1 Overview

As discussed in Chapter 2.2, we treat n-grams as numerical tokens and represent phrase contexts as integer sequences. Since bigrams can be interpreted as target phrases or phrase contexts, we expect auxiliary data structures to store satellite data. Thus we expect three components to finish the calculation: a *gram-indexer* that can convert target phases and phrase contexts into numerical tokens (we call this process "mapping n-grams to phrase indexes"), a *gram-frequency-indexer* that can associate frequency value with target phases (we call this process "mapping target phrase indexes to frequencies") and a *gram-context-indexer* that can associate context arrays with corresponding target phases (we call this process "mapping target phrase indexes to phrase contexts"). We present the overall structure and the working flow in Figure 3.1.

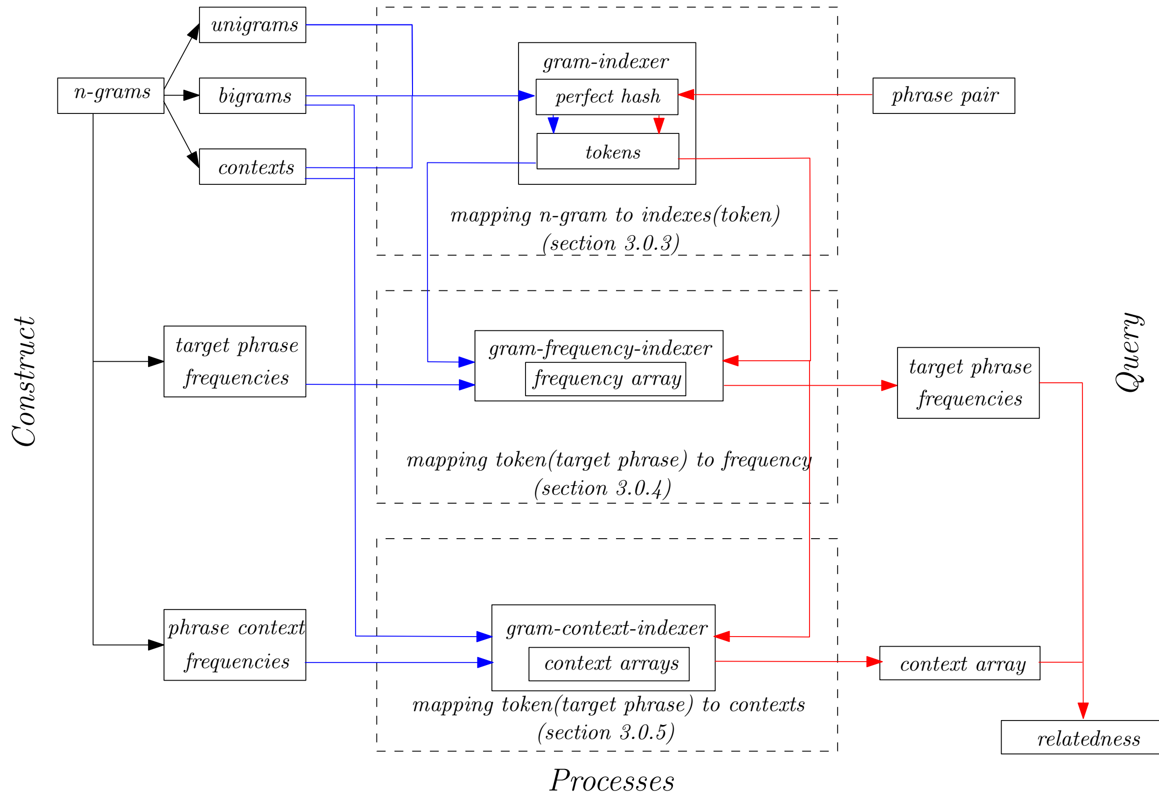



Figure 3.1: Overview of structure and workflow

| Dataset | Numbers of Distinct $n$-grams | File Size [GB] |
|---|---|---|
| keyset | 198,820,259 | 2.70 |
| unigram | 4,210,035 | 0.05 |
| bigram | 142,683,094 | 2.2 |
| context | 51,927,127 | 0.45 |

Table 3.1: Components of key collection, with the number of $n$-grams and size of file

| Dataset | Numbers of context arrays | File Size [GB] |
|---|---|---|
| trigram | 2,116,038 | 9.9 |
| fourgram | 41,296,941 | 12 |

Table 3.2: Number of context arrays and file size represented as integer sequences (including target phrase id, arrays of context id and frequency)

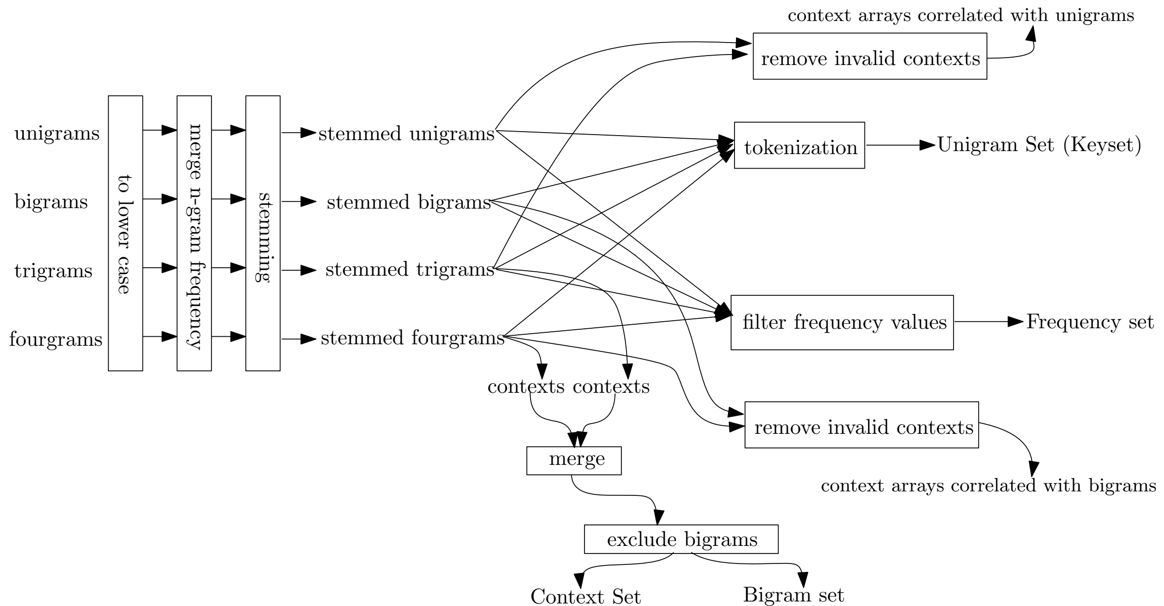



Figure 3.2: Pre-processing for generating required data set

## 3.2 Pre-processing

The size of the original Google Web-1T corpus is around 87 GB. TrWP is based on unigrams, bigrams, trigrams, and fourgrams. This potion of corpus takes 55GB. TrWP performs preprocessing, including transforming all n-grams into lower-case, stemming of all n-grams and merging frequency values of n-grams that become identical as a result. After this step we expect more contexts to be correlated with one target phrase, so we include more semantic information for each calculation and improves the quality of similarity results. Then we extract all context from trigrams and fourgrams beforehand. Ideally, Google unigrams can be treated as vocabularies and all phrase contexts are included in Google bigrams. However, we find quite a few unigrams that appears in Google bigrams, trigrams and fourgrams are not included in Google unigrams. In addition, *left-right-contexts*, which are split by target phrases, are less likely to be included in Google bigrams. Thus, we need to tokenize the whole Google unigram, bigrams, trigrams and fourgrams to get vocabularies (unigrams) and find all possible bigrams, including bigrams as target phrases and possible phrase contexts. Then we get a collection of n-grams which should be represented

by numerical ids. After pre-processing, the size of the whole dataset is shrunk to 23.17 GB. A detailed description of the data set is presented in Table 3.1 and Table 3.2. Keyset contains all unigrams, bigrams and phrase contexts, and n-gram files contain n-grams of different lengths with corresponding frequency. We find the size of unigrams, which serves as vocabularies, is 8,239,666, and the average length of each unigram is 14.4 characters, so we expect that the binary representation of unigrams won't exceed $\lceil \log_2(8,239,666) \rceil = 23bits$. In addition, we have 262,028,256 unique unigrams and bigrams, which we treat as the collection of target phrases and phrase contexts, so we guarantee that n-grams involved in our task can be represent using $\lceil \log_2(262,028,256) \rceil = 28bits$, which won't exceed the range of a 32-bit integer.

## 3.3 Mapping N-Grams to Phrase Indexes

N-gram indexing is applied to unigrams, bigrams and phrase contexts. As discussed in Chapter 2, we treat n-gram indexing and the representation of correlation between corresponding satellite data as two separate processes, so indexing is our only concern in this step. In order to quickly locate the grams during calculation, we assign each item of our keyset a numerical index for fast retrieval and data compression. We apply perfect hashing to ensure constant-time look-up operations and optimized data storage for false positive validation.

**Perfect hashing:** We apply the *Hash, displace and compress* algorithm [5] [1] for the generation of a minimal perfect hash function, which ensures constant time for the retrieval of indexes. A minimal perfect hash function $h$ for a set $S$ of $k$ keys ensures that each key in $S$ maps to a unique integer *ID* in the range $[0, k-1]$. However, if we apply $h$ to a key not in $S$, it also produces an integer in this range. To detect whether a given query key $x$ is in fact in $S$, we store the elements in $S$ in a string array $A$, storing element $x \in S$ at index $h(x)$ in the array. Given a query key $x$, we check whether $x \in S$ by testing whether $A(h(x)) = x$. If so, we report $h(x)$ as $x$'s *ID*, otherwise, we report that $x \notin S$. This naive approach [1] serves as a baseline for later optimization. And we can populate $A$ in parallel since perfect hashing guarantees no hash collision.

[1] C Minimal Perfect Hash Library: http://cmph.sourceforge.net/

**Reducing memory cost:** As discussed in [16], using hash value as fingerprints of n-grams won't eliminate false positives, so we can't avoid the string representation of n-grams in memory. We expect the memory taken by the string representation of unigrams to be 0.11GB, which is manageable. However, if we include all keys in string representation to avoid false positives, memory solely cost by strings is expected to be 4.2 GB, which deserves further optimization. We notice bigrams and contexts can be treated as the combination of unigrams, so they can be represented as a pair of unigram ids. Thus, we recognize two kinds of false positives. The first kind: for unigram queries, the perfect hash function returns a valid index for an invalid query. The second kind: for bigram queries, the bigram may not be in the data set even if the two unigrams it is composed of are in the data set. We expect less memory cost by designing a tiered implementation for bigram indexing. We first use the uigram indexer to map each of the two unigrams in the bigram to a 23-bit integer. Instead of the full bigram string, we use the concatenation of these two 23-bit integers, which is a 46-bit integer, as the representative of bigram. As discussed in section 3.2, average length of unigrams is 14.4 characters, so we expect 29.8 characters per bigram, which is 238.4 bits. Thus we approximately expect to save 80.7% memory for storing bigrams and contexts.

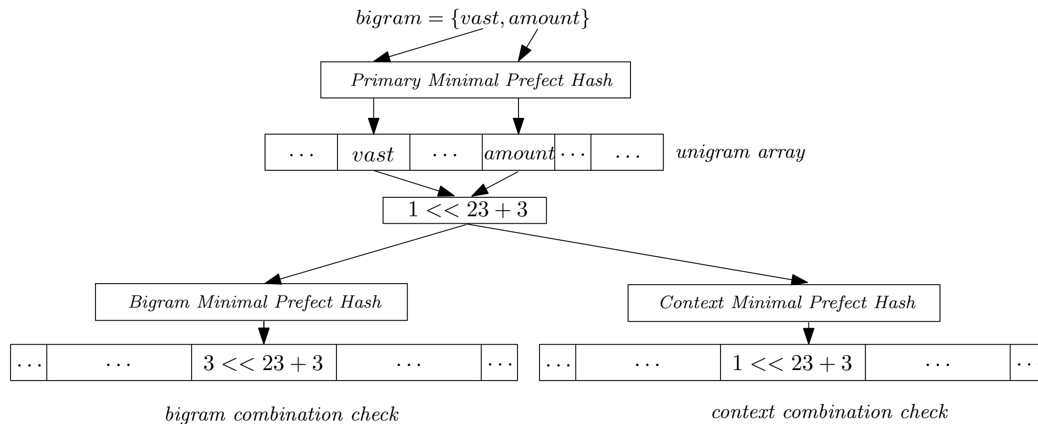



Figure 3.3: Tiered *gram-indexer*

*Note:* Tiered *gram-indexer*, assume we are querying the index of a phrase context "vast amount", we first check the availability of "vast" and "amount", then we check the availability of combination by left shifting the index of "vast" by 23 bits and add it to the index of "amount", which is a 46-bit flag. We use this flag to validate the combination of this two-word query.

**Final design:** We name the tiered data structure for mapping n-gram to indices as *gram-indexer* in Figure 3.3. We have unigram indexer consisted of a primary perfect hash function $F_u$ and a auxiliary string array $A_u$ with each unigram $k_u$ stored in $F_u(k_u)$. We use unigram indexer to retrieve the ids and check the first kind of false positives. For bigrams and contexts, we represent them using 46-bit keys consisted of two 23-bit keys representing corresponding unigrams. We keep a perfect hash $F_b$ and an array of 46-bit integers $A_b$ for keys of bigrams, and similar $F_c$ and $A_c$ for contexts. Then we check the second kind of false positive by checking the existence of the 46-bit key generated from query.

By using such a design, we can avoid false positives when mapping grams to indexes while achieving constant-time look up operations on average and storing the data fairly compactly. We can also guarantee that the indexes of unigrams, bigrams and contexts are distributed in the range $[0, |A_u| - 1]$ , $[|A_u|, |A_u| + |A_b| - 1]$ and $[|A_u| + |A_b|, |A_u| + |A_b| + |A_c| - 1]$, respectively.

### 3.4 Mapping Target Phrase Indexes to Frequencies

In the second step, we map the indexes of the target phrases to their corresponding frequencies. We have a baseline approach [1], which simply stores the frequency for each unigram and bigram in an integer array. The frequency of the unigram or bigram with id $k$ is stored at position $k$ in the array. To reduce memory usage of the frequency array, we observe that, while there are 146,893,129 keys whose frequencies need to be stored, only 482,265 unique frequencies occur. The largest frequency value in the Google Web 1T corpus is more than 95 billion, which requires 37 bits for storage, but most frequency values can be represented using 32 bits. We explored the Google Web 1T corpus and found all n-grams with frequency larger than $2^{32}$ happens to be stopwords or punctuations, which are guaranteed excluded from our data set.

In order to further optimize memory cost, we borrow the notion of *patch coding* [37] by designing a tiered data structure. We set a threshold $b$. Frequency values up to $2^b - 1$ are "small", values greater than or equal to $2^b$ are "large". The data structure now consists of two arrays: a main array $M$ and an escape array $E$. Each entry in $M$ uses $b + 1$ bits. If the first bit of $M$[k] is 0, then the $M$[k] is the frequency of the element with key $k$. Otherwise, the lower $b$ bits of $M$[k] are the index $i$ of an

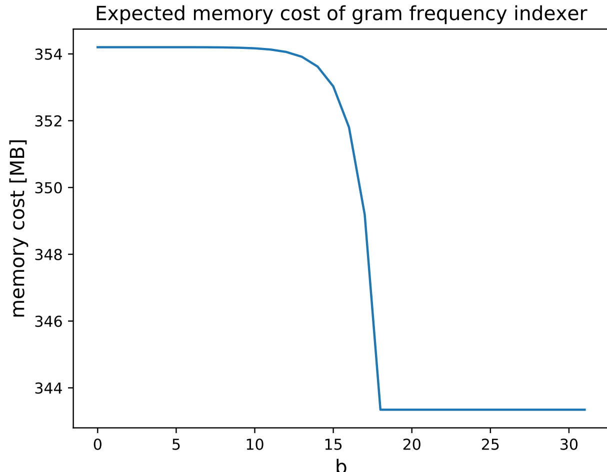



Figure 3.4: Expected memory cost of *gram-frquency-indexer*

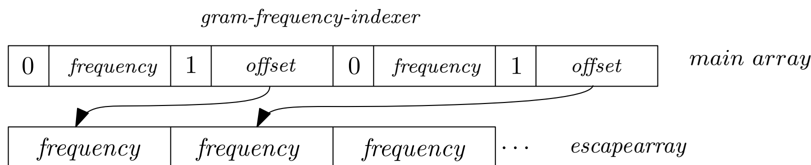



Figure 3.5: Expected 32-bit aligned tiered *gram-frequency-indexer*

entry in $E$: $E$[i] is the frequency of the element with key k in this case. $E$ thus stores all large frequencies using 31 bits per frequency value.

For a given choice of $b$, it is easy to determine the number $m$ of elements with large frequencies and thus calculate the memory usage as $|M| \times (1 + max(b+1, \log_2 |E|)) + 31 \times |E|$, then we can choose a value for b between 1 to 31 that minimize the memory usage. As shown in Figure 3.4 , memory usage resulting from choosing b = 31 is only slightly higher than the optimal value of b = 19 . Since b = 31 leads to word-aligned entries in the main array, this improves query performance while paying only a modest price in terms of space usage (Other components of the framework uses significantly more memory). Thus, we assign $b$ = 31 in our implementation and cancel the descriptor.

### 3.5 Mapping Target Phrase Indexes to Phrase Contexts

In the third step, we associate each target phrase with its corresponding phrase contexts. Each phrase context consists of three components: (1) Context type indicating

whether it is a *left context*, *right context* or *left-right context*; (2) ContextID representing the content of the context in the format of unigram/bigram indexes; (3) Frequency of this context. However, context type is only used in lexical pruning, so only ids and frequencies are required to be stored. As discussed in 1.2, we calculate phrase similarity by identifying all contexts common to the two target phrases to be compared, along with their frequencies, which shares many similarities with *inverted indices*.

*Inverted index* is the core data structure in a search engine. It consists of two main components: a *dictionary* containing all possible terms and a sequence of document identifications associated with each term, known as a *posting list*. We compare the structure of *posting list* and *context array* as follows:

- *Posting list*: Each element in the posting list of a term $t$ is represented using a triplet: ($DocID$, $TF$, [$pos_1$, $pos_2$, ..., $pos_n$ ]), where $DocID$ represents a document that contains $t$. $TF$ stands for term frequency of $t$ in the $DocID_{th}$ document and [ $pos_1$, $pos_2$, ..., $pos_n$ ] are the positions where $t$ occur in this document.

- *Context array*: Each element in the context array of a phrase $p$ can be represented using the triplet (*ContextID*, *Frequency*, *ContextType*).

A search query locating all documents containing a set of words can be answered by retrieving the posting lists of these words and taking their intersections. This is similar to the computation of phrase similarities using the context array, since we intersect the context arrays to find all common contexts of the two words. Thus, techniques developed to support fast queries on inverted indexes can also be used to speed up queries on context arrays.

We can borrow many ideas from the techniques applied on *inverted index*. We name the data structure as *gram-context-indexer*, keeping a *dictionary* storing pointers of target phrases to corresponding context arrays. Differences between inverted index and *gram-context-indexer* is discussed in following contents.

We observe that 23.7% of the phrase contexts exist in only one context array. We call those instances *singleton*s. According to the process described in Section 1.2, *singleton*s should be included when calculating cosine similarities, but they are

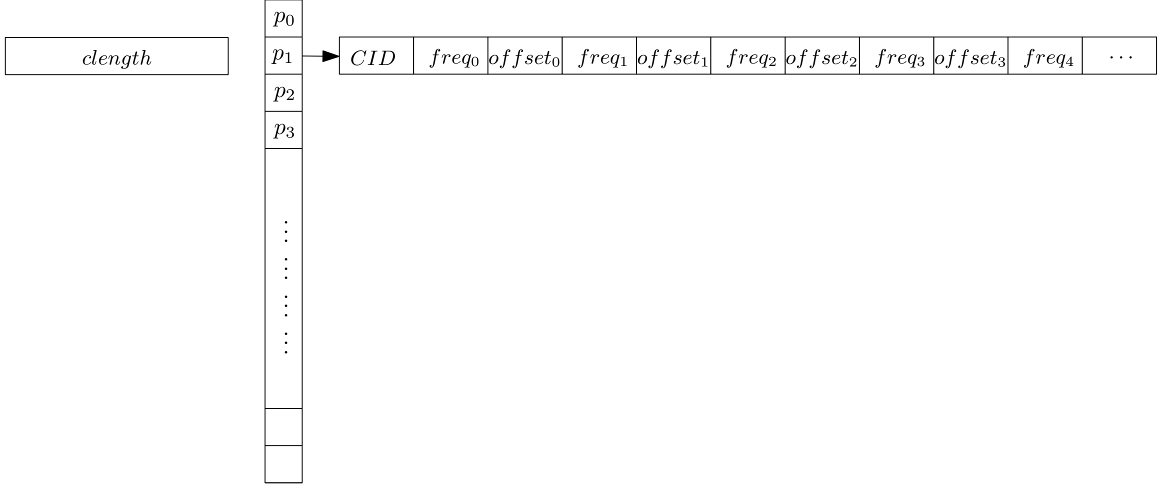



Figure 3.6: Structure of *gram-context-indexer*

*Note:* Structure of *gram-context-indexer*, *CID* stands for the first context id, *offset* stands for differential values of adjacent sequential indices, *freq* stands for the frequency of corresponding phrase context. $P$ indicates the pointer pointed to context information.

guaranteed to be excluded when calculating intersections. Thus, we can save space by keeping a variable *clength* representing the length of context array with *singletons* included, but we do not keep *singletons* in memory. As discussed in section 1.2, cosine similarity is calculated between two binary vectors. Given the length of $P_1$ context array as $clength_1$, $P_2$'s as $clength_2$ and the number of common context as $clength_c$, the value of divisor is expected to be $(clength_1 + clength_2 - clength_c)^2$ and the dividend is expected to be $clength_c$. Thus, we expect to have the cosine similarity calculated directly following $\frac{clength_c}{(clength_1+clength_2-clength_c)^2}$ without operation like union finding.

Our experiments discussed in section 4.3 shows that naive implementation of *gram-context-indexer* leads to high memory cost. We expect far less memory usage on *gram-context-indexer* with little penalty on calculation speed.

We know from Chapter 3.3 that ids of unigrams, bigrams and contexts are distributed in disjoint ranges, we subtract each context id with the capacity of unigrams, thus we can save memory by compressing context ids with smaller values. The construction process of *gram-context-indexer* works as follows: (1) First, we construct the context arrays by reading the trigram and fourgram files. Each trigram produces contexts for three unigram phrases: a *right-context* for the first unigram, a *left-right-context* for the middle unigram, and a *left-context* for the last unigram. A fourgram

similarly produces contexts for three bigram phrases. We append each such context to the context array of the respective phrase. (2) Second, we sort each context array treating indexes as key information and frequencies as satellite information. (3) Next, we calculate the differences between adjacent indexes. (4) Finally, we compress the context array using variable-length encoding.

**Single instruction, multiple data (SIMD):** Single instruction-multiple data (SIMD) architectures allow the same operation to be carried out simultaneously on multiple data elements, usually stored in vectors. This is also known as data-level parallelism. Modern CPUs support a number of different such vector operations. Whenever a computation can be expressed in terms of these operations, this usually leads to significantly faster algorithms.

**Fast differential coding:** Given the fact that deltas of sequential context indices are expected to be non-negative and far smaller than original indices, we store all phrase contexts increasingly by ids and treat the sequence of ids using differential coding.

Differential coding works by dealing with an array of sorted integers $(x_1, x_2, x_3, ..., x_n)$ and representing each element using deltas between adjacent elements as $(x_1, x_2 - x_1, x_3 - x_2, ..., x_n - x_{n-1})$. The original integer array can be recovered by accumulating the prefix sum. This process trivially affect retrieval speed. However, it may be a potential bottleneck if the following steps are highly optimized. The speed penalty is introduced by data recovery, because we cannot recover an integer without knowing the prefix sum of all previous deltas. Lemire proposed several SIMD-accelerated approaches of fast delta coding by balancing recovery speed and length of delta width operated on 32-bit integer sequences [23], so all schemes are started with "D" indicating they are dealing with differential codings with different trade-offs. They work as follows:

- D1: We may take advantage of SIMD operations to parallelize scalar differential coding. For each group of four integers $(x_1, x_2, x_3, x_4)$, we may calculate deltas of adjacent elements as follows:

- Right shift the copy of $(x_1, x_2, x_3, x_4)$. Then prepend the last element of previous group. This produces a quadruple $(x_0, x_1, x_2, x_3)$.
- Compute the delta values $(\delta_1, \delta_2, \delta_3, \delta_4)$ by calculating $(x_4, x_3, x_2, x_1) - (x_3, x_2, x_1, x_0)$ using the *_mm_sub_epi*32 instruction.

D1 requires 6 SIMD instructions to finish decoding for each block with 4 integers. Assume we have deltas $d_1$=$(\delta_5, \delta_6, \delta_7, \delta_8)$, first, we make a copy of $d_1$ and right shift it for two integers, after that, we sum the right-shifted copy and original deltas and we get $d_2$=$(\delta_5, \delta_6, \delta_5 + \delta_7, \delta_6 + \delta_8)$. We make a copy of $d_2$ and right shift one integer, then we perform a sum operation again and we get $d_3$=$(\delta_5, \delta_5 + \delta_6, \delta_5 + \delta_6 + \delta_7, \delta_5 + \delta_6 + \delta_7 + \delta_8)$. The original integers can be recovered by adding $(x_4, x_4, x_4, x_4)$ with $d_3$.

- D2: This approach is similar to D4, but calculates deltas relative to the element two positions earlier in the list: $(\delta_5, \delta_6, \delta_7, \delta_8) = (x_5, x_6, x_7, x_8) - (x_3, x_4, x_5, x_6)$, so the delta is expected to be 2 times larger than D1's.

  D2 requires 4 SIMD instructions to finish recovery. Similar to D1, we need a right-shifted copy of $(\delta_5, \delta_6, \delta_7, \delta_8)$, which is $(0, 0, \delta_5, \delta_6)$, then we add the copy and the original deltas and we get $(\delta_5, \delta_6, \delta_5 + \delta_7, \delta_6 + \delta_8)$. Then the recovery is finished by adding $(x_3, x_4, x_3, x_4)$ with $(\delta_5, \delta_6, \delta_5 + \delta_7, \delta_6 + \delta_8)$.

- DM: We may compute deltas relative to the last element of the previous block. Following the example of D2, deltas are calculated as: $(\delta_5, \delta_6, \delta_7, \delta_8) = (x_5, x_6, x_7, x_8) - (x_4, x_4, x_4, x_4)$. This approach is expected to generate 2.5 times the size of deltas generated using D1.

  DM requires 2 SIMD instructions for recovery. We need *PSHUFB* to get $(x_4, x_4, x_4, x_4)$ and we add it with $(\delta_5, \delta_6, \delta_7, \delta_8)$, then we finish the recovery.

- D4: Instead of computing the delta between each element and its predecessor, this method computes the difference between each element and the element 4 positions earlier in the list. For example, if we have a sequence of integers $\{x_1, x_2, ..., x_8\}$, then the deltas of this sequence are calculated as : $(\delta_5, \delta_6, \delta_7, \delta_8) = (x_5, x_6, x_7, x_8) - (x_1, x_2, x_3, x_4)$. Deltas are calculated every four elements, so the delta is expected to be 4 times larger than D1.

D4 takes full advantage of SIMD and it only requires one SIMD instruction for recovery. For example, we simple add $(x_1, x_2, x_3, x_4)$ and $(\delta_5, \delta_6, \delta_7, \delta_8)$.

From D1 to D4, we expect decrements on the number of required instructions with more memory cost as trade-off. The number of instructions needed for recovery does not reflect wall-clock time, it can only show the relative speed between the four approaches. SIMD based differential coding works by simultaneously handling chunks of data with 4 32-bit integers in our case. However, the number of elements in the input integer sequence is not guaranteed to be a multiple of 4, tail elements are handled with the non-SIMD approach, which means that SIMD based approaches benefit performance only if the input sequences aren't too short.

**Variable-length coding:** According to Zipf's law, the frequency of an n-gram show inversely proportional to its number of words, which means lower frequencies are expected to occur more frequently for trigrams and fourgrams, so we expect far fewer bits to store frequency values of contexts. Deltas of context ids are expected to be smaller than original ids. However, digits represented using machine words are not compact. We introduce variable-length encoding for more compact storage. And we distinguish encoding methods based on the alignment of encoded values, namely bit-aligned coding, byte-aligned coding, word-aligned coding and frame-aligned coding. Bit-aligned coding methods can produce codes consisting of an arbitrary number of bits. This produces the most compact codes but requires us to work with individual bits during encoding and decoding. Byte-aligned codes produce codes consisting of a number of bits that is a multiple of 8. Thus, these codes can be accessed on a byte-by-byte bases. Word-aligned coding keep 4 bits in one word (32-bit or 64-bit) as *selector* and the rest bits as *payload*, then it tries to compile as many integers as possible inside one word. Frame-aligned coding represent data as a sequence of bit-frames with same width for the convenience of parallization.

**Bit-aligned coding:** Bit-aligned coding schemes handle data on a bit-by-bit base. We tried several variable-length encoding methods to perform bit-aligned coding of the context array, including Golomb / Golomb-Rice coding [29], Elias encoding [11] and block encoding [4, 25]. These variable length encoding methods work by adding

a binary prefix to each number that indicates the number of bits used to encode the actual number. Given a number $n$ and assuming the length of its natural binary representation is $|n|$, the bit-aligned coding methods work as follows:

- *Golomb/Golomb-Rice code:* Golomb code works by dividing the integer to be encoded into two components: quotient and remainder, in which the quotient is represented using unary and the remainder using binary. When using the Golomb code to compress a sequence of integers, the divisor $M$ is usually set to 0.69 times the mean value of the integers in the sequence. The Golomb-Rice code chooses $M$ to be a number equal to $2^n$, since this means the division can be implemented much more efficiently using a simple bit shift instead of floating point division. For example: Assume we want to encode 9 and divisor is set to 4, then we know $9 = 2 * 4 + 1$, so the encoded value will be 01 001. This coding method is compact but quite slow when required to encode large values.

- The Elias $\gamma$ code represent the number $n$ using $2\|\log n\| + 1$ bits. It performs encoding by prepending $|n| - 1$ 0s to the natural binary representation of $n$. During the decoding process, we read in 0s until reaching the first 1. Assume the number of 0s is *N*, then the next $N$ bits will be $n$ in binary representation. For example: given an uncompressed number 7 with its binary representation 111, we know it takes 3 bits to represent. Thus we represent its length using prefix 00 and the Elias $\gamma$ code of 7 is 00 111.

- The construction of the Elias $\omega$ code of $n$ is obtained by setting $k = n$ and initializing the code to consist of a single 0. Then, while $k > 1$, we prepend the binary encoding of k to the code and replace $k$ with $|k|$ - 1. And the decoding process reverse this construction. We set $n = 1$. Next we read one bit. If this bit is 0, we report $n$ as the decoded number. Otherwise, we read $n$ more bits, prepend 1 to this bit sequence, replace $n$ with the number encoded by this bit sequence and repeat the above process. For example: Given a number 16 and we know it takes 5 bits to encode. We prepend the binary format of 16 before 0, getting 10000 0. Then we prepend the binary format of 5 - 1 = 4, which gives 100 10000 0, then we prepend the binary format of 3 - 1 = 2, and we get 10 100 10000 0, after this step we find $N$ becomes 1, so we stop.

- The Elias $\delta$ code use Elias $\gamma$ encoded $|n| + 1$ as the prefix and it represents one digit $x$ using $\lfloor \log x \rfloor + 2\lfloor (\lfloor \log x + 1 \rfloor) \rfloor + 1$ bits. During decoding process, we first decode the prefix with Elias $\gamma$ code, assume the decoded value is $N$, then the next $N - 1$ bits will be $n$ in binary format. For example: Given an uncompressed digit 7, we know $7 = 2^2 + 3$, so the Elias $\delta$ code of 7 is 011 11.

- The Block code defines the block size as $k$ and it calculates the number of blocks $N$ required for the representation of $n$ based on $2^k$, then it prepends the unary format of $N$ before the binary-formatted $n$. During the decoding process, we read in 0s until the first 1. Assume the number of 0s is N, then the next $N \cdot k$ bits will be $n$ in binary format. For example: Given a number 7 and block size 2, we know 7 is a 3-bit long integer and we need 2 blocks to store 7, so the block-encoded 7 will be 01 01 11.

We can tell from the working strategy of the bit-aligned coding that integer sequence are compressed as bit streams and decoding process has to be performed on a bit-by-bit base, which is relatively slow due to restrictions caused by frequent bit operations. We expect faster compression methods, which are discussed in following contents.

**Byte-aligned coding:** The atomic unit of byte-aligned coding to represent one integer is one byte. The simplest byte-aligned coding described in this thesis cited from [10] uses one bit per byte as descriptor indicating whether the current byte is the last byte of the encoded number, while the other 7 bits are used to encode the number. This encoding is *variant-byte* (*vbyte*). [32] recognizes that compression can be accelerated by aggregating descriptors in one byte. This approach is named *variant-GB*, where *G* stands for group and *B* stands for binary. In *variant-GB* approach, integers are stored in blocks ranging from 1 byte to 4 bytes for every 4 blocks and descriptors are aggregated in one descriptor byte. Each aggregated descriptor byte contains 4 descriptors, and each descriptor can be 00, 01, 10 or 11, which indicates the actual width of the corresponding block with 1 byte, 2 bytes, 3 bytes or 4 bytes respectively. By reading descriptor byte, we avoid reading data byte by byte. In addition, since each descriptor byte contains 4 descriptors and each descriptor contains 4 states, we expect 16 kinds of different schemes for the data part. Thus, we expect to

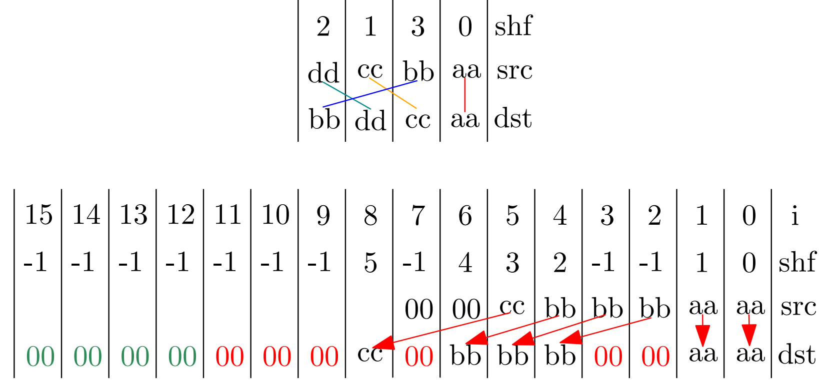



Figure 3.7: Coding scheme of using SIMD operation *pshufb* (*Variant-G8IU*)

Note: *PSHUFB* accept two parameters: shuffle sequence *shf*, data *src*, and return shuffled sequence *dst*. *PSHUFB* assign *src[shf[i]]* to *dst[i]* if *shf[i]* is not -1, or *dst[i]* will be assigned 0.

accelerate the decoding process by calling corresponding decoding methods directly for corresponding block. We leave this implementation as future work.

Data compressed using *Variant-GB* is not of fixed length, so the data block is expected to be as short as 4 bytes or as long as 16 bytes, which makes it harder to be parallelized. However, the decoding process can be further accelerated using SIMD instruction *pshufb* with some modification on the coding scheme. Thus, *Variant-G8IU* is introduced, where "U" stands for unary, "8" stands for 8 bytes, "I" stands for incomplete. Different from *Variant-GB*, *Variant-G8IU* use unary 0, 01, 011, 0111 to describe the length of each integer and integer are compiled in blocks of 8 bytes, so the descriptor is not guaranteed to be fully filled if current integer exceeds the remaining space of current block. *Variant-G8IU* solves this problem by padding descriptor with 1s and the remaining space in current block with 0s.

We may realize the process of encoding / decoding is actually removing / adding 0s before the natural representation of integers [32]. If given the aggregated descriptor, the decoding process can be efficiently completed with SIMD instruction *pshufb* and data blocks of fixed length. *pshufb* works by receiving two parameters: the data array and a shuffle sequence and return a new array with each byte assigned a value according to the shuffle sequence. The working process of *pshufb* is shown in Algorithm 1. For each byte $i$ in the shuffle sequence, *pshufb* either assign the $i_{th}$ byte in returned array with the $i_{th}$ byte from data array, or assign $i_{th}$ byte in the returned array as

**Algorithm 1** *pshufb*

**Input:** shuffle sequence $M$, data source $S$
**Output:** destination sequence $D$

```
1: procedure
2:     for 0 ≤ i < 16 do
3:         if M[i] < 0 then
4:             D[i] ← 0
5:         else
6:             D[i] ← S[M[i] % 16]
7:         end if
8:     end for
9: end procedure
```

0 if $i$ is -1. We show the decoding process of *Variant-G8IU* using *pshufb* in Figure 3.7: by reading the descriptor byte, we generate the corresponding shuffle sequence. Then we use *pshufb* to reinsert 0s to the compressed data and write recovered data to output.

Actually, the construction of shuffle sequence works similarly with the decoding process by inserting -1s in the shuffle sequence indicating positions of bytes from the compressed array. Once valid shuffle sequences are constructed, the decoding process works simply by looking up corresponding shuffle sequence and performing the shuffle. A similar idea can be applied to many block-based encoding methods with a finite set of possible descriptor permutations. For *Variant-GB* format,one descriptor byte contains 8 bits, so $2^8$ possible shuffle sequences can be constructed in advance and all possible shuffle sequences are valid. However, for *Variant-G8IU* format, if one descriptor byte contains consecutive 0s with more than 4 bit-distance, it has to be trimmed.

Shuffle array construction is based on two accessory functions, which is described as follows:

- *num(descriptor byte)*: returns the number of integers encoded in the coding block.

- *len(descriptor byte,i)* : returns the actual length in bytes of the $i_{th}$ encoded integer in the coding block.

**Algorithm 2** Shuffle Sequence Construction of *Variant-G8IU*

**Input:** valid descriptor byte $B$

**Output:** shuffle sequence $Q$

1: **procedure**
2: $i \leftarrow 0$, $j \leftarrow 0$
3: **for** $i = 0 : num(B)$ **do**
4: **for** $n = 0 : 4$ **do**
5: **if** $n < len(B, i)$ **then**
6: $Q[j] \leftarrow i$
7: $i \leftarrow i + 1$
8: **else**
9: $Q[j] \leftarrow -1$
10: **end if**
11: **end for**
12: **end for**
13: **return** $Q$
14: **end procedure**

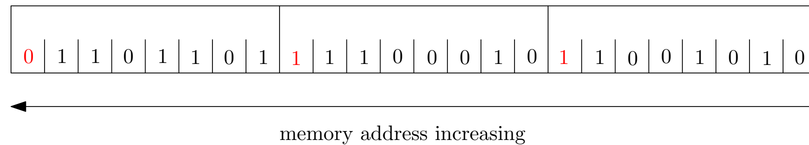



Figure 3.8: Coding scheme of Variant-Byte

*Note:* We encode 1798474, with 0 indicating the termination of encoding.

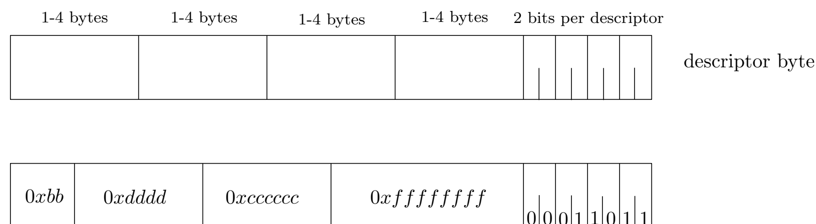



Figure 3.9: Coding Scheme of *Variant-GB*

*Note:* Coding Scheme of *Variant-GB* with data 0xffffff, 0xcccccc, 0xdddd and 0xbb, in which 0xffffffff takes 4 bytes, 0xcccccc takes 3 bytes, 0xdddd takes 2 bytes and 0xbb takes 1 byte.

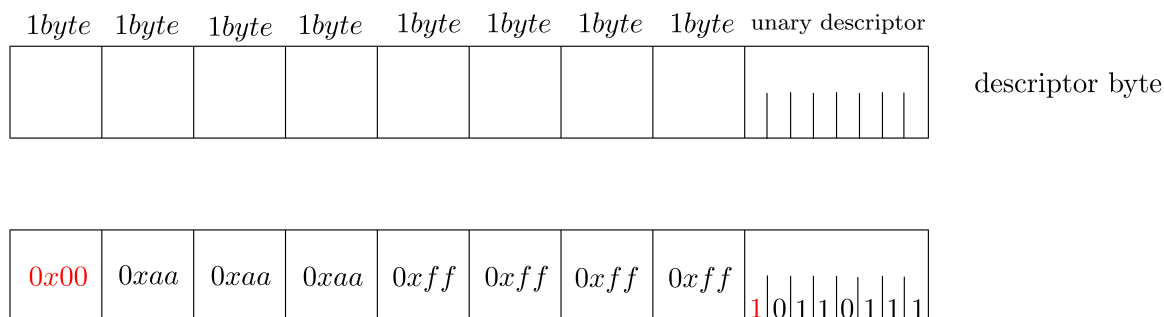



Figure 3.10: Coding Scheme of *Variant-G8IU*

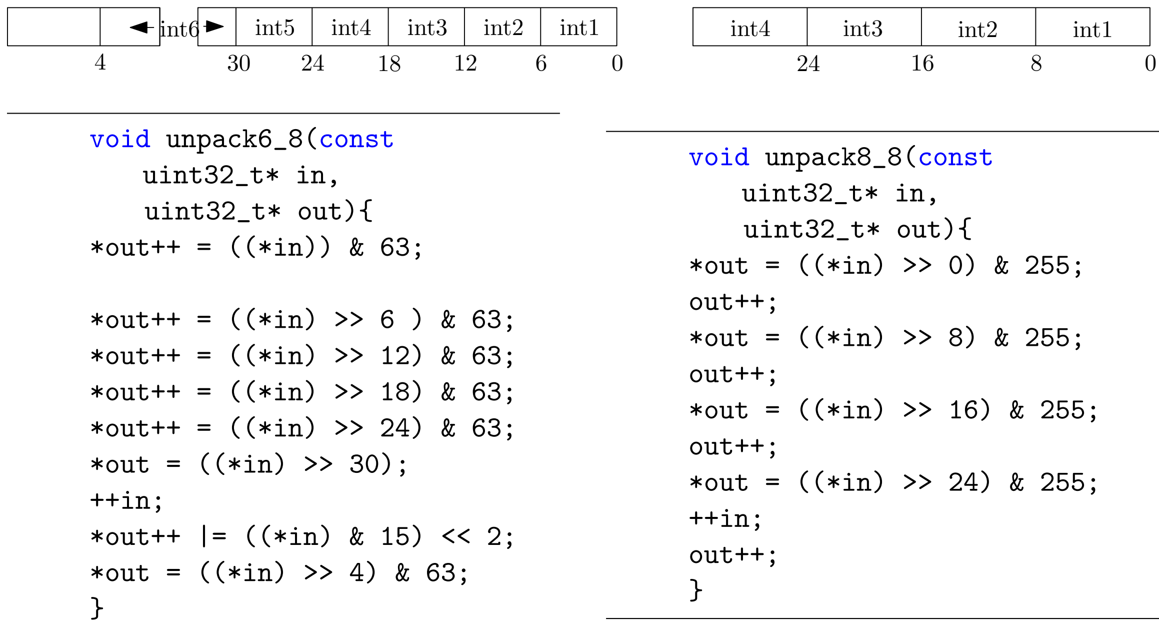
*Variant-G8IU* scheme with data 0xfffffff and 0xaaaaa. Length of 0xfffffff is represented using 0111 and 0xaaaaaa is represented using 011. Assume our next digit is wider than 1 byte, we should not include the next digit in this block using *Variant-G8IU*, we should pad the descriptor with 1 and 0 with the data block.

**Word-aligned coding:** Word-aligned coding, also known as *simple-coding*, try to store as many values as possible inside one word by dividing each word into *selector* and *payload*, in which *selector* shows the number of integers this word may store while *payload* try to store as many integers as the *selector* indicates. Word-aligned coding contains *simple-9*, *simple-8b* and *simple-16*. *Simple-9* keeps first 4 bits in one word as *selector* and the rest 28 bits as *payload*. *simple-8b* optimize *simple-9* by recognizing 7 *selectors* are idle in *simple-9* scheme. Besides, some coding schemes in *simple-9* (e.g: when selector is 4, only 25 bits can be used in *payload*) result in wasted space. *simple-16* introduce optimization by using larger word (64-bit) while maintaining *selector* of same length, so fewer *selector* bits are paid on average for each bit in *payload*. We listed all possible coding schemes for *simple-9*, *simple-8b* and *simple-16* in Table 3.3. Word-aligned approaches have been demonstrated to be a fast approach [34]. However, despite *Simple-16*, other two simple-family codecs can only handle integers within the range of $2^{28}$.

**Frame-aligned coding:** Frame-aligned coding shares many similarities with byte-aligned coding schemes. But data are compressed in frames of fixed length. The frame-aligned coding family includes *PackedBinary* [2], *Patched Frame of Reference (PFor)* [36]. *PackedBinary* uses the bit-width of the largest value in each data block as frame-width, and bit packing process is finished by process described in Figure 3.11, and the bit shifting operation can be efficiently completed by replacing standard bit

| | Simple-9 | | Simple-8b | | Simple-16 | |
|---|---|---|---|---|---|---|
| Selector | Capacity | Integer Width | Capacity | Integer Width | Capacity | Integer Width |
| 0 | 28 | 1 | 28 | 28×1 | - | 0 |
| 1 | 14 | 2 | 21 | 7×2,14×1 | - | 0 |
| 2 | 9 | 3 | 21 | 7×1,7×2,7×11 | 60 | 1 |
| 3 | 7 | 4 | 21 | 14×1,7×2 | 30 | 2 |
| 4 | 5 | 5 | 14 | 14×2 | 20 | 3 |
| 5 | 4 | 7 | 9 | 1×4,8×3 | 15 | 4 |
| 6 | 3 | 9 | 8 | 1×3,4×4,3×3 | 12 | 5 |
| 7 | 2 | 14 | 7 | 7×4 | 10 | 6 |
| 8 | 1 | 28 | 6 | 4×5,2×4 | 8 | 7 |
| 9 | - | - | 6 | 2×4,4×5 | 7 | 8 |
| 10 | - | - | 5 | 3×6,2×5 | 6 | 10 |
| 11 | - | - | 5 | 2×5,3×6 | 5 | 12 |
| 12 | - | - | 4 | 4×7 | 4 | 15 |
| 13 | - | - | 3 | 1×10,2×9 | 3 | 20 |
| 14 | - | - | 2 | 2×14 | 2 | 30 |
| 15 | - | - | 1 | 1×28 | 1 | 60 |

Table 3.3: Word-aligned codecs (Simple-family Codecs)




```
void unpack6_8(const
    uint32_t* in,
    uint32_t* out){
*out++ = ((*in)) & 63;

*out++ = ((*in) >> 6 ) & 63;
*out++ = ((*in) >> 12) & 63;
*out++ = ((*in) >> 18) & 63;
*out++ = ((*in) >> 24) & 63;
*out = ((*in) >> 30);
++in;
*out++ |= ((*in) & 15) << 2;
*out = ((*in) >> 4) & 63;
}
```

```
void unpack8_8(const
    uint32_t* in,
    uint32_t* out){
*out = ((*in) >> 0) & 255;
out++;
*out = ((*in) >> 8) & 255;
out++;
*out = ((*in) >> 16) & 255;
out++;
*out = ((*in) >> 24) & 255;
++in;
out++;
}
```

Figure 3.11: Binary Packing with 6-bit frames and 8-bit frames

shifting operation with SIMD intrinsics: *_mm_and_si*128 (&), *_mm_srli_epi*32 (>>), *_mm_slli_epi*_32 (<<). However, as integers are packed with same width, Packed-Binary will leads to inefficiency when occasional large numbers are included. PFor solves this problem by introducing *patching*. That is, data are stored in blocks of fixed

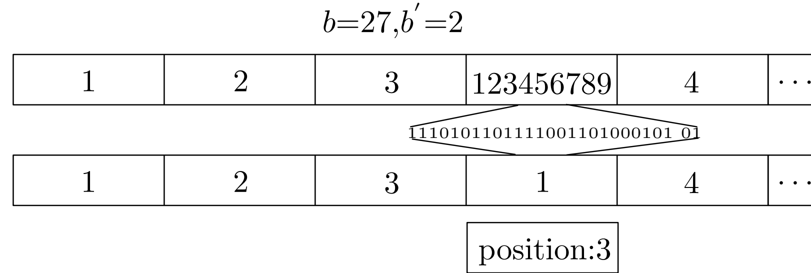



Figure 3.12: Patched Frame of Reference

Note: despite of 123456789, all other members do not exceed 3, thus $b$=27 and $b^{'}=2$, the lower 2 bits of 123456789 is packed together with other members, while one byte is kept to indicate the position of the exception in this block.

size (usually 128 integers), and a frontier $b^{'}$ is set for each block so that majority of values inside this block requires less than $b^{'}$ bits for encoding. The exceptions can be compactly stored in accessory data structures of 32 different exception sequences. For each data block, the frontier $b$ is decided by solving an optimization problem of minimizing $128 \times b^{'} + c(b^{'}) \times (b - b^{'} + 8)$, where $c(b^{'})$ stands for the number of exceptions decided by $b^{'}$ and 8 stands for the memory cost by keeping one byte indicating the position of the exception in this block. $b$ and $b^{'}$ can be efficiently calculated by linearly scanning the block using *bsr* instruction for the calculation of $log_2$ of each element.

Assume we have a block of 128 32-bit integers starting with 1, 2, 3, 123456789, 2, 3, 1, in which 123456789 is the only member exceeding 3. Then we may store each member using 2 bits while keeping metadata indicating high bits of each exception and we need less than 32 escape arrays storing the high bits of all exceptions. This example is explained in Figure 3.12.

**Conclusion:** We can tell from the working strategies of all this coding schemes that when memory is the main concern, bit-aligned coding approaches are expected to perform better when integer sequences are small, since smaller integers result in shorter prefixes, while byte-aligned methods are expected to perform better when dealing with larger values because the length of descriptors only distributes in a small range (1-4). When speed becomes our primary concern, we consider storing data into an aligned format, either as sequences of machine word or data blocks, so

compression and decompression can be accelerated using SIMD instructions. The frame-based approaches are quite suitable for very long integer sequences. However, it cannot handle integer sequences efficiently with capacity less than the block size.

We observe that the length of context arrays are not ideally distributed in a small range. Some context arrays are fairly short while some others are quite long. The frame-aligned coding schemes, like *PFor*, compress data in blocks of 128 integers, thus making it incompatible with short context array. On the other hand, bit-aligned approaches are compatible with any distribution of integer sequence, but they are much slower than frame-aligned and byte-aligned coding. All these facts suggest the necessity of introducing a hybrid compression algorithm. We did not look into this very deeply and use a simple heuristic. Assume we have a monotonic integer sequence $A$ with $|A|$ elements: (1) if $128 \leq |A|$, we compress it using *PFor* (2) if $|A| < 128$ and $8 \leq |A|$, we encode this sequence using *Variant-G8IU* (3) if the sequence contains less than 8 elements, we compress it using Elias $\delta$ coding. Here an element is a phrase context, which contains a pair of integer indicating the id of context and its corresponding frequency.

## 3.6 Intersection

According to Chapter 1.2, the calculation of phrase relatedness requires finding the intersections of phrase context arrays. During our experiment, we found that the length of context arrays varies greatly, some context arrays contain only 2 to 3 elements while others could contain more than 10,000 elements. Thus, we introduce optimization on intersection operation for greatly imbalanced arrays. A textbook approach described in Algorithm 3 for finding the intersection of two sorted lists runs in $O(m + n)$ , where $m$ stands for the size of larger array and $n$ stands for the size of smaller array. Let us note the larger of the two arrays as $V_l$ and the smaller of the two arrays as $V_r$. The algorithm scans $V_l$ and $V_r$. In each step, it checks whether the current elements in $V_l$ and $V_r$. Otherwise, it advances to the next element only in $V_l$ or $V_r$, whichever contains the smaller of the two current elements.

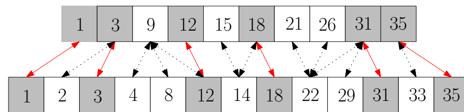
We may also use a hash map to represent the context array, so we can theoretically perform each intersection operation with $O(kn)$ time. However, the general STL unordered map show no optimization on the hidden constant factor $k$, thus making

**Algorithm 3** non-SIMD Linear Scan

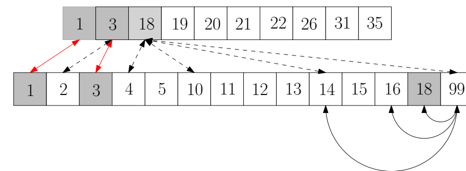
**Input:** context arrays of queried phrase pairs $A_1$ and $A_2$, block size $T$

**Output:** intersected context arrays $A_3$ and $A_4$, containing all elements common to $A_1$ and $A_2$, with corresponding frequency value attached.

```
 1: procedure
 2:     i ← 0, j ← 0
 3:     while i < |A1| and j < |A2| do
 4:         if A1[i] equals A2[j] then
 5:             add A1[i] to A3
 6:             add A2[j] to A4
 7:             i ← i + 1
 8:             j ← j + 1
 9:         else if A1[i] > A2[j] then
10:             j ← j + 1
11:         else
12:             i ← i + 1
13:         end if
14:     end while
15:     return A3,A4
16: end procedure
```




(a) non-SIMD Linear Scan




(b) non-SIMD Galloping

Figure 3.13: Scalar intersection of monotonic integer sequence

it less competitive compared with the non-SIMD linear scan in our experiment. [3] introduced an optimized approach with $O(n \log i)$ running time, named *galloping search* (or *exponential search*), where $i$ stands for the index of matched element in the long list. *Galloping search* take every element $es_i$ from the shorter sequence, looking for the first element $el_i$ in longer sequence that no less than $es_i$. Instead of seeking for $es_i$ sequentially, galloping search double the distance for each step, which makes it advantageous when $m \gg n$. [23] introduced a hybrid SIMD accelerated approach for retrieving intersections of two sorted integer sequences, which consists of V1 intersection, V3 intersection, and SIMD galloping intersection. Acceleration is achieved by performing all-against-all comparisons, in which V1 means "one block is vectorized" and V3 means the vectorized is doubled for addtional two times. V1, which is explained in Algorithm 4, is similar to non-SIMD linear scan but compares each element in $V_s$ to $T$ (usually $T = 8$) elements in $V_l$. V3 is explained in Algorithm 5. It optimizes V1 by recognizing that more comparisons can be skipped by enlarging the searching range. V3 tries to locate a block of $4T$ elements in $V_l$ for every queried element $es_i$ from $V_s$. Once we find a match in those $4T$ elements for current $es_i$, we perform a binary search inside those $4T$ elements, managing to find a block of $T$ elements containing $es_i$. Thus we avoid unnecessary comparisons by checking $4T$ elements every time. However, V1 and V3 will be less competitive compared with *galloping* when $V_l$ becomes extremely large compared to $V_s$. [23] introduces SIMD galloping for further optimization, which follows the basic idea of *galloping*, but it compares blocks of $T$ elements instead of single ones, thus the time complexity of SIMD-galloping is still $O(n \log i)$ . We notice SIMD-based approaches introduce optimization by lower the constant factor of each non-SIMD based algorithms. SIMD-hybrid algorithm is based on a simple heuristic: (1) When $length(V_s) \leq length(V_l) \leq 50 * length(V_s)$, V1 is applied. (2) When $50 * length(V_s) \leq length(V_l) \leq 1000 * length(V_s)$, V3 is applied. (3) When $1000 * length(V_s) \leq length(V_l)$, SIMD galloping is applied. Lemire claims this heuristic works well on 32-bit sequences. For 64-bit integer sequences, different heuristics should be explored.

Lemire's original implementation focuses on evaluating only the intersection speed, that is, it only cares about the existence of queried sets, thus making the implementation not compatible with our data format. We re-implemented the algorithm, adding

**Algorithm 4** V1 Intersection

**Input:** context arrays of queried phrase pairs $A_1$ and $A_2$, block size $T$

**Output:** intersected context arrays $A_3$ and $A_4$, containing all elements common to $A_1$ and $A_2$, with corresponding frequency value attached.

```
1:  procedure
2:      i ← 0, j ← 0
3:      for i = 0 to |A_1| do
4:          R ← {A_1[i],A_1[i],A_1[i],A_1[i]}
5:          while A_2[j − 1 + T] < A_1[i] do
6:              j ← j + T
7:              if j > |A_2| then
8:                  return A_3,A_4
9:              end if
10:         end while
11:         F ← {A_2[j], A_2[j + 1], A_2[j + 2], A_2[j + 3], · · · , A_2[j + T − 1]}
12:         if any R_i ∈ F then
13:             add A_1[i] to A_3
14:             k ← binarysearch(F,A_1[i])
15:             add A_2[k] to A_4
16:         end if
17:     end for
18:     return A_3,A_4
19: end procedure
```

**Algorithm 5** V3 intersection

**Input:** context arrays of queried phrase pairs $A_1$ and $A_2$, block size $T$
**Output:** intersected context arrays $A_3$ and $A_4$, containing all elements common to $A_1$ and $A_2$, with corresponding frequency value attached.

```
 1: procedure
 2:     i ← 0, j ← 0
 3:     for i = 0 to |A_1| do
 4:         R ← {A_1[i],A_1[i],A_1[i],A_1[i]}
 5:         while A_2[j − 1 + 4T] < A_1[i] do
 6:             j ← j + 4T
 7:             if j > |A_2| then
 8:                 return A_3,A_4
 9:             end if
10:         end while
11:         if A_2[j + 2T − 1] > A_1[i] then
12:             if A_2[j + T − 1] > A_1[i] then
13:                 F ← {A_2[j], A_2[j + 1], ··· , A_2[j + T − 1]}
14:             else
15:                 F ← {A_2[j + T], A_2[j + T + 1], ··· , A_2[j + 2T − 1]}
16:             end if
17:         else
18:             if A_2^{j+3T−1} > A_1^i then
19:                 F ← {A_2[j + 2T], A_2[j + 2T + 1], ··· , A_2[j + 3T − 1]}
20:             else
21:                 F ← {A_2[j + 3T], A_2[j + 3T + 1], ··· , A_2[j + 4T − 1]}
22:             end if
23:         end if
24:         if any R_i ∈ F then
25:             add A_1[i] to A_3
26:             k ← binarysearch(F,A_1[i])
27:             add A_2[k] to A_4
28:         end if
29:     end for
30:     return A_3,A_4
31: end procedure
```

**Algorithm 6** SIMD galloping intersection

**Input:** context arrays of queried phrase pairs $A_1$ and $A_2$, block size $T$
**Output:** intersected context arrays $A_3$ and $A_4$, containing all elements common to $A_1$ and $A_2$, with corresponding frequency value attached.

```
1:  procedure
2:      j ← 0
3:      for i = 0 to |A_1| do
4:          R ← {A_1[i],A_1[i],A_1[i],A_1[i]}
5:          δ ← 0
6:          while A_2[j + δ − 1 + T] < A_1[i] and j < |A_2| do
7:              if δ equals 0 then
8:                  δ = T
9:              else
10:                 δ = 2 * δ
11:             end if
12:         end while
13:         binarysearch for δ_min in [⌊δ/2⌋, δ] divisible by T so f_{j+δ_min−1+T} ≤ A_1[i]
14:         j ← j + δ_min
15:         F ← {A_2[j], A_2[j + 1], ..., A_2[j − 1 + T]}
16:         if any R_i ∈ F then
17:             add A_1[i] to A_3
18:             k ← binarysearch(F,A_1[i])
19:             add A_2[k] to A_4
20:         end if
21:     end for
22:     return A_3,A_4
23: end procedure
```

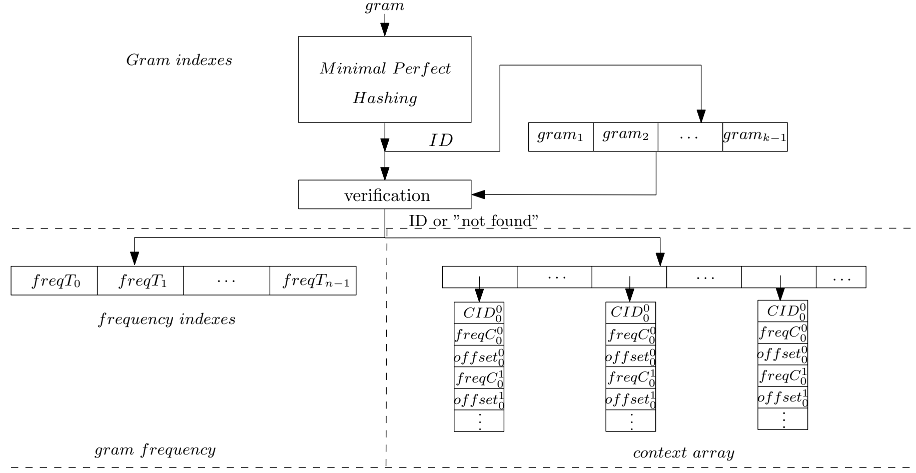



Figure 3.14: Overall structure of phrase relatedness calculation framework

*Note:* High-level System Design of High-Performance TrWP Framework, where *gram* stands for the target phrases, *index* stands for the numerical index of a gram within keyset, *freqT* stands for the frequency of target phrase, *CID* stands for the ID of the first context, *offset* stands for the index interval, *freqC* stands for the frequency of context or target phrase.

a binary search operation to search for corresponding index and corresponding frequency value in this position when a match is detected in the long array. Similar to SIMD based differential coding approaches, SIMD intersections are based on the assumption that the length of $V_l$ is divisible by 4 and tail elements have to be handled using non-SIMD linear scan. The method degrades to the non-SIMD linear scan when the longer context array contains less than $T$ elements.

## 3.7 Overall design

The overall design of the high-performance framework is shown in Figure 3.14. Assuming we have two target phrases $gram_1$ and $gram_2$. (1) We query *gram-index-indexer* to get the index of each target phrase, defined as $index_1$ and $index_2$. (2) We retrieve the frequency of each target phrase from the *gram-frequency-indexer* using $index_1$ and $index_2$. (3) We retrieve the context arrays in the format of bit vectors and decompress it; then we recover the id array. (4) We find common contexts shared by the two target phrases (5) We follow the rest process discussed in section 1.2 to get the queried phrase relatedness.

# Chapter 4

# Evaluation

## 4.1 Experiment Setup

The experiment was performed on a Linux server (2.6.32-573.18.1.el6.x86_64 GNU/ Linux, CentOS 6.7) with 32 Intel Xeon E5-2650 @ 2.00 GHz CPUs and 256 GB of main memory. Our code is written in C++ and compiled using GCC 5.3 with -O3 optimization. We chose Google Web 1T corpus for TrWP evaluation and calculated the similarity of 108 noun phrase pairs as done in [28]. The Google Web 1T corpus contains around 1 trillion words with $n$-gram lengths ranging from 1 to 5. The total file size of unigrams to fourgrams is approximately 55 GB. As discussed in section 3.2, we get pre-processed data of approximately 23.17 GB. We evaluated our framework in several aspects: construction time, query speed, memory consumption. Reducing the query time is our main focus. In our evaluation, we use wall-clock time rather than CPU clock as the criterion to evaluate the query speed.

## 4.2 Evaluation of the *gram-indexer*

We compare the retrieval speed of our *gram-indexer* with some benchmark language model toolkits. SRILM [33] is a language model toolkit providing a collection of data structures and runnable programs for evaluation of language models based on n-grams. It provides a component named Vocab for n-gram indexing, which maintains a hash table in each layer, keys in hash map represents contains a hash table using n-gram as key and index as value and an array storing n-grams in corresponding indexes. The Naive indexer [1] is consisted of a perfect hash and an auxiliary string array.

Our results are shown in Table 4.1. Compared with the SRILM, our two approaches performs better in terms of both space and query time. The Naive Indexer [1] performs no compression, and it is by far the fastest of three data structure to

| Approach | Construction Time [microsec] | Retrieval Time per Query [microsec] | Memory [GB] |
|---|---|---|---|
| SRILM | 794,912,645 | 1.644 | 14.080 |
| Naive Indexer | 17,345,549 | 0.850 | 11.008 |
| Compressed | 1,542,769,885 | 1.350 | 4.352 |

Table 4.1: Comparison of different indexing approaches in the aspect of construction time, retrieval time per query and memory cost in GB

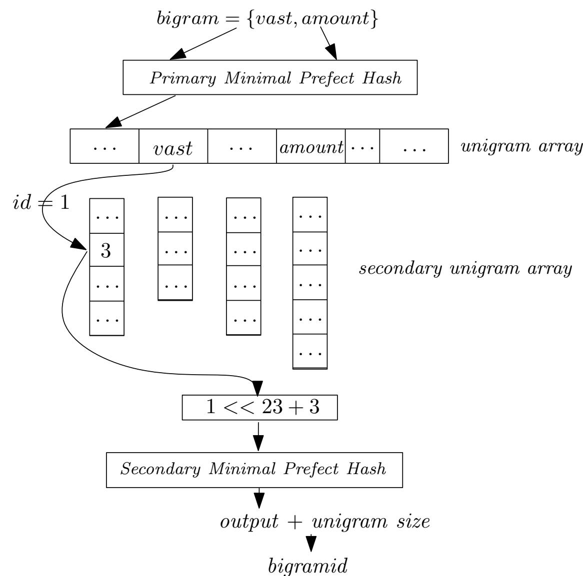



Figure 4.1: Tiered gram-indexer with sequential secondary key

construct. Its query time is about twice as fast as that of SRILM. The Naive Indexer saves 21% memory than SRILM but we expect further optimization on memory. *gram-indexer* uses 60.4% memory compared with the Naive Indexer, but at the expense of a significantly increased construction time, even compared to SRILM. Its query time is 58.8% higher than that of the naive indexer but also 17.6% less than that of SRILM. *gram-indexer* introduces extra benefit of choosing the ids of unigrams and bigrams from disjoint ranges, so other components of this framework can recognize the type of n-gram ids from its id alone. This avoids data structures for remapping in later steps.

**Other approaches** Before deciding on the final implementation of the *gram-indexer*, we tried another approach that promises fast queries in theory but resulted in inferior performance in our experiments. The structure is presented in Figure 4.1. in

| Approach | Retrieval Time per Query [microsec] | Memory [GB] |
|---|---|---|
| Naive *gram-frequeuncy-indexer* | $\ll$1 | 0.686 |
| Aligned *gram-frequency-indexer* | $\ll$1 | 0.343 |
| Compact *gram-frequency-indexer* | 1.65 | 0.343 |

Table 4.2: Comparison of retrieval times and size of different *gram-frequency-indexer* implementation.

which the unigram index generation shares a similar approach as *gram-indexer*. We avoid the second kind of false positives by attaching an integer array for each unigram in unigram array. For each attached array, we store the tokens of successive unigrams of each bigram or context in order, so the availability of concatenation can be checked by a binary search on the attached token array with $O(\log n)$ time, where $n$ stands for the length of attached token array. We found this design slows the construction and query significantly though it is more compact. In addition, the range of bigram tokens and context tokens are not disjoint, which lead to a remapping in *gram-frequency-indexer*, so we do not use this design.

## 4.3 Evaluation of the *gram-frequency-indexer*

Retrieval speed is the most important performance metric for the *gram-frequency-indexer*, because this data structure is fairly small even when optimized for speed and not for size. We compare our compact *gram-frequency-indexer* with *patch coding* and the prefered word-aligned design with the baseline Naive *gram-frequency-indexer* [1], in which *gram-frequency-indexer* is implemented as a 64-bit integer array.

Our results are shown in Table 4.2. As expected, compared with the naive *gram-frequency-indexer*, the introduction of escape array save 50% space without sacrificing retrieval speed since the size of the escape array is fairly small. Memory cost by Aligned *gram-frequency-indexer* and Compact *gram-frequency-indexer* is the same because the potion of frequency values requiring larger than 19 bits to represent is negligible.

| Approach | Memory [GB] | Compression Rate | Context Array Retrieval Speed [microsecs] |
|---|---|---|---|
| *D1+Elias* $\gamma$ | 5.54 | 0.53 | 1513 |
| *D1+Elias* $\omega$ | 4.78 | **0.58** | 1640 |
| *D1+Elias* $\delta$ | 4.79 | 0.57 | **1299** |
| *D1+block* | 4.79 | 0.57 | 1832 |

Table 4.3: Comparison of different bit-aligned methods and differential coding approaches applied on *gram-context-indexer*

| Approach | Memory [GB] | Compression Rate | Context Array Retrieval Speed [microsecs] |
|---|---|---|---|
| *D1+Elias* $\omega$ | 4.78 | 0.58 | 1640 |
| *DM+Elias* $\omega$ | 5.03 | 0.56 | 1643 |
| *D2+Elias* $\omega$ | 5.04 | 0.56 | **1639** |
| *D4+Elias* $\omega$ | 6.55 | 0.43 | 1645 |

Table 4.4: Comparison of different differential coding approaches with most compact encoding method applied on *gram-context-indexer*

## 4.4 Evaluation of the *gram-context-indexer*

We evaluated the memory usage, compression rate and retrieval speed of the *gram-context-indexer* in terms of compression and using different differential codings and compression approaches. We report compression rate as the reduction of memory compared with uncompressed data. We used Lemire's implementation [1] as a reference of *D1* and implemented *D2*, *DM*, *D4*, *VByte*, *Variant-GB*,*Variant-G8IU*, *Elias* $\gamma$ *coding*, *Elias* $\omega$ *coding*, *Elias* $\delta$ *coding*, *block coding* and *Golomb-Rice coding* from scratch. We evaluate other coding schemes mentioned in our work using Lemire's library[2]. Since the experiment is time-consuming when dataset becoming large, we carried out the evaluation only on fourgrams, that is, on contexts of bigram phrases.

Table 4.3 compares the performance of different bit-aligned compression approaches combined with *D1* differential coding. The Golomb-Rice code is not included in the tables as it took more than 8 hours to compress the data and we terminated it. Elias $\omega$ coding outperforms other bit-aligned approaches while Elias $\gamma$ coding performs the

[1]https://github.com/lemire/SIMDCompressionAndIntersection
[2]https://github.com/lemire/FastPFor.git

| Approach | Memory [GB] | Compression Rate | Context Array Retrieval Speed [microsecs] |
|---|---|---|---|
| *D1+VByte* | 6.81 | 0.43 | 716 |
| *D1+VGB* | 7.35 | 0.35 | 509 |
| *D1+Variant-G8IU* | 12.86 | **-0.07** | **226** |

Table 4.5: Comparison of different byte-aligned methods and differential coding approaches applied on *gram-context-indexer*

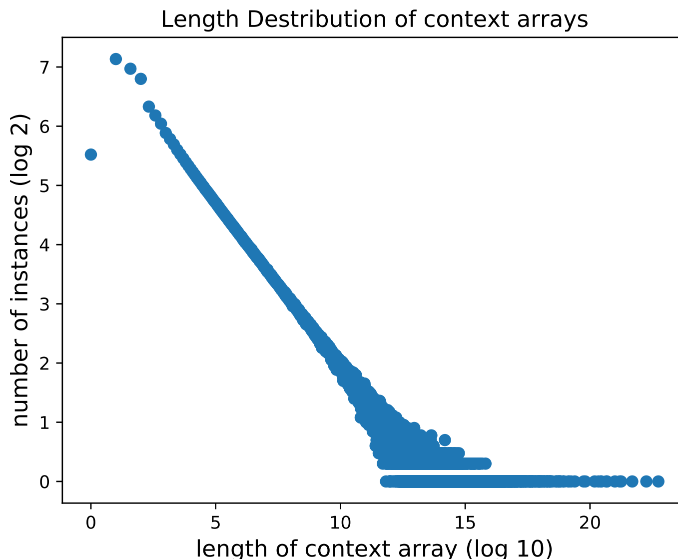



Figure 4.2: Distribution of context lengths

Note: x axis represents the length of context array in logarithm based on 2, y axis represents the number of corresponding instances in logarithm based on 10. Large y-axis value indicates high potion of context arrays with corresponding length in the dataset.

worst in compression. Elias $\delta$ coding achieves the fastest query time while achieving almost as good compression as Elias $\omega$ coding. Table 4.4 shows our comparison of combining Elias $\omega$ with different differential coding methods. As can be seen, the query speed is unaffected by the choice of differential coding method, but the compression rate varies more significantly. As a result, we chose D1 in our implementation, which achieves the best compression rate.Table 4.5 does similar evaluation for byte-aligned compression methods. We compare it with Table 4.3. *Variant-byte* pay 42% more space compared with the most compact bit-aligned method *Elias $\omega$ coding* but the retrieval speed is 2.3 times faster.

We find D4 differential coding performs worse than other three differential coding schemes, which should be slower than D4. We also find Variant-G8IU even lead to more memory cost than the uncompressed data. The result can be explained by checking the length distribution of our context arrays, which is presented in Figure 4.2. We find 71.7% of all context arrays contains less than 5 elements. Such distribution makes the SIMD-based differential coding degrade frequently. Byte-aligned approaches introduce redundant data in exchange for higher speed. Bit-aligned coding methods provide more compact storage but they sacrifice the convenience for parallelization. Simple-9 and Simple-16 are unable to handle integers exceeding $2^{28}$, so we do not evaluate them. We evaluate Simple-8b on the fourgram dataset. Then we find context array retrieval speed is 433 microseconds on average and compression rate is -0.17. Due to the property of our data set, we are unable to solely evaluate *PFor* since the majority of context arrays are short. We evaluate *PFor* as a component of the hybrid approach described at the end of Section 3.5. According to the statistical result, 85.1% of the context array contains less or equal to 8 elements, 13.4% contains more than 8 elements but less than 128 elements and 1.5% context arrays contains more than 128 elements. We tried to use Lemire's library for evaluating *PFor* but the original code only works when the length of context array is divisible by 256. We leave it as a future work to re-implement Lemire's library and evaluate a hybrid approach based on *PFor*.

## 4.5 Evaluation of Overall Performance in Speed

According to Chapter 1.2, we know the calculation process consists of four main steps: (1) Context array retrieval: We decode context arrays of corresponding target phrase and formalize them into sequences of context IDs and frequency values. (2) Context array intersection: We find phrase contexts sharing the same id from context arrays belonging to different target phrases. (3) Cosine Similarity Calculation: We calculate cosine similarity of two binary vectors indicating whether a context exists or not. (4) Other Calculation: We calculate the *RS* value and relatedness score. Then we normalize the relatedness score.

We implemented *linear intersection*, *gallopng*, *V1*, *V3* and *simd-galloping* from scratch. Before evaluating the performance of calculation speed, we show the size

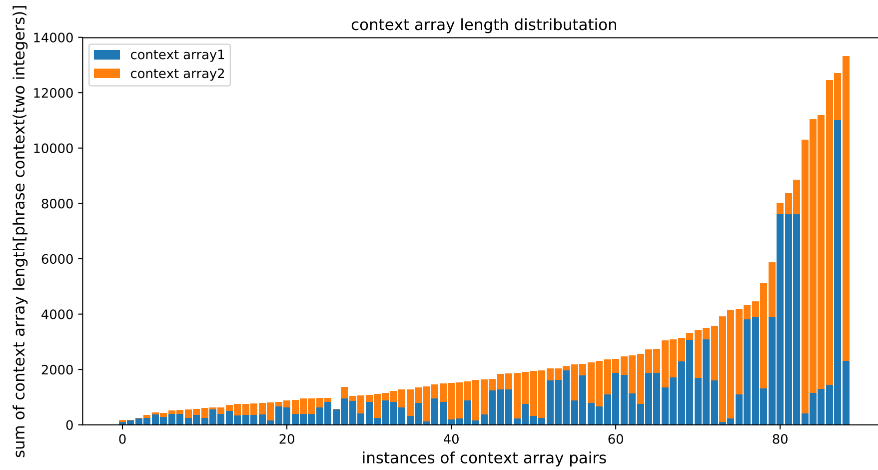



Figure 4.3: Length distribution of evaluated context arrays

| Approach | Memory [GB] | Compression Rate | Context Array Retrieval Speed [microsecs] |
|---|---|---|---|
| *D1+VByte* | 6.81 | 0.43 | 716 |
| *D4+VByte* | 6.82 | 0.43 | 733 |
| *D2+VByte* | 6.81 | 0.43 | **705** |
| *DM+VByte* | 6.81 | 0.43 | 717 |

Table 4.6: Comparison of different byte-aligned methods and differential coding approaches applied on *gram-context-indexer*

| Approach | Context retrieval | Intersection | Statistical Pruning | RS Value | Cosine Similarity | Other | Overall |
|---|---|---|---|---|---|---|---|
| hash map | 18 | 179812 | 130 | 5 | 1 | 4 | 179870 |
| linear | 19 | 141 | 130 | 6 | <1 | 1 | 297 |
| galloping | 18 | 145 | 130 | 5 | 1 | 1 | 300 |
| V1 | 18 | 90 | 130 | 5 | <1 | 1 | **244** |
| V3 | 18 | 115 | 130 | 5 | <1 | 1 | 269 |
| simd galloping | 18 | 93 | 130 | 5 | <1 | 1 | 246 |
| simd hybrid | 18 | 93 | 130 | 5 | <1 | 1 | 246 |

Table 4.7: Comparison of different intersection methods with uncompressed context arrays. Time shown are average break-down time in **microseconds** among all evaluated instances represented in steps.

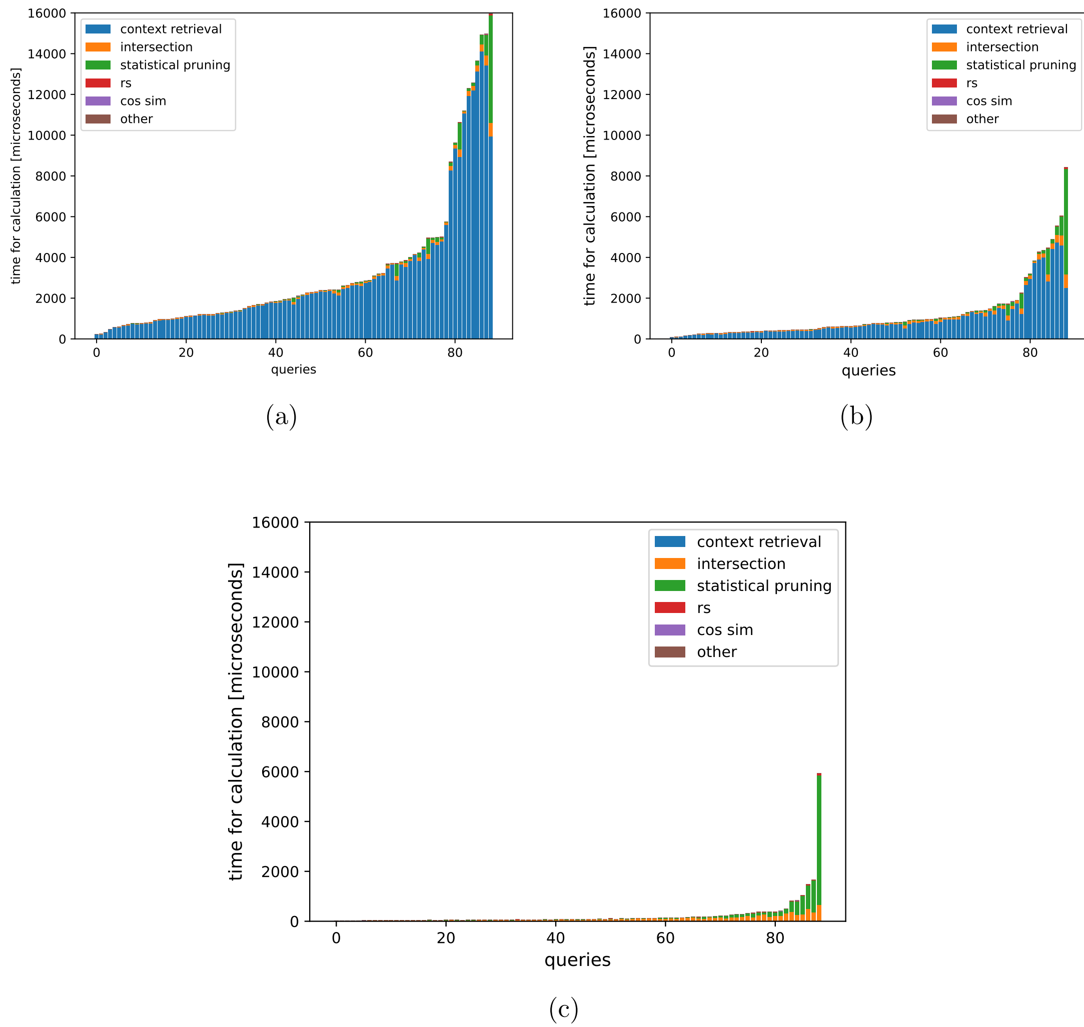
| Combination | Total Memory Cost [GB] | Time Cost per Query [microseconds] |
| --- | --- | --- |
| Compressed *gram-indexer*<br>Compressed *gram-frequency-indexer*<br>D1 differntial coding<br>Elias $\omega$ encoded *gram-context-indexer*<br>V1 intersection | 9.475 | 1852 |
| Naive *gram-indexer*<br>Aligned *gram-frequency-indexer*<br>D1 differential coding<br>Variant-GB encoded *gram-context-indexer*<br>V1 intersection | 12.045 | 722 |
| Naive *gram-indexer*<br>Aligned *gram-frequency-indexer*<br>No differential coding<br>Uncompressed *gram-context-indexer*<br>V1 intersection | 23.07 | 243 |

Table 4.8: Comparisons of framework with most compact storage and fastest calculation speed.

of context arrays we evaluated in Figure 4.3. This test set has 89 phrase pairs and it passes the Pearson Correlation test with R-value equals to 0.73, which means the relatedness score calculated from this test sets highly correlates with the golden standard. We tell from Figure 4.3 that most tested instances are imbalanced context arrays.

Then we show the break down of time spent on calculating phrase relatedness in Table 4.7 and Figure 4.4. We notice that most of time is spent on context array retrieval and context array intersection. Time spent on other calculation and cosine similarity calculation is almost trivial. Intersection based on hashmap performs the worst, and it cost nearly 1000 times of other intersection approaches. SIMD accelerated approaches performs better than non-SIMD approaches, leading to a 5% improvement in calculation speed. We expect a better performance with the hybrid approach. However, since most of the test cases are imbalance in the length of context array, the hybrid approach shows no advantage to other SIMD based approaches.

Based on our evaluation performed on current data set, we compare different combination of modules we designed in Table 4.8 using the same query set shown in Figure 4.3. Fastest combination requires 244 microseconds per query, shown in Figure 4.4 (c), which means we can answer 4098 queries per second. Compactest combination takes 1730 microseconds per query, shown in Figure 4.4 (a) which means we can answer 578 queries per second. Since our primary objective is guaranteeing fast




(a) (b) (c)

Figure 4.4: Break-down in time with different combinations of each module

Note: (a) Compressed *gram-indexer* + Compressed *gram-frequency-indexer* + D1 differential coding + Elias $\omega$ coding + V1 intersection (b) Naive *gram-indexer* + Aligned *gram-frequency-indexer* + D1 differential coding + Variant-GB encoded *gram-context-indexer* + V1 intersection (c) Naive *gram-indexer* + Aligned *gram-frequency-indexer* + No differential coding + Uncompressed *gram-context-indexer* + V1 intersection

calculation, based on the evaluation on context information extract from fourgrams, we could skip the compression step and only care about accelerating intersection and statistical pruning process. However, since most context arrays are short, we left it as future work discussing whether it is a must for introducing compression.

# Chapter 5

# Conclusion

We presented a high-performance computational framework for the phrase relatedness algorithm TrWP. With careful algorithm engineering and analysis of the data set, we made the computational framework work support fast queries using a reasonable amount of space. We found that the optimization of context array storage and retrieval is the key to achieve high efficiency because context arrays take the most memory and account for the largest portion of the query time. However, this work highly depends on the actual property of the n-gram corpus, since we need to know the distribution of context arrays and some extreme values of the pre-processed corpus in order to decide some architecture-depended optimizations and formalize some sub-problems.

We demonstrated that the combination of no compression and SIMD base intersection reaches fastest calculation speed. However, the speed still cannot support massive query for phrase relatedness. One of the future work is expected to be real-time document classification, which requires very fast phrase relatedness calculation. Consider we have two short text segments with 100 words each, the phrase relatedness of 10,000 pairs will be queried, leading to approximately 1 seconds cost solely on phrase relatedness calculation. Thus, we need faster calculation of phrase relatedness.

We realize false positives can be restricted in a trivial ratio in exchange for more compact storage and faster indexing. We may use hash functions that can map inputs to a large range of evenly distributed hash values. And we use hash values as fingerprint to represent each unigram. The length of each fingerprint deserves further study to see the effect on the correctness of TrWP.

We did not do a full comparison of different encoding schemes to find the best trade-off due to time constraints. Each codec needs to be implemented with many tricks in C++ in order to make the codec work efficiently. We surveyed other high-performance coding schemes, like *SIMD-BP-128*, [23] and *FastPFor*[15]. However, we

did not have the time to implement them ourselves and we encounter errors when applying existing library[1] for our work, the errors are possibly caused by improper inputs. Due to the lack of documents, we expect more time to read the source code and make it applicable for our task. Lemire's library guarantee the success of the compression process by allocating buffer far larger than required, and frame-based coding schemes only works with integer sequences with 256n ($n = \{1, 2, 3...\}$) elements. Besides, frame-based approaches, like *SIMD-BP-128*, aggregated data into blocks of 128 integers, but for our data, more than 71 % of the context arrays contain less than 5 elements, where frame-based coding cannot be applied. Thus, we found the framework is still not supportive for large query sets even with the uncompressed array, which indicates it is necessary to explore some other approaches for acceleration.

We tried to implement the library using efficient bit operations to make it compact and fast. However, we did not fully take advantage of GNU built-in functions and SIMD intrinsics. Some components, like context array, are implemented with STL containers for convenience, which means the evaluation results are expected to perform better if those components are rewritten in pure C.

We lack a persuasive evaluation standard like other published works [22, 23, 16], in which all works are evaluated on a standard posting list dataset and a standard query set.

This work mainly focuses on optimizing the process of calculating phrase relatedness. However, our final objective is to answer document similarity on the fly. The document similarity is calculated by algorithms performed on the matrix, but we did not look deep into it. More work is expected on optimizing the process of document relatedness calculation.

[1] https://github.com/lemire/SIMDCompressionAndIntersection.git

# Bibliography


[1] Zichu Ai, Jie Mei, Abidalrahman Moh'd, Norbert Zeh, Meng He, and Evangelos Milios. High-performance computational framework for phrase relatedness. In *Proceedings of the 2017 ACM Symposium on Document Engineering*, DocEng '17, pages 145–148, New York, NY, USA, 2017. ACM.

[2] Vo Ngoc Anh and Alistair Moffat. Index compression using 64-bit words. *Softw. Pract. Exper.*, 40(2):131–147, February 2010.

[3] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82 – 87, 1976.

[4] Paolo Boldi and Sebastiano Vigna. Codes for the world wide web. *Internet Math.*, 2(4):407–429, 2005.

[5] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and space-efficient minimal perfect hash functions. In *Proceedings of the 10th International Conference on Algorithms and Data Structures*, WADS'07, pages 139–150. Springer-Verlag, 2007.

[6] Thorsten Brants and Alex Franz. Web 1t 5-gram version 1. 2006.

[7] William B. Cavnar and John M. Trenkle. N-gram-based text categorization. In *In Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, pages 161–175, 1994.

[8] Hakan Ceylan and Rada Mihalcea. An efficient indexer for large n-gram corpora. In *The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, Proceedings of the Conference, 19-24 June, 2011, Portland, Oregon, USA - System Demonstrations*, pages 103–108, 2011.

[9] Rudi L. Cilibrasi and Paul M. B. Vitanyi. The google similarity distance. *IEEE Trans. on Knowl. and Data Eng.*, 19(3):370–383, March 2007.

[10] J. Shane Culpepper and Alistair Moffat. Efficient set intersection for inverted indexing. *ACM Trans. Inf. Syst.*, 29(1):1:1–1:25, December 2010.

[11] P. Elias. Universal codeword sets and representations of the integers. *IEEE Trans. Inf. Theor.*, 21(2):194–203, September 2006.

[12] Robert Mario Fano. *On the number of bits required to implement an associative memory.* Massachusetts Institute of Technology, Project MAC, 1971.

[13] Marcello Federico and Nicola Bertoldi. How many bits are needed to store probabilities for phrase-based translation? In *Proceedings of the Workshop on Statistical Machine Translation*, StatMT '06, pages 94–101, Stroudsburg, PA, USA, 2006. Association for Computational Linguistics.

[14] MICHAEL FLOR. A fast and flexible architecture for very large word n-gram datasets. *Natural Language Engineering*, 19(1):6193, 2013.

[15] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *Proceedings of the Fourteenth International Conference on Data Engineering*, ICDE '98, pages 370–379, Washington, DC, USA, 1998. IEEE Computer Society.

[16] David Guthrie and Mark Hepple. Storing the web in memory: Space efficient language models with constant time retrieval. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, EMNLP '10, pages 262–272, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[17] Kenneth Heafield. Kenlm: Faster and smaller language model queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, WMT '11, pages 187–197, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics.

[18] Samuel Huston, Alistair Moffat, and W. Bruce Croft. Efficient indexing of repeated n-grams. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, WSDM '11, pages 127–136, New York, NY, USA, 2011. ACM.

[19] Aminul Islam, Evangelos Milios, and Vlado Keselj. Comparing word relatedness measures based on Google $n$-grams. In *Proceedings of COLING 2012: Posters*, pages 495–506, Mumbai, India, December 2012. The COLING 2012 Organizing Committee.

[20] Aminul Islam, Evangelos Milios, and Vlado Kešelj. Text similarity using google tri-grams. In *Proceedings of the 25th Canadian Conference on Advances in Artificial Intelligence*, Canadian AI'12, pages 312–317, Berlin, Heidelberg, 2012. Springer-Verlag.

[21] G. Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science*, pages 549–554, Oct 1989.

[22] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Softw. Pract. Exper.*, 45(1):1–29, January 2015.

[23] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. Simd compression and the intersection of sorted integers. *Softw. Pract. Exper.*, 46(6):723–749, June 2016.

[24] José B. Mariòo, Rafael E. Banchs, Josep M. Crego, Adrià de Gispert, Patrik Lambert, José A. R. Fonollosa, and Marta R. Costa-jussà. N-gram-based machine translation. *Comput. Linguist.*, 32(4):527–549, December 2006.

[25] Adam Pauls and Dan Klein. Faster and smaller n-gram language models. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1*, HLT '11, pages 258–267, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics.

[26] Yehoshua Perl, Alon Itai, and Haim Avni. Interpolation search—a log logn search. *Commun. ACM*, 21(7):550–553, July 1978.

[27] Giulio Ermanno Pibiri and Rossano Venturini. Efficient data structures for massive n-gram datasets. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '17, pages 615–624, New York, NY, USA, 2017. ACM.

[28] Md. Rashadul Hasan Rakib, Aminul Islam, and Evangelos Milios. *f: Phrase Relatedness Function Using Overlapping Bi-gram Context*, pages 137–149. Springer International Publishing, Cham, 2016.

[29] R. Rice and J. Plaunt. Adaptive variable-length coding for efficient compression of spacecraft television data. *IEEE Transactions on Communication Technology*, 19(6):889–897, December 1971.

[30] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. Fast integer compression using simd instructions. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, pages 34–40, New York, NY, USA, 2010. ACM.

[31] Efstathios Stamatatos. Plagiarism detection using stopword n-grams. *J. Am. Soc. Inf. Sci. Technol.*, 62(12):2512–2527, December 2011.

[32] Alexander A. Stepanov, Anil R. Gangolli, Daniel E. Rose, Ryan J. Ernst, and Paramjit S. Oberoi. Simd-based decoding of posting lists. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 317–326, New York, NY, USA, 2011. ACM.

[33] Andreas Stolcke. Srilm-an extensible language modeling toolkit. In *Proceedings International Conference on Spoken Language Processing*, pages 257–286, November 2002.

[34] Andrew Trotman, Michael Albert, and Blake Burgess. Optimal packing in simple-family codecs. In *Proceedings of the 2015 International Conference on The Theory of Information Retrieval*, ICTIR '15, pages 337–340, New York, NY, USA, 2015. ACM.

[35] Taro Watanabe, Hajime Tsukada, and Hideki Isozaki. A succinct n-gram language model. In *Proceedings of the ACL-IJCNLP 2009 Conference Short Papers*, ACLShort '09, pages 341–344, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[36] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 401–410, New York, NY, USA, 2009. ACM.

[37] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 59–59, April 2006.

# Appendix A

## Framework Description

We attach framework UML in Figure A.1. Then we present the functionality of each major class in Figure A.2. Potential contributor of this work may find it useful comparing Figure 3.1, Figure 3.14 and Figure A.1.

We recommend potential contributor of this work to implement following modules as an asset:

- Implement custom tokenizer or find faster alternatives, since tokenization based on regex can be slow.
- Implement visualization methods for your data structure, debugging on bit-manipulations can be challenging.
- Implement custom serialization modules for your data structures, since I/O in data structure construction can be time-consuming.

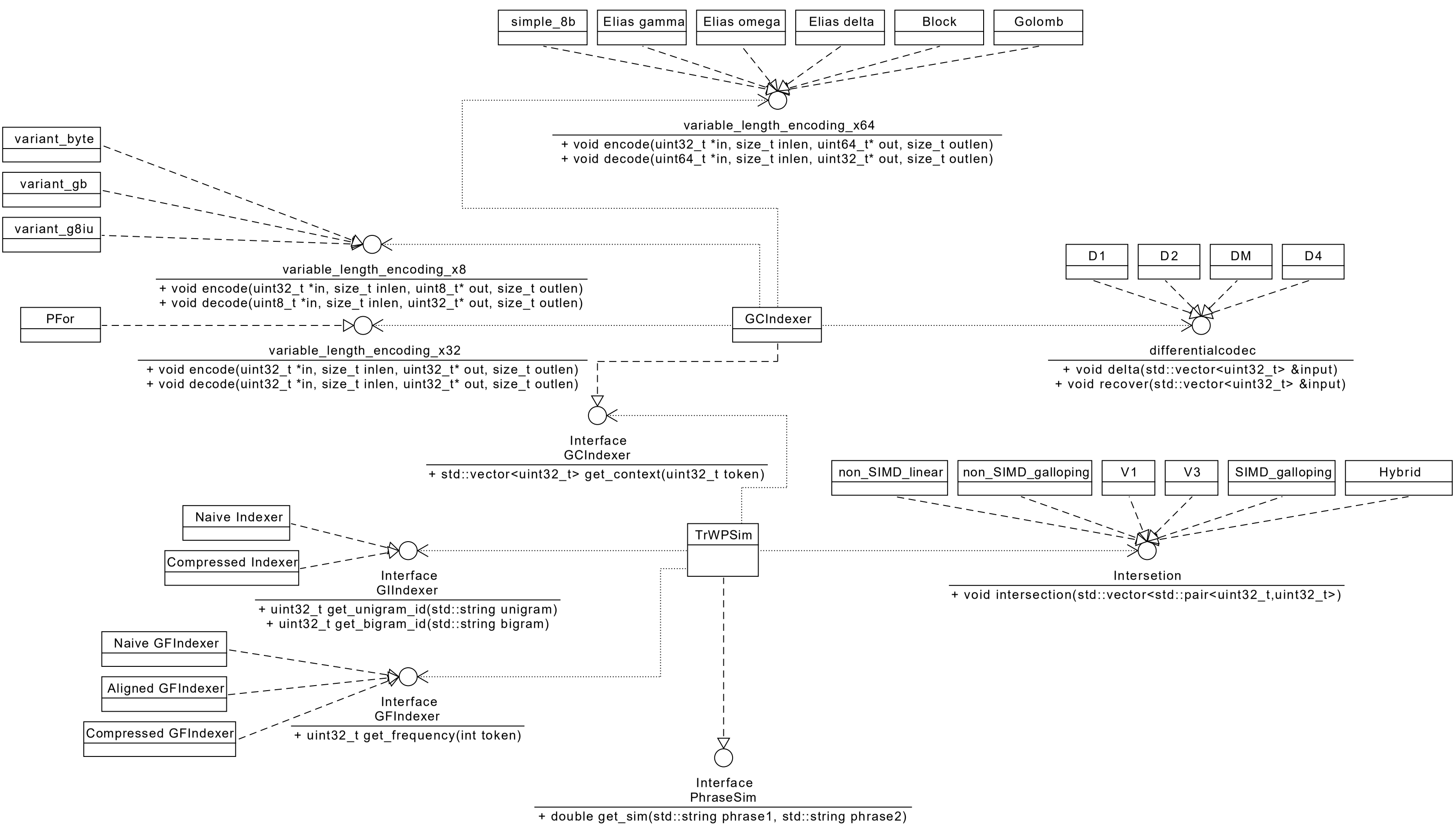



Figure A.1: Framework UML

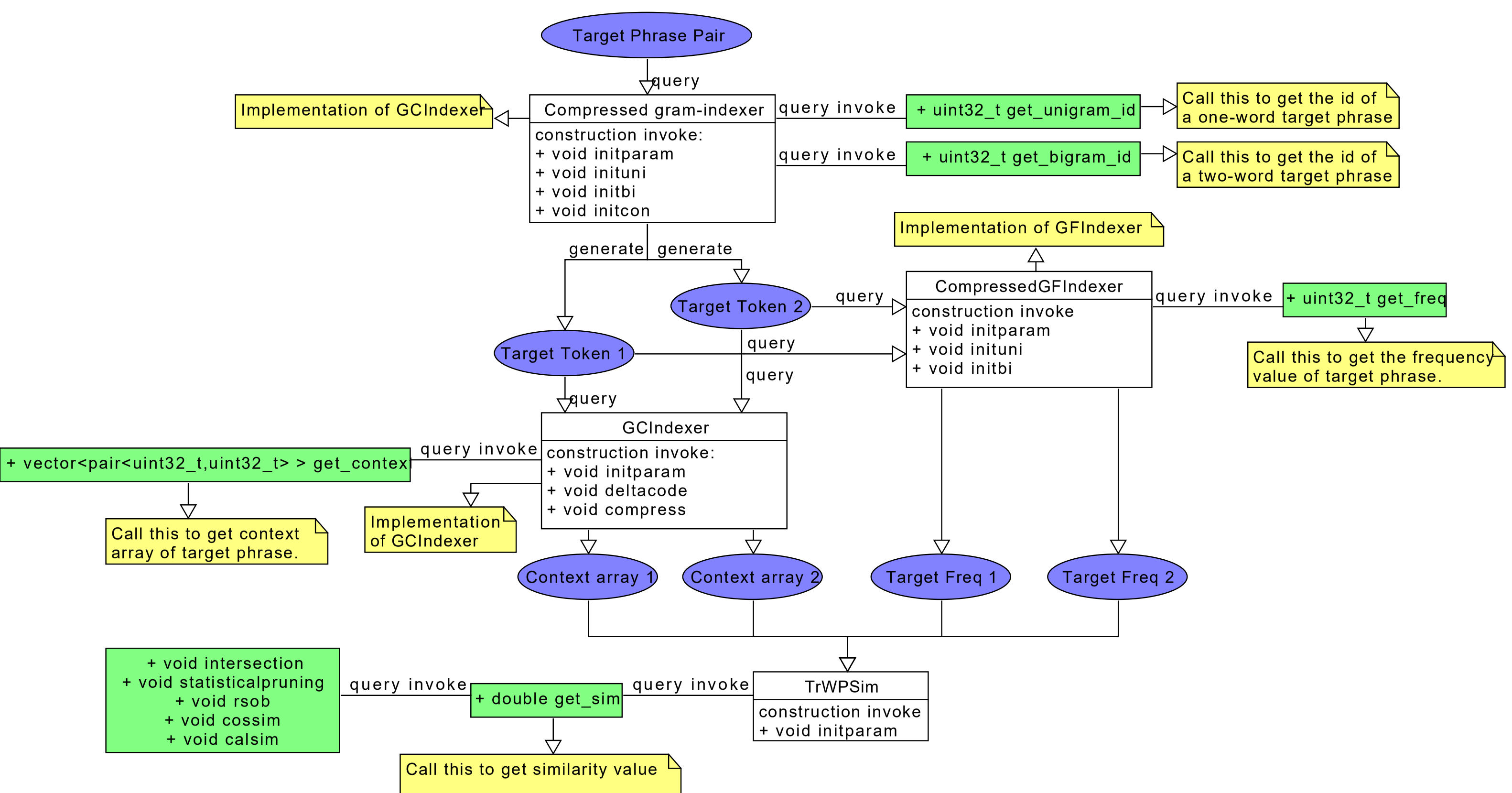



Figure A.2: Calling & Invoking Sequence