DATA COMPRESSION IN DYNAMIC SYSTEMS

by

SU CHEN

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

S. Muthukrishnan

And approved by

____

____

____

____

New Brunswick, New Jersey

May, 2008

ABSTRACT OF THE DISSERTATION

Data Compression in Dynamic Systems

by Su Chen

Dissertation Director: S. Muthukrishnan

Data compression in dynamic systems has several applications in the real world. Unlike the compression of static data, both data and intrinsic data patterns may change over time. A good compression in dynamic systems should either keep compression accurate for dynamic data or change its compression strategies for dynamic data patterns. In this thesis, we study both scenarios with applications from the real world. First, as an example of compression in dynamic systems with changing data, we discuss a lossy compression in databases, called *synopsis*, which helps the query optimizer speed up the query process. We introduce new Haar wavelet synopsis for nonuniform accuracy and time-varying data that can be generated in near linear time and space, and updated in sublinear time. The effectiveness of our data synopsis is validated against other linear time methods by using both synthetic and real data sets. Second, as an example of compression in dynamic systems with changing data patterns, we propose a novel compression algorithm, called IPzip, which compresses IP network traffic both online and offline for efficient data transfer and storage. IPzip achieves better compression ratios by learning patterns residing in both data structures and content. We also propose a methodology to monitor over time the effectiveness of the current compression and start new pattern learning when intrinsic traffic structure changes. Finally, via trace-driven experiments on network traffic obtained from Tier-1 ISPs, we validate that IPzip

achieves better performance compared to previous approaches.

Acknowledgements

These years of PhD study has been a long journey to me: I came across the ocean to a new country; I started a new graduate program and learned new cultures; I lived as a student, an intern and an employee; I met new friends and my best friend became my husband; I missed my parents so much and now I become a parent myself. There are so many things that I will treasure for my whole life. I am very glad that I reach this stage to write down my thanks to those who helped me out in this long journey.

First, I want to thank my advisor, Dr. S. Muthukrishnan, for his warm heart and continuous support. From him, I learned how to solve problems and being creative. He is the person who led me through this very special experience of PhD study. I wish I had more time with him to learn more about both research and life.

Dr. Antonio Nucci was my mentor when I did my intern at Narus. He is such a smart and energetic researcher, and you will never feel tired when working with him. I really appreciate his help on editing and proof reading of my papers. He gives me so many insightful thoughts about how to present and evaluate research works. I would also like to thank Dr. Supranamaya Ranjan and Dr. Ram Keralapura for their thoughtful discussion and support for different projects at Narus, as well as their friendship.

During the intern at Narus, I met Dr. Lixin Gao, who is also one of my committee members. I would like to thank her for many interesting lunch discusses about ideas, projects, and future technologies. And it is interesting to find out that she graduated from the most famous school in my home town.

Dr. Richard Martin is the professor who helped me when I start my study at Rutgers. As an academic advisor, he guided me through my first project. Both Dr. Richard Martin and Dr. Ahmed Elgammal are in my committee. I would like to thank them for their time to review my thesis.

I worked with Dr. Tomasz Imielinski and Dr. Don Smith at the early stage of my PhD study. I would like to thank them for those afternoon meetings and their encouragements for me to continue my study.

Most importantly, I would like to thank my parents Yilin and Caidi. They are the people who first motivated me to pursue my PhD. And they would be the happiest people to know that I reach this point in life. I am so grateful that I was born in this family with endless love from them. I know wherever I go, their love will follow. I also want to thank my husband Guilin for his spiritual support during all the hard times in these years. I could not have done it without him.

I would like to thank all my friends, readers and reviewers for their helps too. I wish you will find valuable information to your research from this thesis.

Best regards,

Su Chen

Dedication

To my loving parents,

Yilin and Caidi,

for all the wonderful things you give me.

Table of Contents

Abstract . . . ii

Acknowledgements . . . iv

Dedication . . . vi

1. Introduction . . . 1

1.1. Adaptive Coding . . . 3

1.2. Lossy and Lossless Compressions . . . 4

2. Usage-Oriented Lossy Compression in Databases . . . 6

2.1. Background . . . 6

2.2. Problem Statement and Previous Work . . . 9

2.2.1. Haar Wavelet Transform . . . 10

2.2.2. Problem Definition . . . 11

2.2.3. Related Works . . . 12

2.3. Nonuniform Point-Wise Approximation Problem . . . 15

2.3.1. 2-Step Algorithm . . . 16

2.3.2. M-Step Algorithm . . . 20

2.4. Nonuniform Range-Sum Approximation Problem . . . 21

2.4.1. Data-Mapping Algorithm . . . 22

2.4.2. Weight-Mapping Algorithm . . . 25

2.5. Tracking Dynamic Changes of Weights and Data . . . 27

2.5.1. Data Change in Data-Mapping Algorithm . . . 27

2.5.2. Data Change in Weight-Mapping Algorithm . . . 29

2.5.3. Weight Change in Data-Mapping Algorithm . . . 29

2.5.4. Weight Change in Weight-Mapping Algorithm . . . 30

2.6. Special Cases for Range-Sum Weights 30
2.7. Experiments 33
2.7.1. Point-Wise Queries 33
2.7.2. Range-Sum Approximation 39
2.8. Summary 40

3. Stream-Aware Lossless Compression in IP Networks 43
3.1. Background 43
3.2. Related Work 47
3.3. Intuitions behind IPzip 48
3.3.1. Gzip Background 49
3.3.2. Network Traffic Correlations 50
3.4. Algorithm 51
3.4.1. Compression Plan Generation 52
3.4.2. Optimal Algorithm 54
3.4.3. IPzip Compression Plan Generation Algorithms 55
3.5. System Architecture 61
3.5.1. Online Compression 63
3.5.2. Offline Compression 65
3.5.3. Traffic Pattern Change Detection 66
3.6. Experiments 68
3.6.1. IPzip vs Other Compressors: average performance 69
3.6.2. IPzip Training Phase: time required to generate a near-optimal compression plan 70
3.6.3. IPzip Compression Phase: compression ratio and speed 72
3.6.4. IPzip in a Dynamic Pattern Changing Environment 74
3.7. Summary 76

4. Conclusion 78

References 80

Vita 85

List of Tables

2.1. Prefix Sum Tables 33

2.2. Weight Reduction Table 34

List of Figures

2.1. Wavelet transform of signal A 10

2.2. W-wav algorithm is not optimal 14

2.3. Methods 15

2.4. Algorithms 17

2.5. Five intervals in which $\Psi_k[j] - \Psi_k[i-1] \neq 0$ (c_k is the normalization factor for Ψ_k) 24

2.6. Example of generation of a prefix sum table for $P[k, l]$ 25

2.7. Data-mapping and weight-mapping algorithms 26

2.8. Accuracy and efficiency 35

2.9. Time 36

2.10. Skewness 37

2.11. Relative error compare for world cup data 38

2.12. Relative error for normal(10, 100) data 40

2.13. Comparison between data-mapping and weight-mapping algorithms($\alpha = 0.5$) 41

2.14. Query skewness 42

3.1. Gzip compression example 49

3.2. Header compression example 50

3.3. Optimal algorithm 54

3.4. Compression plan generation for intra-packet correlation 56

3.5. Payload compression algorithm 58

3.6. Compression plan generation for inter-packet correlation 60

3.7. Classification tree 61

3.8. Goal of our work 62

3.9. System architecture 62

3.10. Online compression of the headers 63

3.11. Online compression of payloads 64

3.12. Offline compression example 66

3.13. Traffic pattern change detection. 67

3.14. IPzip vs Others: Compression rations for flow headers, flow payloads and IPVolume records. 69

3.15. Training data size vs Compression ratio 71

3.16. Training Phase: Time spent on training 72

3.17. Compression ratio over time for Headers (a) and Payloads (b) 73

3.18. Time to compress and decompress Headers (a) and Payloads (b) 73

3.19. Online payload compression: A look at the memory usage 74

3.20. IPzip tracking and reacting to traffic pattern changes 75

3.21. Traffic pattern change 75

Chapter 1

Introduction

Data compression has been an active area of research in the last 60 years. With the rapid progress of new technologies, an enormous amount of data is generated in the process of human-human, human-machine, and machine-machine interactions. According to a study [59] done by Berkeley researchers, the volume of information was doubled during the period of 2000-2003. About 5 exabytes (one exabyte = 10^{18} bytes) of new data was produced in 2002, among which 92% is stored on magnetic media, mostly in hard disks.

The tremendous growth in the amount of digital data brings new opportunities and challenges in data compression. First, in many applications, data are generated according to predefined standards, for example, XML syntax. A substantial gain in compression ratio can be realized, if the structure and content patterns of these data are fully exploited. Therefore, domain knowledge should be included when new compressors are designed for specific data types. Second, many data, such as network traffic data, are generated with extremely high speed and large volume. As a result, high compression speeds are also crucial in these applications in addition to super compression ratios. Third, besides storage space saving, data compression is applied in many scenarios for various purposes, such as reducing the bandwidth in data transfer and speeding up the data processing time. As a consequence, data compression is not always a static operation on static data. In these scenarios, both data and intrinsic data patterns may change over time. Such a system with dynamic data or data patterns is defined as the *Dynamic System*.

Data compression in dynamic systems can be very challenging, since compression algorithms are required to be adaptive to the dynamic environment. First, in a dynamic system with changing data, the algorithm has to keep the compression accurate over

time. Second, in a dynamic system with changing data patterns, the algorithm has to modify its compression strategies for new data patterns. We address both these scenarios in this thesis, with examples from databases and IP networks respectively.

In databases, in order to improve query efficiency, the query engine requires knowledge of the data distribution in the database tables to generate good query plans, which is called *query optimization*. Query optimization is a critical step for complex queries over large tables to reduce the CPU computation cost and I/O cost. However, data distributions can be very large and can consume excessive memory space. Therefore in practice, a DBMS (Database Management System) keeps a lossy compression of the data distribution, named *data synopsis* (see Section 2.1), to approximate the real distribution, and performs query optimization based on the synopsis. For overall better accuracy of the synopsis, frequently-queried data should be approximated more accurately than others. To address such non-uniformed queries over data, we introduce *weight* (see Section 2.1), which is the normalized query frequency for a data point or data interval. In databases, data can be inserted, deleted and updated, and the weight can change when user query changes. Therefore, the data synopsis has to be updated to maintain the approximation accuracy over time. In Chapter 2, we introduce our new dynamic data compression techniques in databases, which can generate and update synopses efficiently for dynamic data and weights.

In IP networks, in order to provide better services to customers, ISPs (Internet Service Providers) need to know the "health condition" of their networks by collecting traffic data or traffic data statistics. These feedback data compete with customers for bandwidth, since they are transferred to data centers for further analysis via commercial links. Such data are compressed before delivery to maximize the gain of information collection without affecting user satisfaction, However, the existing compression techniques are either inefficient in terms of compression ratio or too slow to work at streaming speed. Thus we design a new learning algorithm to discover the correlations inside the traffic data to improve the compression ratio without sacrificing the compression speed. Traffic data are dynamic, and their correlations may have different patterns at different times, and thus compression strategies used in the past may not be valid

for the current data. Therefore, when data pattern changes, the compression strategies must be updated accordingly. In Chapter 3, we introduce our dynamic traffic pattern learning method and how to exploit them to achieve better compression ratios.

1.1 Adaptive Coding

A task closely related to dynamic compression is *adaptive coding.* Adaptive coding methods were developed to solve the problem that statistical coding methods, such as Shannon-Fano coding and Huffman coding, require to know data distributions and build statistical models before compression. Adaptive coding is defined by Williams as "one-pass algorithms that change the way that they compress in response to the history" [48]. Unlike previous statistical coding techniques, adaptive coding methods determine the code based on the running estimation of probabilities of the data source. The decoder runs the same algorithm to synchronize with the encoder to ensure the same mapping between symbols and codes. For example, in Adaptive Huffman Coding, both the encoder and decoder build a Huffman tree, which represents the frequencies of symbols that have been observed so far. When the next symbol is read, the tree is updated by either adding a new node or changing the frequencies of the existing nodes. Then the newly updated frequency is compared with other frequencies to decide the correct position of the current node in the tree. A new symbol is sent by itself with an escape letter; others are encoded by their current position in Huffman tree. The decoder maintains the same tree as the encoder so that the received codes can be correctly translated back to their symbols. Adaptive coding can achieve compression ratios almost as good as those of their corresponding static methods. If the data source is not time-invariant, this method can outperform the statistic method because of its adaptability to changing data.

In terms of the ability to capture dynamic data patterns, adaptive coding can be classified as an instance of the second scenario, where changing data characteristics can be learned. However, the adaptive methods restrict their learning ability to data statistics only, with an assumption that the data source can be characterized as Markov models. Indeed, lots of data in real world applications have their own domain specific

structures, which cannot be easily described by traditional statistical models. These statistical coding approaches mostly ignore the high level structures of the data sets, such as the tree structures in XML data and table structures in IP traffic headers. Therefore they cannot benefit from domain knowledge to improve the compression ratios.

In this thesis, both data structure and content related patterns are exploited for better compression rates, and the focus is on exploring the dynamics of data patterns where statistical models are not satisfactory.

1.2 Lossy and Lossless Compressions

Both lossy and lossless compressions are addressed in this thesis. A brief introduction to both of them is provided here. The following is the definition of *lossy data compression* from Wikipedia: "A lossy compression method is one where compressing data and then decompressing it retrieves data that may well be different from the original, but is close enough to be useful in some way" [51]. Lossy compression is very popular in multi-media data compression, such as sound compression, image compression and video compression. In these areas, lossy compression can achieve a huge reduction in compressed size with only a small degradation in the quality in decompressed data.

Wavelet compression has been a very popular lossy compression method for about 20 years. In addition to its good compression ratio, wavelet compression has many other remarkable advantages, such as that data can be recovered at different resolutions or within certain ranges. For example, wavelet compressed maps can easily support "zoom in" and "zoom out" by adding or removing coefficients of the current level. If a small region of the map is displayed, only data in this block needs to be decompressed. In Chapter 2, we show how users can benefit from the wavelet property that data are only associated with their local coefficients to reduce compression costs.

Different from lossy compression, *lossless data compression* is defined by Wikipedia as "a class of data compression algorithms that allow the exact original data to be

reconstructed from the compressed data" [52]. Lossless compression is the major compression method for text data, where a small piece of missing information can cause major misinterpretation in the recovered data.

Algorithms in the LZ (Lempel-Ziv) family are the most widely used techniques in lossless compression. LZ compression algorithms dominate text compression due to their superior ability in rapidly identifying repeated patterns in a data stream. Another advantage of the LZ algorithms is that they are able to capture local context, thus they are more "dynamic" than traditional statistical coding methods.

In this thesis, wavelet compression is used as the lossy compressor in databases; and gzip[1] is used as the lossless compressor in IP traffic compression. Examples on how they compress data are given in Section 2.2.1 and Section 3.3.1.

Different methods are used to measure the qualities of lossy and lossless compressions. For lossy compression, an error, i.e., the distance between the original data and the approximated data is calculated for a given approximation space. The smaller the error, the better the compression result. For lossless compression, the compression quality is evaluated by the size of the compressed data. The *compression ratio* is defined here as the compressed data size divided by the original size, that is, a lower value of compression ratio represents a better compression quality.

[1]The implementation in this thesis is based on the zlib library [58], which is a variation of LZ77 algorithm.

Chapter 2

Usage-Oriented Lossy Compression in Databases

In this chapter, the method of lossy compression of time-varying data in databases is discussed. This is an interesting problem even if only the compression of static data is considered, because distinct from other lossy compression applications, in databases, data to be compressed are not equally important, i.e., frequently visited data should be approximated with better accuracy than others. Hence, a second variable "weight" is introduced, which is calculated from the query frequency, to address the issue of which data is more important than others. More interestingly, it is found that the "optimal" solution [39] used to generate data approximations based on weighted wavelet basis are actually only "optimal" on those weighted basis, i.e., for a given approximation space, it cannot produce the approximation with the smallest error. Therefore, we introduce new algorithms to further reduce approximation errors. In addition, when data and weights are dynamic, the cost of our algorithms is very low in keeping approximation accuracy, while most of the previous works overlooked this "accuracy-keeping-cost" in dynamic environments like databases. In the sections to follow, the difficulty of this problem and how it is solved are discussed in detail. We start with an introduction to the background of lossy data approximation in databases.

2.1 Background

How to query data efficiently is a fundamental issue in databases. Estimating the cost of complex queries, including CPU time, memory usage and I/O operations, requires a detailed knowledge of how the data are distributed and stored in database tables. In practice, database system maintains a concise lossy compressed data structure, called *data synopsis*, that approximates the data distribution at any point in time. The query

optimizer uses this data summary to decide how a query is to be executed in order to retrieve the requested data at minimal cost in terms of overall processing overhead. In addition to accuracy, the cost of generating and updating the synopses is a major concern, since the cost of optimization may overwhelm its benefits if the cost of synopsis is too high.

According to the characteristics of the environment at which data synopses are applied, data synopses might be classified as (i) *uniform* vs. *nonuniform* and (ii) *static* vs. *dynamic*.

Data synopses applied to data that require the same quality of approximation are known as "uniform". All data subset will be represented by the same *weight*, that reflects the fact that any data subset must be treated in exact the same way as the others. Scenarios in which at least one data subset is required to have a better quality of approximation than others, for example a data invoked in the query process more often than the others, are known as "nonuniform". In this case, the weight associated to each data subset is different; the larger the weight value, the higher the quality of its approximation to the data subset.

At the same time, data synopses can be defined for "static" or "dynamic" data structures. When data and weights do not change over time, the scenario is said to be "static". In this context, it is important to generate a good data synopsis for the data on hand. When data or weights change over time, the scenario is said to be "dynamic". Dynamic scenarios are challenging in its management. In this case, data synopses should be generated and updated over time in order to provide approximations with high accuracy, and provide meaningful information to the query optimizer at any point in time.

There are two types of data synopses that can be used to answer database queries: *point-wise approximation* and *range-sum approximation*. The former is defined for single data point query while the latter is designed for data intervals query, e.g., a set of data points. As a consequence, the point-wise approximation can be regarded as a special case of range-sum approximation when the data interval collapses into one single data point.

In this chapter, new wavelet data synopses are introduced, which can be generated in linear time and updated in sub-linear time for dynamic-data and nonuniform weights. The major contributions in this thesis are summarized as follows.

. Two linear algorithms are proposed here for the point-wise approximation problem, which are called *2-step* and *M-step*. Results show clearly how the proposed algorithms outperform the weighted wavelet algorithm [39] on approximation error on both synthetic and real data sets, with only $O(B^3)$ extra running time.

. Two linear algorithms are proposed for the range-sum approximation problem, called the *data-mapping* and the *weight-mapping*. To the best of our knowledge, these new algorithms represent the first linear algorithms invented for arbitrary weights. For n weights and B compression space, it is shown that both the time and space complexities are $O(n^2 + B^3)$, and they are linear in the weight size $O(n^2)$.

. All the algorithms proposed in this chapter are the first ones that focus not only on the generation of the data synopsis but also on its update over time. For dynamic data and weights, the time complexity of synopsis updates is $O(\log(n) + B^3)$ for point-wise approximation, and $O(n + B^3)$ for range-sum approximation.

. If the structure of given weights can be simplified, the proposed algorithm can be tuned accordingly, and the complexity can be reduced from $O(n^2 + B^3)$ to $O(n + B^3)$.

The rest of this chapter is organized as follows. Section 2.2 provides the mathematical definition of point-wise and range-sum approximations, and introduces related works. In Sections 2.3 and 2.4 two new algorithms for generating point-wise approximation, e.g. *2-step* and *M-step*, and two new algorithms for generating range-sum approximation, e.g. *data-mapping* and *weight-mapping*, are presented. In section 2.5, the novel incremental method that can minimize the cost when experiencing dynamic data and weights is introduced. Section 2.6 shows 3 examples where the proposed algorithm can simplify the inner states, to accelerate the synopses generation, if the weights

has patterns inside. As a result, the overall cost can be reduced to sublinear in term of the input weight size. In Section 2.7 the performance achieved by these algorithms are validated by using both synthetic and real-data, while Section 2.8 concludes this chapter.

2.2 Problem Statement and Previous Work

Small space synopses in database systems have been studied for decades to improve accuracy and efficiency for both query approximation and query optimization. Traditional data synopses only represent the data residing in the database, while another critical component, i.e., the characteristics of how data are queried, is missing. The accuracy of these synopses cannot truly reflect the quality of approximation when data points are not equally important.

Query Feedback systems [1, 3, 11, 14, 31, 36] have been proposed to address this issue. In these systems, data approximations are corrected by their real values returned from queries, so that the frequently visited data can be more accurate than others. Real system LEO is built to help collecting feedback information [36]. However query workload information is not fully explored in these systems, since the date accuracy only reflect how recently they are queried, but not how often they are queried.

Recently, real weights that indicate how many queries cover the data, are extracted from workloads and used to generate "usage-oriented" synopses [38, 38, 40]. In this chapter, the weights from workloads instead of query feedback are used to quantify the accuracy of our synopses.

Considerable work is available in literature for the point-wise data synopses generation by using histogram data synopses [1, 3, 29, 37, 38, 49]. However, most recently, several papers [6, 19, 39, 40, 41, 42] show the effectiveness of using wavelet decomposition in reducing large amount of data to compact sets of wavelet coefficients, termed *wavelet synopses*. Wavelet synopses have been proved to provide fast and reasonably accurate approximate answers to queries. Among wavelet synopses, the Haar wavelet synopsis has been found to be the most interesting one for database applications due

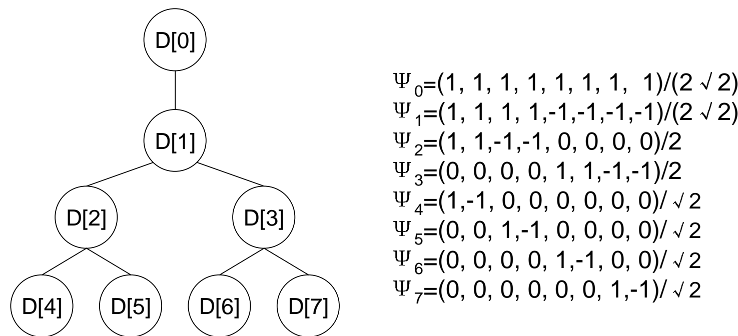


Figure 2.1: Wavelet transform of signal A

to its simple structure.

In this thesis, the focus is on the application of Haar wavelet synopsis to dynamic data and nonuniform weights for both point-wise and range-sum approximations. In next section, a brief introduction to Haar wavelet transform is given.

2.2.1 Haar Wavelet Transform

Wavelet transform is a very efficient tool in digital signal processing. It breaks data into components at different resolutions, so that both low-frequency components at long ranges and high-frequency components at short intervals can be well captured. Therefore, it outperforms the Fourier transform on non-stationary data since Fourier transform does not have the ability to associate the frequency components with their locations. The property of "component-location-association" makes wavelet transform very desirable for dynamic data, since when a data value changes, only those components associated with this changing data point need to be updated. How this property can be used to dramatically reduce the cost of updates is shown in Section 2.5.

Wavelet transform can be looked at as representing data using a wavelet basis dictionary, in which each wavelet basis is a scaled vector of a "mother wavelet". As a special case of Daubechies wavelet, Haar wavelet has a simple "mother wavelet" $[-1, 1]$. Figure 2.1 shows an example of Haar wavelet bases for a signal A with length 8, in which, wavelet bases $\Psi_1, \Psi_2, ..., \Psi_7$ are the scaled vector of $[-1, 1]$ at different regions. The wavelet base Ψ_0 comes from the "father wavelet" $[1, 1]$, which is the wavelet base for

the capture of low frequency components.

The wavelet transform computes the coefficient vector D of signal A for each wavelet basis. The coefficient vector D is the *inner product* (represented as $\otimes$) of the signal A and the wavelet vectors $\Psi = \{\Psi_0, ..., \Psi_n\}$ at different levels, e.g., $D[i] = A \otimes \Psi_i$.

Suppose A contains the following data.

$$A = (1, 9, 10, 3, 3, 5, 4, 7)$$

Then its coefficients $D[i] = A \otimes \Psi_i = \sum_{j=1}^{8} A[j]\Psi_i[j]$. The numerical values of D are summarized as follows.

$$D = [14.849, 1.414, -1.5, -1.5, -5.657, 4.950, -1.414, -2.121]$$

Signal A can be reconstructed from the products of wavelet bases and their coefficients, i.e., $A = \sum_{i=1}^{8} D[i]\Psi_i$. When fewer spaces are given in approximation, coefficients with large absolute values are chosen. For example, if there are only spaces for 6 coefficients, $D[1]$ and $D[6]$ are dropped. In this lossy approximation case, A cannot be recovered exactly. An approximation error is defined to quantify the distance between A and its approximation $\widehat{A}$, i.e., the error equals to $\sum_{i=1}^{8}(A[i] - \widehat{A}[i])^2$.

2.2.2 Problem Definition

Different from traditional Haar wavelet transform, where all data points are equally important, in the approximation discussed here, the weighted approximation errors have to be minimized, with weights representing the query frequency of the data. The mathematical definitions of the two nonuniform approximation problems are as follows.

Problem 1 (Point-Wise Approximation) *Let $A_{1\times n}$ be a generic data vector of dimension n and $\Pi_{1\times n}$ be the weight vector of dimension n that reflects the approximation quality of each data point of A, e.g. to each $A[i]$ is associated a weight $\Pi[i]$. The weight vector is normalized between $[0, 1]$, e.g. $\sum_{i=1}^{n} \Pi[i] = 1$. Let Ψ be the set of n candidate wavelets and B be the allowed number of wavelets to be used for the data synopsis. Let*

D be the set of coefficient associated to the B chosen wavelets, e.g. $D[i]$ is associated to Ψ_i.

The point-wise approximation problem can be stated as to identify the optimal subset of B wavelets from Ψ and their associated coefficients D, in order to minimize the weighted point-wise approximation error defined as:

$$\epsilon^{(P)}(A) = \sum_{i=1}^{n} \Pi[i](A[i] - \widehat{A}[i])^2 \tag{2.1}$$

where $\widehat{A}$ represents the wavelet approximation of data set A, e.g. $\widehat{A} = \sum_{i=1}^{B} D[i]\Psi_i$

Problem 2 (Range-Sum Approximation) *Let $A_{1\times n}$ be a generic data vector of dimension n, and let $A(i,j) = \sum_{k=i}^{j} A[k]$, with $i \leq j$, represent an additive function that operates on all elements of data vector A from $A[i]$ to $A[j]$. Let $\Pi_{n\times n}$ be the weight matrix of dimension $n \times n$ such that $\Pi[i,j] = 0$, $\forall i \geq j$. Each element $\Pi[i,j]$ represents the weight associated to A(i,j). The weight matrix Π is normalized between $[0,1]$, e.g. $\sum_{i=1}^{n}\sum_{j=1}^{n}\Pi[i,j] = 1$.*

The range-sum approximation problem can be stated as to identify the optimal subset of B wavelets from Ψ and their associated coefficients D, in order to minimize the weighted range-sum approximation error defined as:

$$\epsilon^{(R)}(A) = \sum_{i=1}^{n}\sum_{j=1}^{n} \Pi[i,j](A(i,j) - \widehat{A}(i,j))^2 \tag{2.2}$$

where $\widehat{A}(i,j)$ represents a generic linear function of the wavelet approximation of $A(i,j)$, e.g. $\widehat{A}(i,j) = f(\sum_{i=1}^{B} D[i]\Psi_i)$.

2.2.3 Related Works

The scenario of equally important data is well studied. In previous works, most of the wavelet synopses are generated under the assumption of uniform weights for both

point-wise and range-sum approximations [6][19][40]. For point-wise approximation, the Parseval's theorem provides a solution that applies to all orthonormal data transforms, i.e., the best approximation is achieved by the largest coefficients. For range-sum approximation, [40] presents an optimal solution on Haar wavelet synopsis.

However, the methods used to study the more general case of nonuniform weights prove to be either suboptimal in terms of approximation errors [39] or too expensive in terms of running time [17]. Matias and Urieli [39] provided the first linear algorithm, called ***W****eighted-****wav****elet* (W-wav), which is able to preserve Parseval's orthonormal condition by using a smart combination of wavelets and weights. As a result, their method provides the best synopses on weighted Haar bases, when the largest coefficients are used as synopses. Unfortunately, this approach has approximation errors that do not decrease monotonically with respect to the compression space B and the outcome approximation error may not be bounded. Figure 2.2 shows such an example. Data A is extracted from an exponential distribution. The approximation error with $B = 0$ is defined as $\epsilon_0 = \sum_i \Pi[i]A[i]^2$, i.e. all data values are assumed to be 0. The approximation error of W-wav with $B = 0$ is $\epsilon_0 = 623.49$. With more compression space $B = 2$, the error increases to 686.55. Their approximation is far from the *ideal approximation* (see Section 2.3), and worse than the *2-step method* proposed in this thesis (see Section 2.3) that reduces the error to 306.95 with $B = 2$. This problem exists not only for exponential data sets, but for all data sets that are characterized by only a few very large values.

Guha and Harb [17] studied this problem for different L_p norm errors. They proved that the best coefficients can be found by searching a bounded region specified by the optimal error of L_∞. Because the real value of this optimal error is unknown, the algorithm needs to guess out a value. The accuracy of the solution is strictly related to the cardinality of the search space. The larger the search space, the more accurate the result at the cost of a longer running time. The time complexity reaches $O(n^3)$ for L_2 error, and the space complexity is super linear in n. As a consequence, the complexity of their approach is too high for synopses used for databases.

Muthukrishnan [35] studied a special case of this problem where the weights are

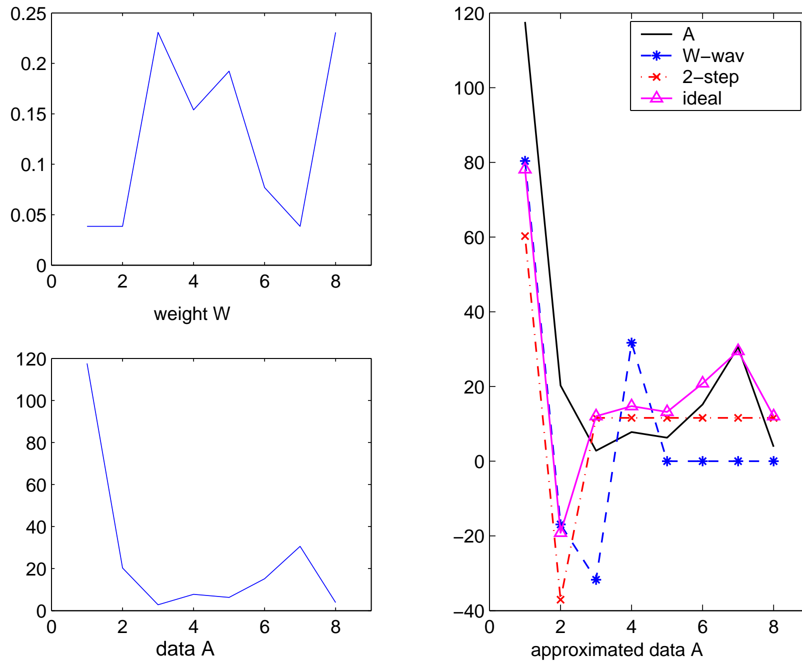


Figure 2.2: W-wav algorithm is not optimal

assumed to be organized into k intervals, and a unique weight value is associated to each interval. The running time is $O(nkB^2 \log(n))$. When $k = n$, this case collapses to the point-wise approximation problem, with a quadratic running time $O(n^2B^2)$ in terms of n.

For range-sum queries, data synopses have been studied only on uniform weights or hierarchical weights [20, 21, 29, 37]. There are very few publications on range-sum queries with nonuniform workload. Two linear algorithms are proposed in this thesis to generate range-sum synopses with arbitrary weights, and synopses updates requires only sublinear time.

All previous work has been focused only on the generation of data synopses, but ignoring the importance of a more realistic scenario in which both data and weights can change over time.

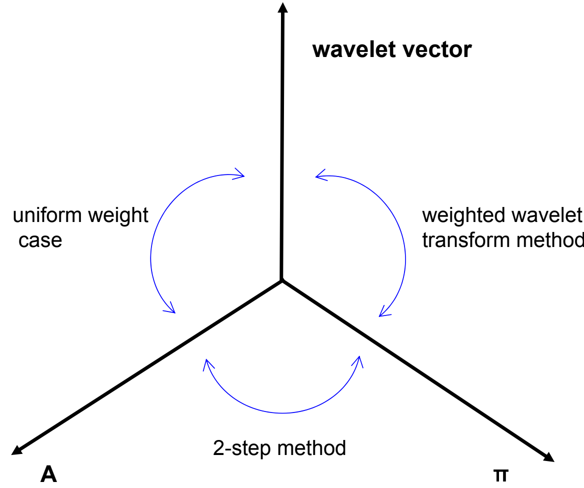


Figure 2.3: Methods

2.3 Nonuniform Point-Wise Approximation Problem

In order to approach this problem, let us define three variables that come to play into the problem (see Figure 2.3): (i) the data vector A, (ii) the wavelet vectors Ψ and (iii) the weight vector Π. For uniform weights, the data vector A is mapped to the wavelet vector Ψ in order to generate data synopses. For nonuniform weights, the mapping is arbitrary. The *W-wav* algorithm combines the weight vector with the wavelet vector by stretching wavelets vertically. The new wavelet basis is thus weight-specific. Data synopsis obtained when considering a specific weight vector might be sub-optimal for a different weight vector. As a result, the wavelet basis has to be recomputed every time a weight change is experienced, leading to large cost in database management. A different approach to solve this problem is to combine the weight vector with the data vector. The intuition behind this approach comes from a good understanding of the error function.

Given the data vector A and the weight vector Π, the error function, Equation (2.1), becomes Equation (2.3).

$$\epsilon^{(P)}(A) = \sum_{i=1}^{n}(\sqrt{\Pi[i]}A[i] - \sqrt{\Pi[i]}\widehat{A}[i])^2 = \sum_{i=1}^{n}(A_W[i] - \widehat{A}_W[i])^2 \qquad (2.3)$$

where $\widehat{A}$ represents the approximation of A, while $A_W[i] = \sqrt{\Pi[i]}A[i]$.

With this simple manipulation, the error function is reduced to a uniform-weight case. As a result, the best approximation $\widehat{A}^*$ of A can be solved from $\widehat{A}^*[i] = \widehat{A}^*_W[i]/\sqrt{\Pi[i]}$, where the optimal approximation $\widehat{A}^*_W$ of A_W exists according to Parseval's theorem. This method is referred to as the *ideal approximation* in this thesis. Unfortunately, this method is impractical because it requires the knowledge of the weight value $\Pi[i]$ for each point, and thus inadmissible because the compression space B is not large enough to store Π. One might consider to introduce an approximation of Π, but this will lead to a strong sub-optimality.

In this thesis, two algorithms called *2-step* and *M-step* are proposed based on the intuition to first select wavelets according to the optimal error ϵ^*, and then optimize their coefficients on $\widehat{A}$.

2.3.1 2-Step Algorithm

The name of this algorithm comes from its 2-step mechanism adopted to derive the data synopsis, as shown in Figure 2.4.

In *Step A*, the weighted data A_W is defined to be a point-wise product of A and $\sqrt{\Pi}$, e.g., $A_W[i] = A[i]\sqrt{\Pi[i]}$. The algorithm selects a set of wavelets with the largest B coefficients from the wavelet transform of A_W.

Step B computes the best coefficients $D[i]$ for the chosen wavelets by solving the partial differential equation of the function described in Equation (2.1), e.g. $\frac{\partial \epsilon^{(P)}}{\partial D[k]} = 0$, $\forall k \in [1, B]$.

The nice characteristics of the proposed algorithm is related to the fact that its approximation error is bounded by $|\widehat{A}[i] - \widehat{A}^*[i]| = |\widehat{A}_W[i](1 - \frac{1}{\sqrt{\Pi[i]}})|$ after the first iteration, and it will be reduced significantly at the second iteration.

The complexity of this method is linear in the input size when B is small [35]. The time and space complexity in Step A is $O(n)$ for generating A_W from A and W, and computing its wavelet transform, which is linear in the data size. Thus whether Step B can be computed in linear time is critical to keep the overall cost low. The following lemma describes why Step B requires only linear time.

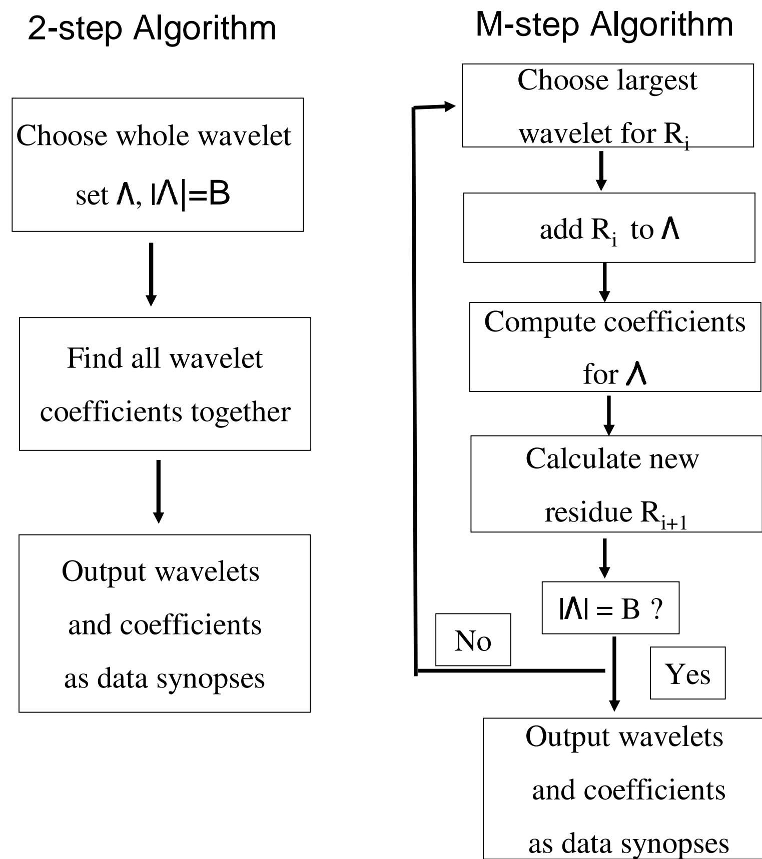


Figure 2.4: Algorithms

Lemma 3 *For any given wavelet set with size B, the best coefficient set D can be found in $O(n+B^3)$ time [35].*

Proof Let the chosen wavelet set be $\Lambda = \{\Psi_1, ..., \Psi_B\}$, the approximated data R is the product of the chosen wavelets and their coefficients, $R = \sum_{i \in \Lambda} D[i]\Psi_i$.

The approximation error ϵ is the L_2 distance between original data and approximated data:

$$\begin{aligned} \epsilon &= \sum_i \Pi[i](A[i] - R[i])^2 \\ &= \Pi \otimes ((D[1]\Psi_1 + ... + D[B]\Psi_B - A) \odot (D[1]\Psi_1 + ... + D[B]\Psi_B - A)) \end{aligned}$$

while $\otimes$ is the inner product[1] and $\odot$ is the point-wise product of two vectors[2].

To find the best $D[i]$ that minimizes the error, the following equations need to be solved.

$$\begin{cases} \frac{\partial \epsilon}{\partial D[1]} = 0 \\ ...... \\ \frac{\partial \epsilon}{\partial D[B]} = 0 \end{cases} \Leftrightarrow PD = Q$$

In which, $\frac{\partial \epsilon}{\partial D[i]} = 2\Pi \otimes (\Psi_i \odot (D[1]\Psi_1 + ... + D[B]\Psi_B - A)) = 0$. A matrix equation $PD = Q$ can be used to represent the equation above, where $P[i,j] = \Pi \otimes (\Psi_i \odot \Psi_j)$ and $Q[i] = \Pi \otimes (\Psi_i \odot A)$.

Since the time for solving $PD = Q$ is at most $O(B^3)$, the dominate part of the complexity is the time for generating matrix P and Q. An efficient way to generate P and Q is proposed here, which reduces the cost from $O(Bn)$ to $O(n)$.

This method is based on an important observation that $\Psi_i \odot \Psi_j$ can be computed in constant time for any i, j, instead of $O(n)$ time, due to the fact that Ψ_i and Ψ_j can only be in one of following 3 cases.

case 1: Ψ_i and Ψ_j are not overlapping,

[1] $z = X_{1\times n} \otimes Y_{1\times n} \Leftrightarrow z = \sum_{i=1}^{n} X[i]Y[i]$

[2] $Z_{1\times n} = X_{1\times n} \odot Y_{1\times n} \Leftrightarrow Z[i] = X[i]Y[i]$

$\Psi_i \odot \Psi_j = 0$

case 2: Ψ_i and Ψ_j are the same, i.e., $i = j$,

$\Psi_i \odot \Psi_i = \{\frac{1}{l}, ..., \frac{1}{l}\}$, where l is the length of Ψ_i's *support interval*, which is the non-zero parts of Ψ_i.

case 3: Ψ_i covers Ψ_j,

$\Psi_i \odot \Psi_j = \pm c_i \Psi_j$, where c_i is a constant that normalizes Ψ_i, named **normalization factor**. In Figure 2.1, for example, $c_7 = \sqrt{2}$.

As a consequence, $P[i, j]$ is either 0 (case 1), or the average value of Π in Ψ_i's non-zero interval (case 2), or the wavelet transform coefficient of Π scaled by $+c_i$ or $-c_i$, depending on if Ψ_j is on the left or right side of Ψ_i (case 3). For both case 2 and case 3, $P[i, j]$ can be computed from the wavelet transform of Π. It should be pointed out that, in one wavelet transform of Π, values for all $P[i, j]$ can be computed, therefore, only $O(n)$ time is used to generate matrix P.

All $Q[i]$ can be generated in $O(n)$ too, since $Q[i] = \Pi \otimes (\Psi_i \odot A) = (\Pi \odot A) \otimes \Psi_i$, which are the wavelet transform coefficients of $\Pi \odot A$.

It can be proved that a solution exists for this equation. First consider the special case that Π is not all 0 in any of the chosen wavelet Ψ_i's support interval, $i = 1 \cdots B$.

Then matrix P can be decomposed to be the product of matrix T and its transpose.

$$P = TT' = \begin{pmatrix} \sqrt{\Pi} \odot \Psi_1 \\ ... \\ \sqrt{\Pi} \odot \Psi_B \end{pmatrix} \left(\sqrt{\Pi'} \odot \Psi'_1, ..., \sqrt{\Pi'} \odot \Psi'_B \right)$$

Because $\Psi_1, ..., \Psi_B$ are linear independent, and $\sqrt{\Pi} \odot \Psi_i \neq \vec{0}$ (based on the assumption that $\exists j, j \in \Psi_i$'s support interval, $\Pi[j] \neq 0$, so the $rank(T) = rank(T') = B$. This means $rank(P) = B$ and P is non-singular, therefore $PD = Q$ is solvable.

In general, for some Ψ_i's support interval, Π are all 0. Then it is possible to set $D[i] = 0$, and reduce P to a $(B - k) \times (B - k)$ matrix, where k is the number of such

Ψ_is. Thus, this equation can always be solved.

In the worst case, we have B equations with B variables, and all $D[i]$ can be solved in $O(B^3)$ time.

Adding up the time for P,and Q generation and the $PD = Q$ solving time, the total time complexity to compute the coefficients for given wavelets is $O(n + B^3)$. □

In theory, the complexity for solving linear equation is less than $O(B^3)$. However, since n dominates the running time, the implementation here uses $O(B^3)$ to solve the equation.

In summary, the time complexity of the *2-step* algorithm is $O(n)$ for the computation of the wavelet transform and $O(n + B^3)$ for the computation of best coefficients. The space complexity is $O(n)$ for wavelet transform, and $O(B^2)$ for the selection of coefficients. It should be pointed out that the $O(n)$ space can be reused in the second step since the algorithm requires to keep only the indices of the B chosen wavelets. As a consequence, the total execution time is $O(n + B^3)$ while the space required is $O(max\{n, B^2\})$. Furthermore, its complexity becomes linear in the data size n when $B \ll n$, which is typical in database applications.

2.3.2 M-Step Algorithm

The *M-step* algorithm represents an improvement over the *2-step* algorithm based on the observation that the error obtained by the *2-step* algorithm can be further reduced by selecting new wavelets at each iterations (see Figure 2.4). The selection of the new wavelets is carried out in order to minimize the difference between the approximated data and the original data, termed *residue* and denoted as R_i where i represents the ith-iteration. The algorithm starts by setting the initial residue $R_0 = A_W$, and the chosen wavelet set to be the empty set. At each iteration i, the algorithm computes the wavelet transform for the current residue R_i, chooses the wavelet Ψ_k with the largest coefficient that does not belong to the chosen set. The new wavelet Ψ_k is added to the chosen set. The algorithm computes the new coefficient vector D for all chosen wavelets by solving $\frac{\partial \epsilon^{(P)}}{\partial D[k]} = 0$. At this point, the new residue R_{i+1} can be computed as $R_{i+1} = A_W - \sum_{k=1}^{i} D[k]\Psi_k$, and the algorithm is ready to enter the next iteration $i+1$.

The M-step algorithm continues until the cardinality of chosen wavelet set is equal to B.

At each step of this algorithm, every data point of the residue can change due to the newly added wavelet. As a result, the algorithm needs to recompute the wavelet transform for every R_i. After B steps, the running time adds up to $O(nB + B^4)$. The storage space is $O(\max\{n + B, B^2\})$, because the algorithm needs to memorize up to B chosen coefficients.

A variation of the M-step algorithm is to choose all I wavelets together at each step. This reduces the running time to $O(nB/I + B^4/I)$ at the cost of a reduced accuracy. When $I = B$, this algorithm collapses to the *2-step* algorithm.

A common property of the two algorithms proposed in this section is their approximation errors decreasing monotonously. If a "bad" wavelet is chosen at any iteration, the negative effects can always be eliminated in the next step by setting its coefficient to 0. How the two algorithms are able to efficiently reduce their estimation errors compared with *W-wav* algorithm is shown in Section 2.7.

2.4 Nonuniform Range-Sum Approximation Problem

The same idea as presented in Section 2.3 for the point-wise approximation, can be easily generalized to the case of the range-sum queries, where each weight is associated with a data interval. For a data set A with length n, there are $\frac{n(n+1)}{2}$ intervals and weights. A **Naive** method to solve the range-sum approximation problem is to generate a new data set $A^{(new)}$, in which every data point $A(i, j)$ represents an interval extracted from the original data A and computed as $A^{(new)}(i, j) = \sum_{k=i}^{j} A[k]$. As a consequence, it is straightforward to write the error function of the new data $A^{(New)}$ as $\epsilon^{(P)}(A^{(New)}) = \sum_{i,j} \Pi[i, j](A^{(New)} - \widehat{A^{(New)}})^2$, and thus the wavelet synopsis can be found by using methods for the point-wise approximation case.

Although the idea is simple, the complexity of this approach is high because the length of $A^{(New)}$ is $O(n^2)$ and thus largely exceeds the synopses space B. Because the complexity is determined by the number of intervals constituting the original data

A and the weights associated to each interval, in this section two new algorithms are proposed, named *data-mapping* and *weight-mapping*, which generate new data vectors of size n. To the best of our knowledge, these algorithms are linear algorithms used for arbitrary weights for the first time.

2.4.1 Data-Mapping Algorithm

The *data-mapping* algorithm transforms the original range-sum approximation problem into a simpler point-wise approximation problem by introducing a simple data and weight transformation. Given the original data A, a new data $A^{(DM)}$ is obtained as the partial sum of A, e.g., $A^{(DM)}[i] = \sum_{k=1}^{i} A[k]$ with $A^{(DM)}[0] = 0$.

By using $A^{(DM)}$, $A(i,j)$ can be rewritten as the difference between $A^{(DM)}[j]$ and $A^{(DM)}[i-1]$, i.e., $A(i,j) = A^{(DM)}[j] - A^{(DM)}[i-1]$. As a consequence, Equation (2.2) can be looked upon as a function of the new data $A^{(DM)}$.

$$\epsilon^{R} = \sum_{i,j} \Pi_{i,j}[(A^{(DM)}[j] - A^{(DM)}[i-1]) - (\widehat{A^{(DM)}[j]} - \widehat{A^{(DM)}[i-1]})]^2$$

Thus the new weights associated to the new data vector $A^{(DM)}$ can be expressed as Equation (2.4).

$$\Pi^{(DM)}[i] = \sum_{k=1}^{i} \sqrt{\Pi[k,i]} + \sum_{k=i+1}^{n} \sqrt{\Pi[i+1,k]} \tag{2.4}$$

In order to minimize the error function and thus obtain the best wavelet coefficients D, the differential equation obtained by setting the derivative of the error function with respect to D to 0, i.e., $\frac{\partial \epsilon^{(R)}(A^{(DM)})}{\partial D} = 0$, is solved. As a result, the problem ends up with a set of B linear equations of the form for the general wavelet coefficient vector $D[k]$, Equation (2.5).

$$
\begin{aligned}
& D[1]\sum_{i,j}\Pi[i,j](\Psi_1[j]-\Psi_1[i-1])(\Psi_k[j]-\Psi_k[i-1])+...+\\
& D[B]\sum_{i,j}\Pi[i,j](\Psi_B[j]-\Psi_B[i-1])(\Psi_k[j]-\Psi_k[i-1])\\
= & \sum_{i,j}\Pi[i,j](A^{(DM)}[j]-A^{(DM)}[i-1])(\Psi_k[j]-\Psi_k[i-1]) \qquad (2.5)
\end{aligned}
$$

If the B linear equations are solved one by one, the complexity of the procedure is $O(n^2B^2 + B^3)$. In order to reduce the time complexity of this step, Equation (2.5) is organized a in a matrix notation of the form $PD = Q$, where $P[k,l] = \sum_{i,j}\Pi[i,j](\Psi_k[j] - \Psi_k[i-1])(\Psi_l[j] - \Psi_l[i-1])$ and $Q[k] = \sum_{i,j}\Pi[i,j](A^{(DM)}[j] - A^{(DM)}[i-1])(\Psi_k[j] - \Psi_k[i-1])$.

Then a prefix sum table for $\Pi[i,j]$ is constructed to help the computation of the weights for any interval (i,j) in constant time, so that the overall complexity can be reduced to $O(n^2 + B^3)$. In the following paragraphs, the method used to reduce the polynomial cost to linear is introduced.

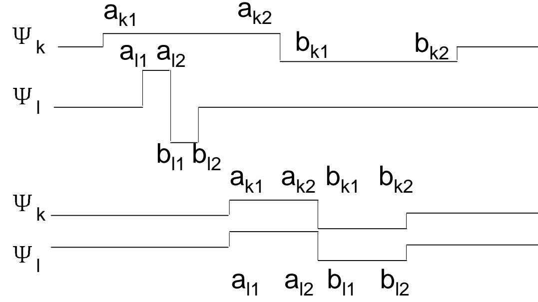
Cost for generation of matrix P, Q

First, matrix P can be computed in $O(B^2)$ due to two major observations:

(i) There is only a constant number of intervals, i.e., less than 25 intervals, in which $P[k,l] \neq 0$, since there are only 5 non-zero intervals for $\Psi_k[j] - \Psi_k[i-1]$ and $\Psi_l[j] - \Psi_l[i-1]$ (Figure 2.5).

(ii) In these intervals, $P[k,l]$ can be computed in constant time from $\sum_{i\in I,j\in J}\Pi_{i,j}$, since the normalization factors c_k and c_l are constants specified by only Ψ_k and Ψ_l.

(iii) The computation of $\sum_{i\in I,j\in J}\Pi_{i,j}$ can be carried out in constant time with the help of a prefix sum table. The prefix sum table for $\Pi_{i,j}$ can be easily generated by adding every row of the table $\Pi_{i,j}$ to its next to generate $\sum_{i\in[1,k]}\Pi_{i,j}$ for each j, then adding every column to its next right column to generate $\sum_{i\in[1,k],j\in[1,l]}\Pi_{i,j}$ (Figure 2.6). At the end of this process, the data is very well organized so that the computation of the $\sum_{i\in I,j\in J}\Pi_{i,j}$ for any interval $I = [i1, i2]$ and $J = [j1, j2]$ needs only 3 operations: $\Pi^{[i1,i2]}_{[j1,j2]} = (\Pi^{[1,i2]}_{[1,j2]} - \Pi^{[1,i2]}_{[1,j1-1]}) - (\Pi^{[1,i1-1]}_{[1,j2]} - \Pi^{[1,i1-1]}_{[1,j1-1]})$, where Π^I_J represent $\sum_{i\in I,j\in J}\Pi_{i,j}$.



If $i - 1 < a_{k1}$,
 1:if $i \leq j \leq a_{k2}$, $\Psi_k[j] - \Psi_k[i-1] = \frac{1}{c_k}$
 2:if $b_{k1} \leq j \leq b_{k2}$, $\Psi_k[j] - \Psi_k[i-1] = -\frac{1}{c_k}$
 else $\Psi_k[j] - \Psi_k[i-1] = 0$,
  since $\Psi_k[j] = \Psi_k[i-1] = 0$.
If $a_{k1} \leq i - 1 \leq a_{k2}$,
 3:if $b_{k1} \leq j \leq b_{k2}$, $\Psi_k[j] - \Psi_k[i-1] = -\frac{2}{c_k}$
 4:if $b_{k2} < j$, $\Psi_k[j] - \Psi_k[i-1] = -\frac{1}{c_k}$
 else $\Psi_k[j] - \Psi_k[i-1] = 0$,
  since $\Psi_k[j] = \Psi_k[i-1] = \frac{1}{c_k}$.
If $b_{k1} \leq i - 1 \leq b_{k2}$,
 5:if $b_{k2} < j$, $\Psi_k[j] - \Psi_k[i-1] = \frac{1}{c_k}$
 else $\Psi_k[j] - \Psi_k[i-1] = 0$,
  since $\Psi_k[j] = \Psi_k[i] = -\frac{1}{c_k}$.

Figure 2.5: Five intervals in which $\Psi_k[j] - \Psi_k[i-1] \neq 0$ (c_k is the normalization factor for Ψ_k)

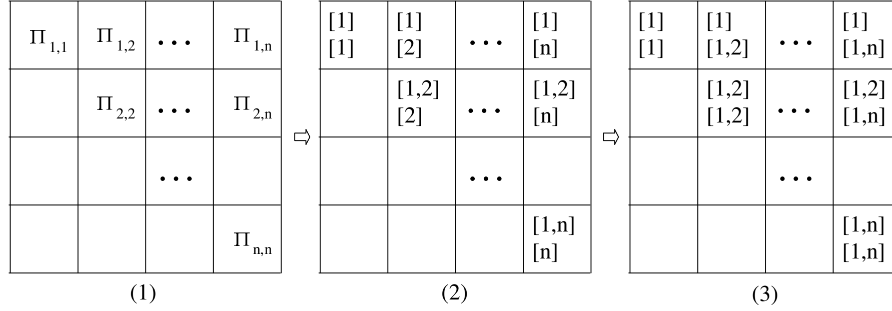


Figure 2.6: Example of generation of a prefix sum table for $P[k,l]$

Similarly, matrix Q can be computed in $O(B)$ time after constructing a prefix sum table of similar form for $\Pi[i,j]A(i,j)$.

As a result, the time required to generate matrices P and Q is only $O(B^2)$ with the help of their prefix sum tables, which need $O(\frac{n(n+1)}{2})$ to construct. Therefore, the total cost is successfully reduced from $O(n^2B^2 + B^3)$ to $O(n^2 + B^3)$, including the $O(B^3)$ time for solving the equation $PD = Q$.

2.4.2 Weight-Mapping Algorithm

In some scenarios, for example, monitoring the trends of data change through its approximation, one may prefer to approximating the original data A, instead of its prefix sum data $\widehat{A^{(DM)}}$. In this case, the new weights that are associated with A need to be identified, since the original weights $\Pi[i,j]$ are given for interval $[i,j]$.

Similar to those for the data-mapping scenario, new weights are derived from error function, Equation (2.2) by substituting $A(i,j)$ with $\sum_{i\le k\le j} A[k]$, i.e., $\Pi^{(WM)}[k] = \sum_{i\in[1,k],j\in[k,n]} \sqrt{\Pi[i,j]}$ Now the new error function $PD = Q$ ($\frac{\partial\epsilon^{(R)}}{\partial D[k]} = 0$) is specified as Equation (2.6).

$$D[1]\sum_{i,j}\Pi[i,j]\Psi_k(i,j)\Psi_1(i,j) + D[2]\sum_{i,j}\Pi[i,j]\Psi_k(i,j)\Psi_2(i,j) + ... +$$
$$D[B]\sum_{i,j}\Pi[i,j]\Psi_k(i,j)\Psi_B(i,j) = \sum_{i,j}\Pi[i,j]\Psi_k(i,j)A(i,j) \qquad (2.6)$$

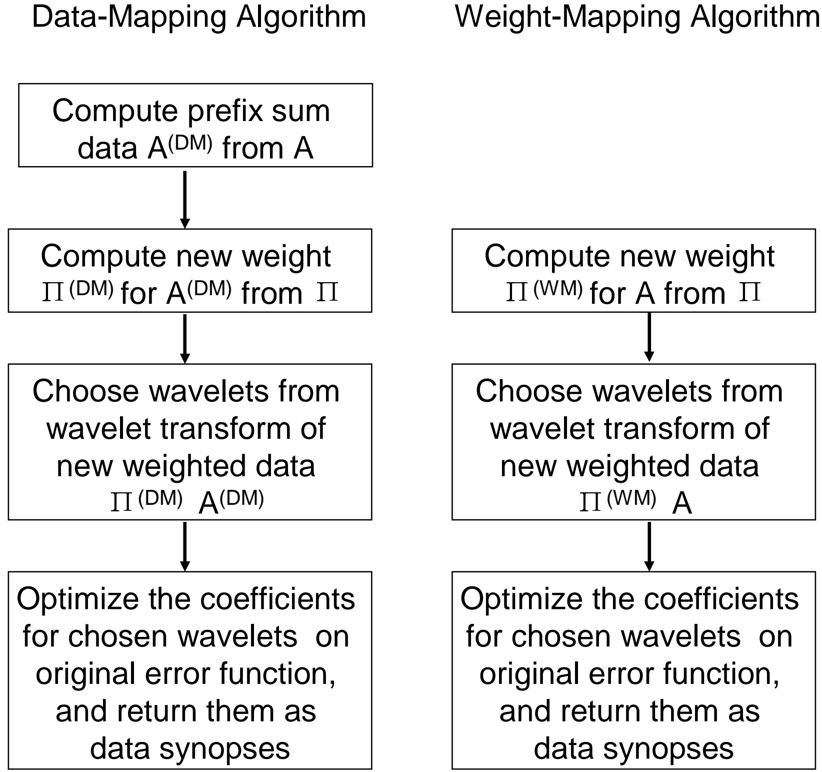


Figure 2.7: Data-mapping and weight-mapping algorithms

Similar to the scenario in the data-mapping algorithm, there are constant number of intervals, In these intervals, if the prefix table for each of $i\Pi[i,j], j\Pi[i,j], i^2\Pi[i,j], j^2\Pi[i,j]$, $ij\Pi[i,j]$, and $\Pi[i,j]A(i,j)$ is computed, matrix P, Q can be generated in $O(B^3)$ time. As a result, the total cost for weight-mapping algorithm is still $O(n^2 + B^3)$, due to the prefix table construction time $O(n^2)$ and equation solving time $O(B^3)$.

The data-mapping and weight-mapping algorithms are summarized in Figure 2.7. The major strength of these two algorithms is their capability to drastically reduce the complexity of the original problem. Indeed, they require $O(n^2)$ time and space to compute the new data set and $O(n^2 + B^3)$ time and $O(n^2 + B^2)$ space to compute wavelets and the coefficients. As a consequence, the total running time is bounded by $O(n^2 + B^3)$, while its total space is bounded by $O(n^2 + B^2)$.

2.5 Tracking Dynamic Changes of Weights and Data

The synopses generating algorithms have been introduced in Sections 2.3 and 2.4. As discussed in the introduction, in databases, data and weights change over time, especially for weights, user queries may be very different from time to time, for example, between day and night, week days and weekends. In this section, another important issue is discussed, i.e., how to keep the data synopses accurate over time when dealing with dynamic data and weights.

For point-wise approximation, when a data point or a weight value changes, only up to $\log(n)$ wavelets changes. By comparing them with the chosen B wavelets, we can find the new wavelets in $max(log(n), B)$ time. So the time cost is $O(\log(n) + B^3)$ for update, and $O(B^3)$ for finding new coefficients.

For range-sum queries, the dominant part of the cost is associated to the prefix sum table construction time $O(n^2)$. An incremental method is proposed here to keep the overall cost below $O(n+B^3)$ while requiring an extra $O(n)$ space to store updates. The case that a data point $A[t]$ is changed to $A'[t]$ is taken as the starting point.

2.5.1 Data Change in Data-Mapping Algorithm

In the data-mapping algorithm, a change in a single data point may cause changes in up to n values in the prefix sum data $A^{(DM)}$, depending on where this data point is. So the wavelet transform of $A^{(DM)}$ needs to be recomputed, which requires $O(n)$ time.

At the second step, matrix P does not change since $A^{(DM)}$ is not involved in the calculation of P (see Equation (2.5)). However, vector Q needs to be recomputed. If the prefix sum table for Q is updated, i.e., prefix sum table of $\Pi[i,j]A(i,j)$, and Q is computed from the new table, the cost is $O(n^2)$. This is because the update affects all entries $_{[1,i]}^{[1,j]}$ with $i \geq t$ or $j \geq t$, which is up to $O(n^2)$ entries.

Recall that there are only constant number of intervals, in which $Q[k] \neq 0$. Let $\{I, J\}$ be these non-zero intervals, and $v_{I,J}$ be $Q[k]$'s value in one of the interval, so that $Q[k]$ is the sum of $v_{I,J}$s over all intervals $\{I, J\}$, i.e., $Q[k] = \sum_{I,J} v_{I,J}$.

Suppose the difference between the new and the old values is $\delta_t^A = A'[t] - A[t]$, it

can be proven that the new $Q'[k]$ can be computed from the old $Q[k]$ in constant time.

Let's start from the update in one of its non-zero intervals, $I = [i1, i2], J = [j1, j2]$. Because only when $t \in I, J$, the new update is reflected in the partial sum data, i.e., when $i \leq j < t$ or $t < i \leq j$, $A'(i,j) = A(i,j)$, and when $i \leq t \leq j$, $A'(i,j) = A(i,j) + \delta_t^A$.

Therefore, the new value $v'_{I,J}$ can be separated into two parts: one includes $A'[t]$, i.e., $i \in I \cap [1,t], j \in J \cap [t,n]$, and the other does not, i.e., $i \in I \cap [1,t), j \in J \cap [1,t)$ and $i \in I \cap (t,n], j \in J \cap (t,n]$.

$$\begin{aligned}
v'_{I,J} &= \sum_{i \in I \cap [1,t), j \in J \cap [1,t)} \Pi[i,j]A(i,j) + \sum_{i \in I \cap (t,n], j \in J \cap (t,n]} \Pi[i,j]A(i,j) \\
&+ \sum_{i \in I \cap [1,t], j \in J \cap [t,n]} \Pi[i,j]A'(i,j) \\
&= \sum_{i \in I \cap [1,t), j \in J \cap [1,t)} \Pi[i,j]A(i,j) + \sum_{i \in I \cap (t,n], j \in J \cap (t,n]} \Pi[i,j]A(i,j) \\
&+ \sum_{i \in I \cap [1,t], j \in J \cap [t,n]} \Pi[i,j]A(i,j) + \sum_{i \in I \cap [1,t], j \in J \cap [t,n]} \Pi[i,j]\delta_t^A \\
&= \sum_{i \in I, j \in J} \Pi[i,j]A(i,j) + (min(i2,t) - i1) * (j2 - max(j1,t))\delta_t^A \sum_{i,j} \Pi[i,j] \\
&= v_{I,J} + (min(i2,t) - i1) * (j2 - max(j1,t))\delta_t^A \sum_{i,j} \Pi[i,j] \qquad (2.7)
\end{aligned}$$

As a result, $v'_{I,J}$ can be computed from $v_{I,J}$ in $O(1)$ time (Equation (2.7)) with the help of the prefix sum table of $\Pi[i,j]$, so that $Q'[k]$ can be computed from $Q[k]$ in constant time. Since I, J are only a small constant, the total update time for Q is $O(B)$.

In general, when there are x number of changes in the data, the updating time of Q is $O(B + x)$ if adding changes to Q one by one.

When $x < n$, the $Q[k]$ is computed by using the original table, then updates to $Q'[k]$: The update time contains the following components.

(1) Recomputing the wavelet coefficients: $O(n)$.

(2) Choosing B wavelets and computing P and Q: $O(B^2)$.

(3) Updating Q to Q' for all δ_t^A: $O(B + x)$.

(4) Solving the new equation $PD = Q'$: $O(B^3)$.

When the number of changes reaches n, the prefix sum tables is computed, and the algorithm is run on the new tables, which takes $O(n^2 + B^3)$.

So the amortized cost is $O(n + B^3)$, because $O(\frac{1}{n}[\sum_{x=1}^{n-1}(n + B^3 + x) + (n^2 + B^3)]) = O(n + B^3)$

2.5.2 Data Change in Weight-Mapping Algorithm

The only difference between weight-mapping and data-mapping algorithms exists in the first step. When $A[t]$ changes, the time involved in finding the new wavelets is $O(\log(n))$ instead of $O(n)$. At the second step, only vector Q's prefix table changes with $A[t]$, since $P[k,l] = \sum_{i,j} \Pi[i,j]\Psi_k(i,j)\Psi_l(i,j)$, and $Q[k] = \sum_{i,j} \Pi[i,j]\Psi_k(i,j)A(i,j)$. The amortized cost is still $O(n+B^3)$, due to $O(\frac{1}{n}[\sum_{x=1}^{n-1}(\log(n)+B^3+x)+(n^2+B^3)]) = O(n+B^3)$.

If a weight value $\Pi[i,j]$ changes to $\Pi'[i,j]$, where $\sum_{i,j} \Pi'[i,j] = \sum_{i,j} \Pi[i,j] + \delta^{\Pi}_{i,j}$, both $P[k,l]$ and $Q[k]$ need to be updated.

By applying the same method in data updates, $P[k,l]$ and $Q[k]$ can be computed from the original prefix sum tables. Then $\delta^{\Pi}_{i,j}$ or $\delta^{\Pi}_{i,j}A(i,j)$ is added to get $P'[k,l]$ and $Q'[k]$.

2.5.3 Weight Change in Data-Mapping Algorithm

In data-mapping algorithm, a single update in the original weights Π leads to at most two changes in $\Pi^{(DM)}$ (Equation (2.4)). Thus only $O(\log(n))$ wavelet coefficients need to be re-calculated. Because of these new coefficients, the equation $PD = Q$ has to be re-generated, which requires $O(B^2)$ time. Similar to the case in data changes, for x number of changes in weights, the update time from $PD = Q$ to $P'D = Q'$ is $O(B^2+x)$ for $P[k,l]$ and $O(B+x)$ for $Q[k]$. The equation solving time is $O(B^3)$. So the amortized time for x updates is O($\frac{1}{n}[\sum_{x=1}^{n-1}(\log(n) + B^3 + x) + (n^2 + B^3)]) = O(n + B^3)$.

2.5.4 Weight Change in Weight-Mapping Algorithm

For weight changes, the only difference between data-mapping and weight-mapping is that a single update in weights may cause up to n changes in weighted data $\Pi^{(WM)} \odot A$. So $O(n)$ wavelet transform time is required, and amortized cost is still $O(n + B^3)$, because $O(\frac{1}{n}[\sum_{x=1}^{n-1}(n + B^3 + x) + (n^2 + B^3)]) = O(n + B^3)$.

In summary, the incremental method proposed in this thesis computes new values by exploiting the computed values in the history, thus they do not need be calculated from scratch. Therefore, the update costs for both data-mapping and weight-mapping are successfully reduced from linear to sub-linear in time.

2.6 Special Cases for Range-Sum Weights

In the previous sections, the most general cases for the weights have been discussed, i.e., each point of the weight may differ from another. However, in some special scenario, weights may have some nice structure, For example, in online query approximation, user may specify their own approximation quality of a data interval in terms of the size or the values contained in the data interval. Then it is possible to capture these structures in weights to reduce the running time. In this section, three of such cases are discussed.

Uniform weights There is only one unique weight for all intervals. $\forall i, j, k, l$, $\Pi[i, j] = \Pi[k, l]$. This is the unweighted range-sum problem.

Uniform length weights The weights are the same if their interval lengths are the same. $\Pi[i, j] = \Pi[k, l]$, if $j - i = l - k$, and $\Pi[i, j] = \Pi[k, l] + h$, if $j - i = l - k + 1$, where h is the unit weight difference. We use $\Pi_{|l|}$ to represent the weight for an interval with length l, so $\Pi_l = \Pi_1 + (l - 1) * h$. In this scenario, the importance of the region is decided by how many data it covers.

Hierarchical sum weights There are a unique weight $\Pi_{i,i}$ for each i. All other weights can be derived from them, $\Pi_{i,j} = \sum_{k=i}^{j} \Pi_{k,k}$. This is the case that range weights are sum of single data point weight in it.

Recall that in Section 2.4, after we construct the prefix tables, the computational

costs for generating P, Q and solving $PD = Q$ are independent of n. The dominant part of the overall cost $O(n^2 + B^3)$ turns out to be table construction time $O(n^2)$.

In these special cases, weights for different points and intervals are not totally independent. This offers us an opportunity to reduce the table size from $O(n^2)$ to $O(n)$.

To begin with, a simple but important lemma in cost reduction is introduced here.

Lemma 4 *For vector* $V = \{V[1], V[2], ..., V[n]\}$*, the sum of any arithmetic series* $f(k, d, i, j) = kV[i] + (k + d)V[i + 1] + ... + (k + (j - i)d)V[j]$ *can be computed in* $O(1)$ *time after* $O(n)$ *preprocessing.*

Proof The following two prefix sum table from vector V can be generated in $O(n)$ time.

$$S_1^V = \{V[1], V[1] + V[2], ..., V[1] + ... + V[n]\}$$
$$S_2^V = \{V[1], V[1] + 2V[2], ..., V[1] + ... + nV[n]\}$$

Then any $f(k, d, i, j)$ can be computed in 9 operations as in Equation (2.8).

$$\begin{aligned} f(k, d, i, j) &= kV[i] + (k + d)V[i + 1] + ... + (k + (j - i)d)V[j] \\ &= kV(i, j) + d(V[i + 1] + 2V[i + 2] + ... + (j - i)V[j]) \\ &= k(S_1^V[j] - S_1^V[i - 1]) + d(S_2^V[j] - S_2^V[i]) - id(S_1^V[j] - S_1^V[i]) \end{aligned} \tag{2.8}$$

□

At high level, for all these special cases, because the weights for different intervals are not independent of each other, the prefix sum tables of size $O(n^2)$ can be reduced to a vector of size $O(n)$, which is similar to S_1^V, S_2^V.

It can be proved that with the help of these new prefix vectors, entries in matrix P and vector Q can still be computed in $O(1)$ time. The running time for all other parts of the algorithm remains same, except that the table construction time is reduced from

$O(n^2)$ to $O(n)$. So the total running time is reduced to $O(n + B^3)$.

In this section, only the prefix vector for Q, i.e., the prefix vector of $\sum_{i\in I,j\in J} \Pi_{i,j} A(i,j)$ is discussed. The prefix vector for P (prefix vector of $\sum_{i\in I,j\in J} \Pi_{i,j}$) can be looked upon as a special case when $A(i,j) = 1$. The uniform weight case is used as an example to show how to reduce the prefix table size, and the results for the other cases is summarized in Table 2.2.

It should be pointed out that only two cases of the overlapping of intervals, e.g., $I \cap J = \phi$ and $I = J$, are needed to consider to further simplify the computation, because if $I \cap J \neq \phi$, and $I \neq J$, then $j1 \leq i2$. This interval can be divided into 3 parts: $I1 = [i1, j1-1], J1 = [j1, j2]$, $I2 = [j1, i2], J2 = [j1, i2]$ and $I3 = [j1, i2], J3 = [i2+1, j2]$. In these new intervals $I1 \cap J1 = \phi$, $I2 = J2$ and $I3 \cap J3 = \phi$. This division creates 3 sub-intervals, each of them can be solved by prefix vectors defined in Table 2.1.

Example: prefix table reduction for uniform weights

In the uniform weight case, $\Pi_{i,j}$ is a constant for all intervals. Here $S[i]$ is defined as $S[i] = \sum_{k=1}^{i} A[k]$.

$$
\begin{aligned}
& \sum_{i\in I,j\in J} A(i,j) \\
= & \sum_{i\in I,j\in J} (S[j] - S[i-1]) \\
= & \sum_{i\in[i1,i2]} \Big(\sum_{j\in[j1,j2]} S[j] - \sum_{j\in[j1,j2]} S[i-1] \Big) \\
= & \sum_{i\in[i1,i2]} ((S[j1] + ... + S[j2]) - (j2 - j1 + 1) S[i-1])) \\
= & (i2 - i1 + 1)(S[j1] + ... + S[j2]) - (j2 - j1 + 1)(S[i1-1] + ... + S[i2-1]) \\
= & (i2 - i1 + 1)(S3[j2] - S3[j1-1]) - (j2 - j1 + 1)(S3[i2-1] - S3[i1-2])
\end{aligned}
\tag{2.9}
$$

where interval $I = [i1, i2], J = [j1, j2]$, and $S3$ is defined in Table 2.1.

As a result, if a prefix sum vector $S3$ is pre-computed, the prefix sum $\sum_{i\in I,j\in J} A(i,j)$

Table 2.1: Prefix Sum Tables

$S_1 = \{\Pi[1,1], \Pi[1,1] + \Pi[2,2], ..., \Pi[1,1] + ... + \Pi[n,n]\}$
$S_2 = \{\Pi[1,1], \Pi[1,1] + 2\Pi[2,2], ..., \Pi[1,1] + ... + n\Pi[n,n]\}$
$S_3 = \{S[1], S[1] + S[2], ..., S[1] + ... + S[n]\}$
$S_4 = \{S_1[1]S[1], S_1[2]S[2], ..., S_1[n]S[n]\}$
$S_5 = \{S_1[0]S[1], S_1[1]S[2], ..., S_1[n-1]S[n]\}$
$S_6 = \{S_2[1]S[1], S_2[2]S[2], ..., S_2[n]S[n]\}$
$S_7 = \{\Pi_{1,1}S_3[1], 2\Pi_{2,2}S_3[2], ..., n\Pi_{n,n}S_3[n]\}$
$S_8 = \{\Pi_{1,1}S_3[1], \Pi_{2,2}S_3[2], ..., \Pi_{n,n}S_3[n]\}$
$S_9 = \{S_3[1], S_3[1] + S_3[2], ..., S_3[1] + ... + S_3[n]\}$
$S_{10} = \{S_3[1], S_3[1] + 2S_3[2], ..., S_3[1] + ... + nS_3[n]\}$

for any interval I, J can be computed in constant time as in Equation (2.9). Therefore $\sum_{i \in I, j \in J} \Pi[i,j]A(i,j) = \Pi[i,j] \sum_{i \in I, j \in J} A(i,j)$ can be computed in $O(1)$ time from $\sum_{i \in I, j \in J} A(i,j)$ since $\Pi[i,j]$ is a constant.

In this section, the algorithms proposed are shown to be able to be applied to different special weights, with a dramatic complexity reduction. This is because the algorithms capture the real complex component, i.e., the $O(n^2)$ weights, in this problem through its prefix sum table. When the weights is simplified, the cost of our algorithm can be simplified accordingly.

2.7 Experiments

In this section, the accuracy and efficiency of the proposed lossy compression in databases on different data sets are demonstrated. Here the relative error is defined as $RE = \frac{\epsilon_B}{\epsilon_0} = \frac{\sum_i \Pi[i](A[i] - \widehat{A}[i])^2}{\sum_i \Pi[i](A[i])^2}$ for point-wise approximation, and $RE = \frac{\epsilon_B}{\epsilon_0} = \frac{\sum_{i,j} \Pi_{i,j}(A^{[i,j]} - \widehat{A}^{[i,j]})^2}{\sum_{i,j} \Pi_{i,j}(A^{[i,j]})^2}$ for range-sum approximation, where ϵ_B is the absolute approximation error with B buckets. The experiments are done by using a Linux machine with 2.80GHz processor and 2047 MB memory.

2.7.1 Point-Wise Queries

In this section, the 2-step (see Section 2.3), M-step (see Section 2.3) and W-wav algorithms (see Section 2.2.3) are compared by using both synthetic and real data sets.

Table 2.2: Weight Reduction Table

$I \cap J = \phi$	$LS_j = ((L_2+1)\Pi_{\lvert L_1 \rvert} + \frac{L_2(L_2+1)h}{2} S_3[j1+L_2] - [(\Pi_{\lvert 1 \rvert} + h(L_1-1))(S_9[j1+L_2] - S_9[j1-2]) + h[(1-j1)(S_9[j1+L_2] - S_9[j1-1]) + (S_{10}[j1+L_2] - S_{10}[j1-1])$
$I \cap J = \phi$	$LS_i = \Pi_{\lvert L_1 \rvert}(S_9[i1+L_2] - S_9[i1-1]) + h(1-i1)(S_9[i1+L_2-1] - S_9[i1-1]) + (S_{10}[i1+L_2-1] - S_{10}[i1-1]) + \Pi_{\lvert L_1 \rvert} \frac{L_2(L_2+1)h}{2} S_3[i1-1]$
$I = J$	$LS_j = ((L_2+1)\Pi_{\lvert 1 \rvert} + \frac{L_2(L_2+1)h}{2} S_3[i2] - [\Pi_{\lvert 1 \rvert}(S_9[i2] - S_9[i1-2]) + h[(1-i1)(S_9[i2] - S_9[i1-1]) + (S_{10}[i2] - S_{10}[i1-1])$
$I = J$	$LS_i = \Pi_{\lvert 1 \rvert}(S_9[i2] - S_9[i1-1]) + h(1-i1)(S_9[i2-1] - S_9[i1-1]) + (S_{10}[i2-1] - S_{10}[i1-1]) + \Pi_{\lvert L_1 \rvert} \frac{L_2(T_2+1)h}{2} S_3[i1-1]$
$I \cap J = \phi$	$HS_j = [(1-i1)(S_1[i2] - S_1[i1-1]) + (S_2[i2] - S_2[i1-1])](S_3[j2] - S_3[j1-1]) + (i2-i1+1)[S_4[j2] - S_4[j1-1] + (2S_1[j1-1] - S_1[i2])(S_3[j2] - S_3[j1-1])]$
$I \cap J = \phi$	$HS_i = (j2-j1+1)[S_1[j1-1](S_3[i2] - S_3[i1-1]) - (S_5[i2] - S_5[i1-1])] + [(j2+1)(S_1[j2] - S_1[j1-1]) - (S_2[j2] - S_2[j1-1])]S^{[i1,i2]}$
$I = J$	$HS_j = (S_6[i2] - S_6[i1-1]) - S_2[i1-1](S_3[i2] - S_3[i1-1])$
$I = J$	$HS_i = (i2+1)(S_8[i2] - S_8[i1-1]) - (S_7[i2] - S_7[i1-1]) + (i2-i1+1) * (S_1[i2] - S_1[i1-1]) + i1(S_1[i2] - S_1[i1-1]) - (S_2[i2] - S_2[i1-1])$

LS_j and HS_j stands for $\sum_{i \in I, j \in J} \Pi_{i,j} S[j]$, in uniform length weights case and hierarchical sum weights case; LS_i and HS_i stands for $\sum_{i \in I, j \in J} \Pi_{i,j} S[i]$ in those cases. L_1 is start interval length, $L_1 = j1 - i1 + 1$. L_2 is max shift length, $L_2 = min\{i2 - i1, j2 - j1\}$.

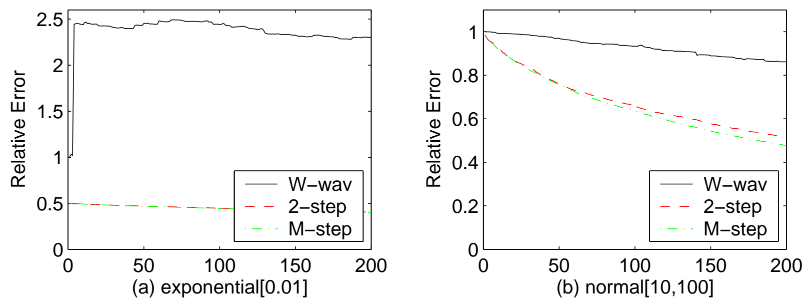


Figure 2.8: Accuracy and efficiency

In these experiments, $I = 1$ is used for the M-step method. For other I values, the approximation error is between the error of 2-step algorithm, and the error of M-step algorithm with $I = 1$.

Synthetic Data

The synthetic data set contains 4,096 data points, and they are extracted from a normal, an exponential and a uniform distribution, respectively. The weights are extracted from a zipf distribution with $\alpha = 0.2$, 0.5 and 0.8. Next the 2-step and M-step algorithms for point-wise approximation are compared against the W-wav algorithm in terms of *accuracy*, *efficiency*, *time* and *skewness*.

Accuracy The results for the accuracy are shown in Figure 2.8(a) for the data set extracted from an exponential distribution with parameter $\lambda = 0.01$. A major consideration to make is related to the shape of these curves: the error for the W-wav algorithm jumps to $2.5\epsilon_0$ and remains there even after $B = 200$, while both the 2-step and M-step algorithms always keep the error under $0.5\epsilon_0$. The reason for such behavior is related to the fact that when a data set contains very large values, the W-wav algorithm takes more wavelets from this region than necessary, while the 2-step and M-step algorithms can use 0 as coefficients for the “unwanted” wavelets.

Efficiency The efficiency results are shown in Figure 2.8(b) for the data set extracted from a normal distribution with $mean = 10$ and $variance = 100$. Notice here that the 2-step and M-step algorithms are capable of reducing the error to $0.5\epsilon_0$ around $B = 200$, while the error of W-wav algorithm is still as high as $0.85\epsilon_0$. This is caused

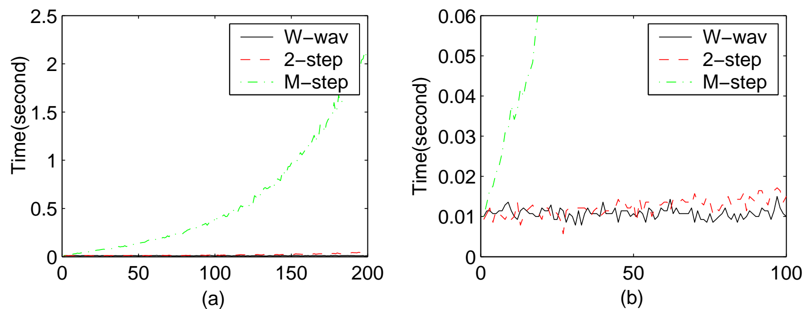


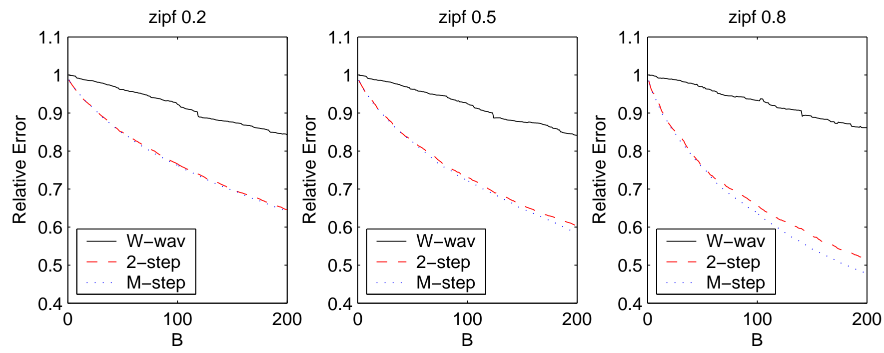
Figure 2.9: Time

by cancelation among overlapped wavelets using the original coefficients. The 2-step and M-step algorithms can lower this side-effect by finding the best coefficient for each wavelet.

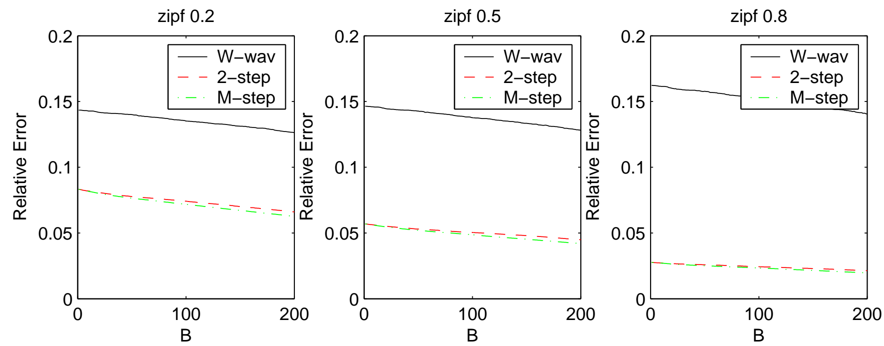
Time The running times of the three algorithms are shown in Figure 2.9(a) and Figure 2.9(b). The M-step method requires a very long running time, which reaches 2 second when $B = 200$ (Figure 2.9(a)). With a plot at a smaller scale (Figure 2.9(b)), it can be observed that the running time for the W-wav algorithm is $O(n)$, and it is constant for all B. The running time of the 2-step algorithm is bounded by $O(n + B^3)$. For $B^3 < n$, i.e., $B < 16$, there is no difference in running time between the 2-step and the W-wav algorithms. Even when B reaches 200, $200^3 \gg 4,096$, the extra time for the 2-step algorithm is only 0.05 second.

Skewness Last but never the least, in Figure 2.10(a) and Figure 2.10(b), the three algorithms are compared while the weight distributions are changed (zipf 0.2, 0.5 and 0.8). The data sets are extracted from normal and uniform distributions, respectively. It is important to notice here how the performance of the W-wav method decreases as weights become more skewed, e.g., from zipf 0.2 to zipf 0.8. This happens because the W-wav method combines weights with wavelets. The more skewed the weights, the higher the probability for wavelets with heavy weights to be used. On the other hands, both the 2-step and M-step algorithms can ignore those exaggerated wavelets by setting their coefficients to 0.

Real Data



(a)Normal[10, 100] data



(b)Uniform[1, 10] data

Figure 2.10: Skewness

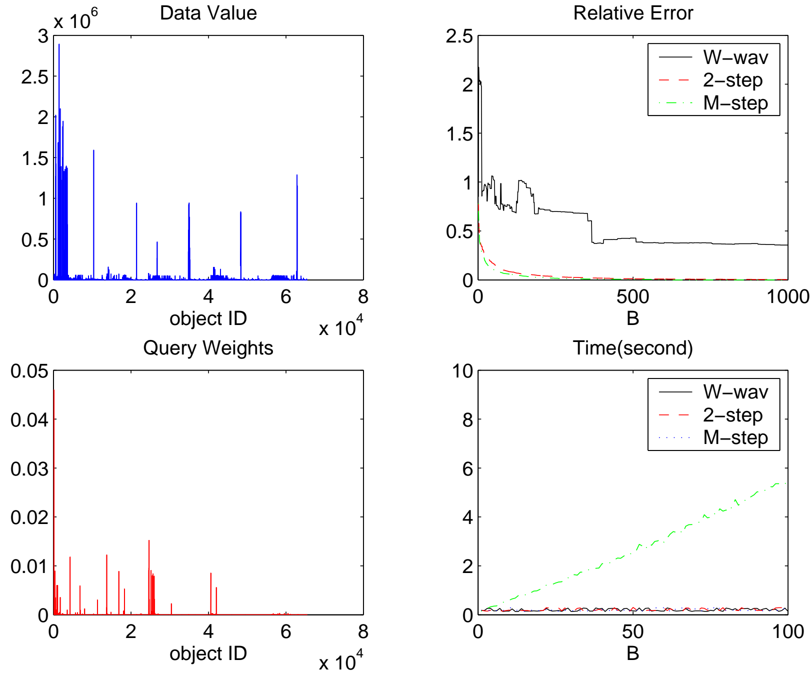


Figure 2.11: Relative error compare for world cup data

In order to validate the above performance metrics on more realistic data, a data set collected from the 66th day of the World Cup 1998 [53] is used, because it has a large number of queries. In this experiment, the data represents the query subject, which may be a webpage or a picture, while the data value represents the max response size of the subject. The weights correspond to the number of queries for each data point after normalization. In this case as well, the same dynamics as before have been observed for the relative error: the W-wav method provides a relative error above $2\epsilon_0$ at first, then it is slowly reduced to $0.35\epsilon_0$ at $B = 600$ and remains there through $B = 1000$ (Figure 2.11). The 2-step algorithm reduces its error to $0.35\epsilon_0$ with $B = 13$ while the M-step algorithm reduces its error to $0.35\epsilon_0$ with only $B = 9$. The difference in running time between the 2-step and the W-wav methods is very small when $B < 100$.

2.7.2 Range-Sum Approximation

In this section, the proposed algorithms, i.e., the data-mapping method and the weight-mapping method are compared with other methods, including the *Naive* method described in Section 2.4.

- **Data** Our data-mapping method,
  $\Pi^{(DM)}[i] = \sum_{k=1}^{i} \sqrt{\Pi[k,i]} + \sum_{k=i+1}^{n} \sqrt{\Pi[i+1,k]}$
- **Data2** A simple straight forward data-mapping method,
  $\Pi^{(DM)}[k] = \sum_{1 \leq i \leq j \leq k} \sqrt{\Pi[i,j]}$
- **Weight** Our weight-mapping method, $\Pi^{(WM)}[k] = \sum_{i \in [1,k], j \in [k,n]}$
- **Weight2** A simple subtractive weight-mapping method, $\Pi^{(WM)} = \Pi^{(DM)}[i] - \Pi^{(DM)}[i-1]$, we use $\Pi^{(DM)}$ in **Data**
- **Naive** Naive method with new signal $X = \{A(0,0), A(0,1), ..., A(n-1, n-1)\}$.

The *Naive* method produces relative errors in the range 40-90 while *Data* pushes it down below 1 (Figure 2.12). This is due to the fact that B is too small for a data set with $O(n^2)$ length. Figure 2.13 shows the comparison of the *Data* and *Data2* methods in terms of their accuracy. It should be noticed how *Data* performs better than *Data2* over different data sets, because the weights computed by *Data2* are not as accurate as the weights computed by *Data* that are derived directly from the error functions.

For exponential and uniform data, *Weight* is much better than *Weight2*, with an error of almost 0. In normal data, however, it is worse (Figure 2.13). The reason is that *Weight2* focuses only on intervals start with i, $i+1$, $i-1$ or ends with i, $i-1$, $i+1$, and ignores all other intervals $[i,j]$ covering it. For the exponential and uniform data sets, some intervals may contain very different values from others, and ignoring these intervals incurs large errors. *Weight* considers all intervals that cover the data point, but it exaggerates the middle part of the signal. For normal data, all intervals are similar, exaggerating certain intervals makes them unfairly important than others, which causes error.

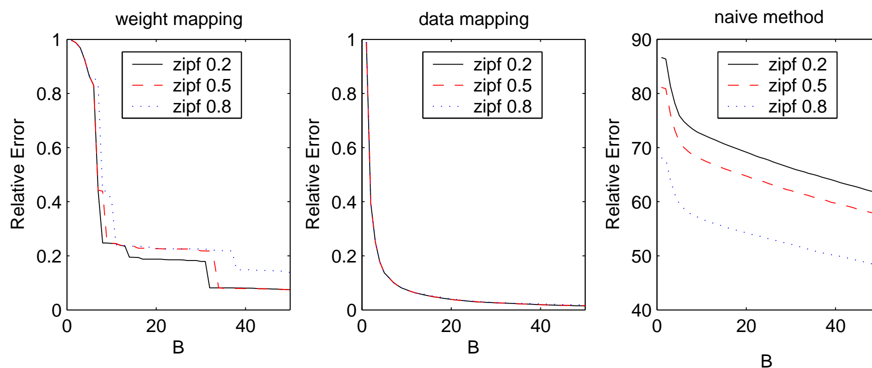


Figure 2.12: Relative error for normal(10, 100) data

Figure 2.14 shows the relative error of *Data* and *Weight* methods as a function of skewness of weights (zipf 0.2, 0.5 and 0.8). It can be observed that the skewness of the queries does not affect the algorithm accuracy because both *Data* and *Weight* methods sum up certain $\Pi[i, j]$s to compute a new weight, and the summation cancels out the skewness effect.

2.8 Summary

The nonuniform approximations for both point-wise and range-sum queries have been studied here. Although the approximation is in one dimension, it can be easily generalized to multi-dimensions, because the methods used to choose wavelets and coefficients are not restricted by dimensionality. The wavelets are selected from the optimal solution for weighted data. However they are not optimal with respect to the original data. To overcome this problem, a second step is performed to optimize their coefficients for the original data, which takes extra $O(B^3)$ time, but improves accuracy significantly. How to find the optimal wavelets for original data is still an open problem. Incremental algorithms have been designed here to reduce synopses-accuracy-keeping cost for dynamic data and weights. The algorithms exploit the values computed in history to lower the cost of computing new values when data and weights change, thus successfully reduce the complexity from linear to sublinear.

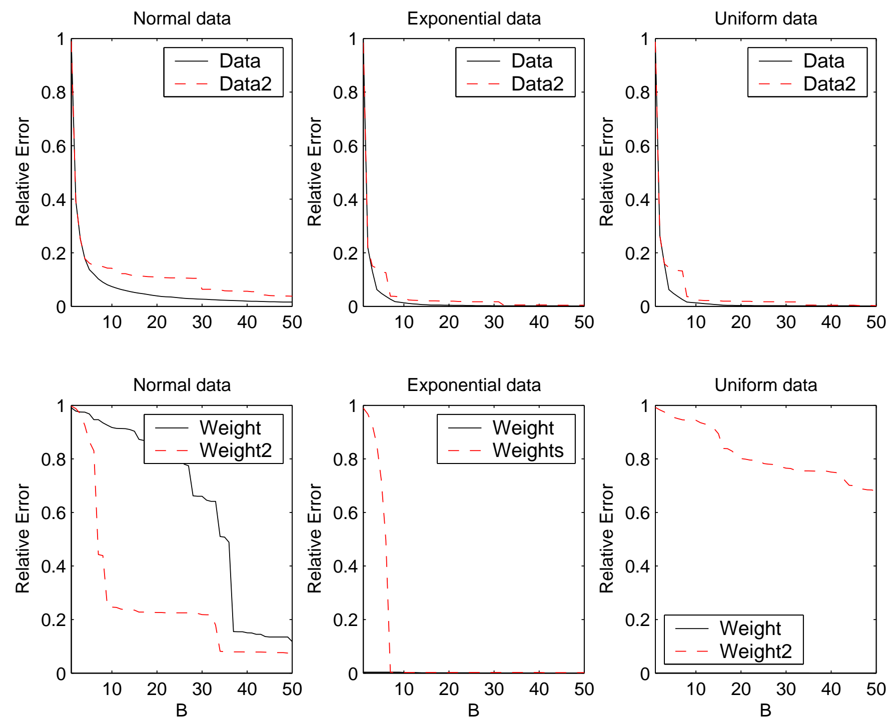


Figure 2.13: Comparison between data-mapping and weight-mapping algorithms($\alpha = 0.5$)

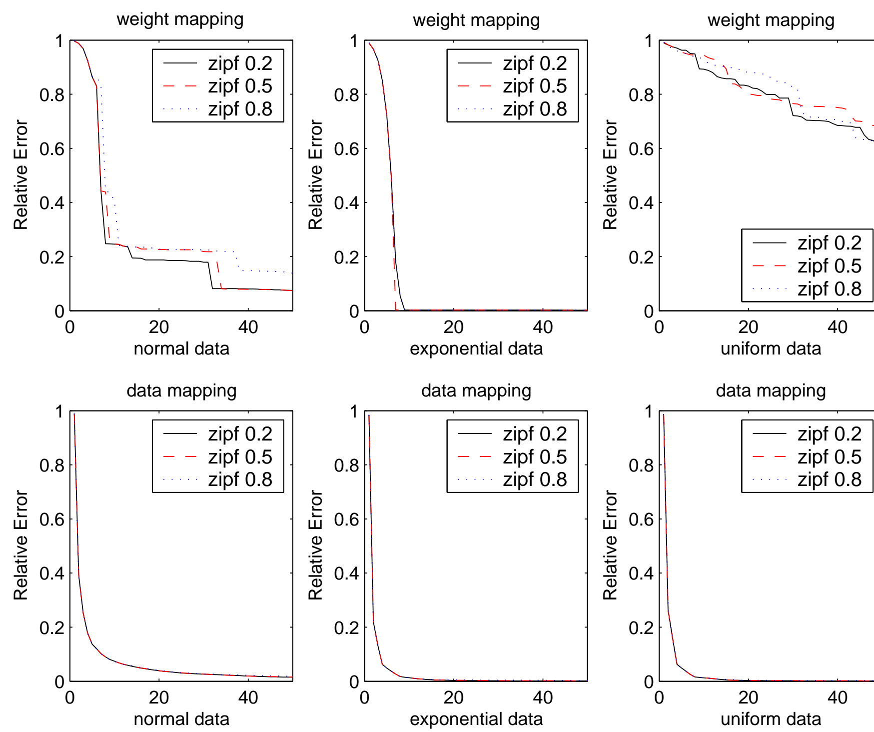


Figure 2.14: Query skewness

Chapter 3

Stream-Aware Lossless Compression in IP Networks

In Chapter 2, we have discussed the algorithms that can keep the accuracy of compressed data with low costs in the first scenario in dynamic systems where data are varying over time. In this chapter, we study how to compress data in the second scenario where the data patterns are changing over time. The traffic compression in IP networks is used to illustrate this problem. Besides dynamic data patterns, traffic compression in IP networks itself is an interesting problem due to the following three reasons. First, unlike other compressions, online traffic compressors have to work at very high speed to accommodate the data-arriving rate in core networks. Second, online and offline traffic compressions have very different resource constrains that require different compression strategies to maximize their performance. Third, for different applications, the value of information contained in different parts of the data may not be equal. For example, in traffic classification, the most valuable part of the data is the one that contains *Application Signature*, which is a sequence of strings that indicates which layer-7 application this packet belongs to. In this chapter, we discuss all these challenges in IP network compression and how to design both online and offline compression algorithms that can work under these constrains. This chapter starts with an introduction of IP network and the motivation of traffic compression.

3.1 Background

The Internet Protocol (IP) is the choice of transport protocol both on wired and wireless networks, and this choice is leading to the convergence of telecommunication and data networks. These converged networks will be the building blocks for the All-IP vision. As the networks evolve to provide more bandwidth and applications, network operators

must be equipped themselves with new tools in order to better understand the behavior of the network and its users, and thus attract more customers. In order to be ready and competitive in the market place, large ISPs have been deploying distributed monitoring (or sensing) infrastructures that collect IP data-streams from a variety of geographical monitoring stations, e.g., either using passive probes or exported from routers via Netflow, and deliver such information to a centralized processing station for further analysis. In spite of its importance, network operators struggle with the challenging problem of defining the right trade-off between the granularity of data required to be collected and the volume of data that can be really exported to the processing station, since it competes with user traffic and thus potentially degrades its performance. This problem becomes even more critical for wireless operators for which bandwidth is the most precious resource.

Unfortunately, different operational practice requires different traffic granularity. For example, flow-level information results to be very useful for network management, traffic engineering and for some degrees, traffic visibility and security[1]. Despite the increased resolution into the source and destination of traffic demands, and their nature (as exposed by the port numbers), no information is preserved on the timing of individual packets in a flow or their individual content, which is key for applications like content-billing, layer-7 security or lawful intercept (mandatory in North America since May 15th, 2007). The only way one can gain visibility into the explicit timing of each packet and its content is through the collection of information on a per packet basis[2].

Sampling has been widely accepted as a de-facto solution to solve this problem when part of full packet content information is not required. Unfortunately, for highly content demanding applications, sampling is not a viable solution and thus the problem still remains open.

---

[1]Flow-level information can be collected either using passive probes tapping specific links or mirroring router ports, or directly exported from router interfaces supporting Netflow V5 to V8

[2]Packet-level information might be collected either via passive tapping specific links or mirroring router ports, or directly exported from router interfaces supporting the latest version of Netflow (Netflow V9).

At the same time, carriers have manifested clearly a strong desire not just to collect and analyze traffic, but also to store the exported information for analyzing trends of application usage and user behavior over time. This information comes handy for the purpose of understanding the popularity of a specific IP application or service over time, or trends of security vulnerabilities being exploited, etc. More recently, what has been just a nice-to-have functionality has become a strict requirement as carriers have been asked by government agencies to store specific data for years in their facilities before discarding them. An example of such requirement is data retention, which requires to store layer-4 information and key packet payload information for all carrier's customers. Notice that all the above translate into huge storage requirements for carriers, just think that a one hour collection of TCP/IP header on a 10Gb/s link can easily reach 3 Terabytes of storage.

Several compression algorithms have been published in the literature to handle this problem. However, as shown in the remainder of this section, none of them are specialized to carry out an efficient compression when dealing with pure IP traffic.

In this chapter both problems are tackled by proposing a novel *lossless* compression algorithm, called *IPzip* that is applicable for both issues: (i) *online case*, whose goal is to reduce the network bandwidth required to export the IP data from monitoring stations to the centralized processing station or storage facility; (ii) *offline case*, whose goal is to reduce the overall storage space. Although both the cases require dealing with how to compress the data-streams, each of them has to be handled with slightly different design criteria. For example, in the online case, the compressor must guarantee *short compression time and small memory usage* due to the limited buffer sizes and the high data rate of packet arrivals. On the other hands, in the offline case, the importance of how fast the compressor can compress diminishes in respect of the importance of achieving a *good compression ratio*. The new contributions of this thesis are summarized as follows.

. The first *lossless* data compression algorithm, called *IPzip* is proposed here for IP backbone network traffic. The main contribution of IPzip is its deep analysis

and discovery of inner properties of IP data-streams that, if fully exploited, can be used by existing Lempel-Ziv based compressors, like Gzip, to severely reduce the size of the original data sets. For example, for some Netflow style data, the compressed size can be as small as 1% of original data size (see Section 3.6).

- IPzip exploits the hidden *intra-packet correlation* and *inter-packet correlation* properties of the data-streams, and produces an efficient *compression plan* that reorganizes the data-streams both within and across packets to improve compression ratio. Since the generation of the optimal compression plan is NP-Complete, we propose novel heuristics that produce near-optimal plans.

- Since IPzip produces a near-optimal compression plan by exploiting the structure of the network data-streams, it is key to monitor the effectiveness of the compression plan over time as Internet traffic is highly dynamic. We propose a mechanism that monitors the performance of the compression plan being currently used, and efficiently switches to a new compression plan when required.

- Two versions of IPzip are proposed here to be used for online and offline data-streams compression. By using packet traces collected from several Tier-1 carriers world-wide, we demonstrate how IPzip can achieve a superior compression ratio of up to 20% in the offline case and up to 15% in the online case when compared to Gzip.

This chapter is organized as follows: Section 3.2 discusses related works in the literature, and Section 3.3 gives the intuitions behind the IPzip compression. The new IPzip algorithm is introduced in Section 3.4, and a simple algorithm is proposed in Section 3.5.3 to solve the problems when traffic pattern changes. The effectiveness of IPzip is demonstrated through experiments on real backbone traces in Section 3.6, and Section 3.7 concludes this chapter.

3.2 Related Work

A lot of work has been done in compression by researchers for a variety of different data structures and contexts. While the networking community has approached the problem of compressing IP datastreams more as packets come-by, i.e., the online scenario, the database community, has focused more on applying such techniques to save storage space, i.e., the offline scenario. Common to both approaches is that previous researches focus only on packet/flow headers and only minimal attention has been paid to the compression of less regular data-streams as payloads.

Network researchers have proposed simple entropy-based compression algorithms that achieve high compression speed at the cost of inferior compression ratio. Most of these methods, like TCP/IP header compression [44, 45], are based on the idea of reducing the transmission bandwidth or latency for low-speed serial links by replacing the packet headers with the connection index to which the packets belong to. On a similar path, some researchers [24, 26, 32, 43] proposed to replace the 5-tuple flow id with a shorter code to compress TCP/IP headers. Authors [33] derived a theoretical bound for all these entropy-based coding, assuming that some fields, such as flow ids and their inter-arrival time, are completely independent of each other. However, all the above coding methods fail to explore the correlations within the packet headers, and thus the information bound proposed ended up not as tight as expected, i.e., the compression ratio can be further improved. Robust Header Compression (ROHC) [46] discovers dependencies in packet headers within one flow. However, the correlations between flows are ignored.

Database researchers approached the problem of compressing large tables using a more theoretical method by investigating the structure of tables with the major goal of reducing storage space. Although these algorithms are able to achieve tremendous data reduction, they consume too much system resources, thus are unable to perform "on-the-fly' compression'. Among those, Pzip [12, 13] tries to find the best partition of the columns on either original column sequence or reordered columns, then compresses each partition separately. However, such a solution considers only the pairwise relation

between the columns, which may not be accurate for the total ordering of columns in the partition, and fails to explore the relationship between rows. In their implementation, they compress one column after another, leading to the need of large buffers to collect enough data to achieve a good compression ratio. Spartan divides the table into predictive and predicted columns [2, 22], and compresses the predictive columns only. This method is only suitable for lossy compression or tables with strong columns correlations, because the columns that can be predicted in traffic headers are very limited in a lossless scenario. Similar to Spartan, authors in [50] try to classify the correlated columns into predictive and partially predicted columns, then sort the partially predicted columns by the predictive columns in a similar way to Burrows-Wheeler transform [4]. After sorting, similar data are gathered together in the predicted columns, so that the compression ratio can be improved. Fascicles and ItCompress [27, 28] scan all rows to learn the correlations inside the table. However, both sorting and row-scanning require to buffer large volume of data to achieve good compression ratio, which is unaffordable in real online compression applications.

Regarding the compression of packet payload, not much work has been published in the literature. When adapting general compressors to such a problem, the performances achieved end up to be either poor from a compression ratio's perspective, like Gzip, or impractical due to the long compression time required, like Bzip[3]. Only recently new algorithms to compress packet payloads have been invented by the networking community. For example, IPComp [47], Stacker, Predictor [54] and TCP Compression Filter [55]. Their common characteristic is their lack of learning the intrinsic structure of packet payloads that, if fully exploited as shown in this chapter, can lead to better results.

3.3 Intuitions behind IPzip

In this section, we first provide a brief overview of one of the most known data compression algorithm, e.g. Gzip (Section 3.3.1). By explaining how it works with a simple

---

[3]http://en.wikipedia.org/wiki/Bzip2

Output	History	Look ahead	Remaining data
	8 7 6 5 4 3 2 1		
		a x a x a x a x	b y b y b y b y c z c z c z c z
a	a	x a x a x a x b	y b y b y b y c z c z c z c z
x	a x	a x a x a x b y	b y b y b y c z c z c z c z
(2,6)	a x a x a x a x	b y b y b y b y	c z c z c z c z
b	x a x a x a x b	y b y b y b y c	z c z c z c z
y	a x a x a x b y	b y b y b y c z	c z c z c z
(2,6)	b y b y b y b y	c z c z c z c z	
c	y b y b y b y c	z c z c z c z	
z	b y b y b y c z	c z c z c z	
(2,6)	c z c z c z c z		

Input data : a x a x a x a x b y b y b y b y c z c z c z c z
Compressed data: a x (2,6) b y (2,6) c z (2,6)

Figure 3.1: Gzip compression example

example, we highlight its major limitations and thus identify the key aspects on how it can be improved. These constitute the intuitions behind IPzip (Section 3.3.2).

3.3.1 Gzip Background

Figure 3.1 shows an example of how Gzip compresses strings. Suppose, we set the Gzip parameters of history window size and look-ahead window size to be 8. Gzip starts by reading 8 characters, then checks whether the string starting with current pointer appears in the history window. For example, at step 1, $axaxaxax$ is read and the current pointer points to a. Since the history window is empty, the output is a. At step 2, a is moved to the history window, after that x is read and output, since it does not exist in the history window. Then at the next step, a appears again, which can be found in the history window. Further, on checking the string following it, Gzip discovers that the new string $axaxax$ repeats the history string starting at position 2. Since the length of the string is 6, the output is $(2, 6)$. The above process is repeated to get the compressed string $ax(2, 6)by(2, 6)cz(2, 6)$.

sp - source port, dp – destination port, ts - timestamp

Original data				Reorder columns				Reorder rows				Group 1		Group 2
sp	ts	dp		sp	dp	ts		sp	dp	ts		sp	dp	ts
a	1	x		a	x	1		a	x	1		a	x	1
b	2	y		b	y	2		a	x	5		a	x	5
c	3	z		c	z	3		a	x	8		a	x	8
b	4	y		b	y	4		a	x	9		a	x	9
a	5	x		a	x	5		b	y	2		b	y	2
c	6	z	⇨	c	z	6	⇨	b	y	4	⇨	b	y	4
c	7	z		c	z	7		b	y	11		b	y	11
a	8	x		a	x	8		b	y	12		b	y	12
a	9	x		a	x	9		c	z	3		c	z	3
c	10	z		c	z	10		c	z	6		c	z	6
b	11	y		b	y	11		c	z	7		c	z	7
b	12	y		b	y	12		c	z	10		c	z	10

Figure 3.2: Header compression example

3.3.2 Network Traffic Correlations

Next, the intuition behind IPzip is described via an example. Suppose, there is a data set consisting of the following 3 fields extracted from packet headers in the order of source port, timestamp and destination port (see Figure 3.2). Suppose that source and destination ports are correlated with each other, e.g., source ports a, b and c are always associated with destination ports x, y and z, respectively.

Gzip with No Reordering

First, notice that due to the order of arrival of packets in Figure 3.2, this data set is hard to compress. If it is compressed row-wise using Gzip with a window sizes of 8, the compressed string is exactly the same as the original one. Even if it is compressed column-wise, it is only possible to obtain a compressed string as "a b c b a c c a(4,2)b b 1 2 3 4 5 6 7 8 9 10 11 12 x y z y x z z x(4,2)y y". Assuming that our alphabet has only these 18 characters, "a-c,x-z,1-12", 5 bits are necessary to code each character. Given that the window size is 8, 3 bits are needed to represent the position and length of the string. Hence, the total length of two characters, e.g., "ac" is reduced from 10 to 6 by represented as (position, length), which results in a total saving of 8 bits.

Intra-Packet Correlation

However, given the strong correlation within the packet such as that exhibited by

the source and destination ports, it makes sense to group them together to achieve a much better compression ratio. Thus, if the columns are reordered as "source port, destination port and timestamp", it can be compressed into "a x 1 b y 2 c z 3(6,2)4 a x 5 c z 6(3,2)7 a x 8(3,2)9 c z 10 b y 11(3,2)12" in row-major, and "a b c b a c c a(4,2)b b x y z y x z z x(4,2)y y 1 2 3 4 5 6 7 8 9 10 11 12" in column-major.

Inter-Packet Correlation

Further, given the correlations exhibited across packets such as those having the same values for the "source port, destination port" pair, an even better compression ratio can be achieved. For instance, if the columns are partitioned in to two groups: "source port, destination port" and "timestamp" and then reordered the rows with the "source port, destination port" as the sort key, then a highly compressible string for the source and destination ports is obtained. If it is compressed row-wise, "a x (2,6) b y (2,6) c z (2,6)" (see Figure 3.1) is obtained; and if column-wise, "a(1,3) b(1,3) c(1,3) x(1,3) y(1,3) z(1,3)" is obtained. The compressed timestamp string is the same as itself. As a result, we save 36 bits in row-major, and 24 bits in column-major, which means *an overall gain of 3 to 4 times compared to Gzip*. To recover the original data, it is necessary to perform decompression first, then sort them by the timestamp.

Similar correlations exist within and across packets in network traffic data, which, if adequately exploited, can yield much better compression ratios. Thus, the problem has been reduced to first finding these correlations, and then rearranging the data accordingly to achieve high compression ratios, e.g., *compression plan*. Section 3.4 provides the details of how IPzip generates such a plan.

3.4 Algorithm

The hypothesis for IPzip is that network traffic exhibits correlations within packets as well as across packets, which can be exploited to achieve a better compression ratio. IPzip develops a compression plan that exploits these correlations, and this plan is then given to Gzip compressors to finally compress the traffic stream. In this section, optimal algorithms for reordering bytes within a packet and packets themselves are presented

first. Since, solutions to these optimal algorithms are super-exponential, approximate solutions used by IPzip are introduced to find a good trade-off between the complexity and the associated performance.

3.4.1 Compression Plan Generation

Network data can be viewed as a tuple with both structured data and unstructured data. For example, network packet data is a tuple <header, payload>, with header as the structured part, and payload as the unstructured part. The structured data denoted by T can be viewed as a table with header fields defining the columns and packets defining the rows. The unstructured data denoted by S can be viewed as a list of strings with each string record representing a variable-sized payload. In some applications, when only structured part is collected, it can be viewed as a special case with length of its unstructured part equal to 0. For a network data set that can be represented as tuple $< T, S >$, the notation $< T[i], S[i] >$ is used to represent the ith packet in the data set. The compressions of the structured part T is separated from that of the unstructured part S, because of the structure and content dissimilarities between them. The compression problems can be defined as follows.

Problem 5 (Compression Based on Intra-Packet Correlation) *Let T be the structured data need to be compressed. T has n columns. For a given compressor C, let $C(T)$ be the compressed bytes of T. The goal is to divide the columns into $G_1, ..., G_{\hat{K}}$ groups. Each group G_i contains k_i columns, $T = \cup_{i=1}^{\hat{K}} G_i$, and $\sum_{i=1}^{\hat{K}} k_i = n$, so that the compressed size $C(T) = \sum_{i=1}^{\hat{K}} C(G_i)$ is minimized.*

Thus, Problem 5 is reduced to finding the best column-wise partition of the data. At this time, it is imperative to point out that each byte is defined as a column, e.g., a 4-byte IP-address consists of 4 columns. The reason a column is used to represent a byte is that (i) it is a good tradeoff between fine correlation granularity and complexity, i.e., if each column is a packet header field, the ability to discover the correlations between bytes inside a field is lost; on the other hand, if each column is a bit, there are too many columns to explore. (ii) when IPzip is used as general table compressors, it does not

require to know the table semantics (if each column is a field, for each table it compress, IPzip has to know every field in the table).

Because there is more similarity inside one group than between groups, the compressor processes each group independently to improve the compression ratio. Thus, there are multiple best answers such that if $\{G_1, ..., G_K\}$ is the best grouping, then any permutation of $\{G_1, ..., G_K\}$ is a best grouping, too. For the purposes of compression only, it is not necessary to find all of them. The algorithm in Section 3.4.2 returns one of the optimal solutions. It is not difficult to perform a minor modification to it so that it returns all solutions.

Problem 6 (Compression Based on Inter-Packet Correlation) *Let* $S = \{S[1], S[2], ..., S[m]\}$ *be the unstructured data where* m *is the number of records or packet payloads. For a given compressor* C*, let* $C(S)$ *be the compressed bytes of* S*. The goal is to divide* S *into* $G_1, ..., G_{\bar{K}}$ *groups. Each group* G_i *contains* k_i *of records,* $S = \cup_{i=1}^{\bar{K}} G_i$*, and* $\sum_{i=1}^{\bar{K}} k_i = m$*, so that the compressed size* $C(S) = \sum_{i=1}^{\bar{K}} C(G_i)$ *is minimized.*

Thus, Problem 6 is reduced to finding the best reordering of all rows, i.e., payloads such that when similar rows are compressed together, a better compression is achieved. It is not difficult to see that an algorithm that solves Problem 6 can also find the best reordering of rows that minimizes the compression ratio. The following lemma describes the complexity of finding the optimal answers.

Lemma 7 *The best grouping can be found in* $O(n^2 n!)$ *for Problem 5, and* $O(m^2 m!)$ *for Problem 6.*

Proof For Problem 5, the permutations of all columns is computed with complexity $O(n!)$, and then dynamic programming is used to find the optimal partition of the columns for each permutation with complexity $O(n^2)$. So, the total cost is $O(n^2 n!)$. Similarly, Problem 6 can be solved by permutating all rows and applying dynamic programming to find best grouping of all rows. So the total cost is $O(m^2 m!)$. The details are presented in the optimal algorithm in Section 3.4.2. □

1: $\mathscr{T} \leftarrow \{\}$
2: Generate all possible ordering of columns for table T, add them to $\mathscr{T}$
3: Best$[T, n] \leftarrow +\infty$
4: for every $T_i \in \mathscr{T}$
5: for $(j = 1..n)$
6: Best$[T, n] \leftarrow$ min(Best$[T, n]$, Best$[T_i^l, j]$ + Best$[T_i^r, n - j]$)
7: return Best$[T, n]$

Figure 3.3: Optimal algorithm

3.4.2 Optimal Algorithm

The method for finding the optimal solution for Problem 5 and 6 are very similar. Here Problem 5 is used as an example. The exhaustive search for the optimal answer contains finding both the best ordering of the columns and the best partition. Let $\mathscr{T} = \{T_i\}$ be a set of all possible ordering of columns in table T, i.e., columns in T_i is a permutation of columns in T. Let Best$[T, n]$ be the smallest compressed size of table T with n columns, and let T_i^l be the subset of the left j columns in table T_i, and T_i^r be the subset of the right $n - j$ columns. The algorithm in Figure 3.3 describes how to find the optimal solution.

In this algorithm, there are $n!$ possible permutations of all columns in table T, so the loop in line 4 will be executed $n!$ times. With dynamic programming, the smallest compressed size for every interval will be computed only once, and there are only $O(n^2)$ intervals for each T_i, so the complexity of finding its best grouping for T_i is $O(n^2)$. As a result, the complexity of this algorithm is $O(n^2 n!)$.

However, the cost for the optimal algorithm is too high for practical use and hence IPzip's near-optimal algorithm is introduced here, which learns the correlation pattern through a small training set, generates a compression plan, and then compresses the original data set according to this plan. This separation moves all the complexity to the training step, which can be taken offline, so that the real online compression can still be done very fast. This holds true under the assumption that the training set represents the original data set, so that the near-optimal plan generated from training set will still be a good plan for the original data set. Section 3.5.3 discusses the algorithm that

tracks the performance of IPzip over time and switch to a more efficient compression plan when required.

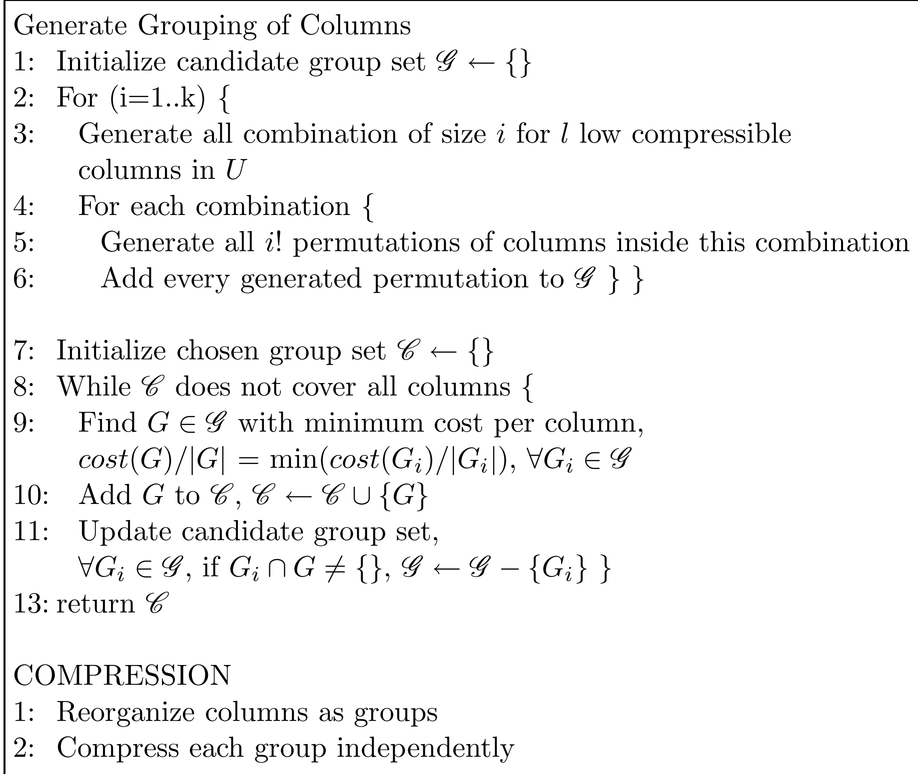
3.4.3 IPzip Compression Plan Generation Algorithms

In this section, we present the IPzip compression plan generation algorithms, which exploit the intra- and inter-packet correlations inherent in network traffic.

IPzip Compression Plan Generation for Intra-packet Correlation

IPzip's plan generation algorithm for structured data exploits the fact that packet headers contain certain highly compressible columns. For instance, the IP Version field in the IP header is always 4 in real traffic, and hence highly compressible. For such columns, differential encoding is applied, which computes the difference of current data from previous data, to transfer them into a 0-dominated sequence and finally compress them separately. After these high-compressible columns are removed, the computation cost of finding a good compression plan for the remaining columns is reduced significantly. In the following an algorithm to handle the low-compressible columns is presented.

The algorithm works as follows. Let U represent the set of low-compressible columns, whose cardinality is denoted by l. Let k denote the maximum group size, i.e., maximum number of columns allowed in a group. The algorithm generates all possible candidate groups, denoted as $\mathscr{G} = \{G_1, ..., G_{\hat{N}}\}$, where the number of columns in each $G_i \in \mathscr{G}$ denoted as $|G_i|$ is between 1 and k. The algorithm then computes the compressed size for each of them. Let $|G_i|$ represent the size of the generic group $G_i \in \mathscr{G}$, i.e., the number of columns in G_i. Let the cost for each group G_i be its compressed size, denoted as $cost(G_i)$. So, the problem is reduced to finding the best set of groups that covers all l columns with minimum cost, denoted as $\mathscr{C}$. This is a well known NP-complete problem, called *minimum set cover*. Chvatal's greedy algorithm [10] is used in this thesis to get an approximate answer. If OPT is the cost of the optimal coverage and H_l is the l-th harmonic number, $H_l = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{l} \approx \ln l$, then the cost of coverage found by the greedy algorithm is no worse than $H(l)OPT$.



Generate Grouping of Columns
1: Initialize candidate group set $\mathscr{G} \leftarrow \{\}$
2: For (i=1..k) {
3: Generate all combination of size i for l low compressible columns in U
4: For each combination {
5: Generate all $i!$ permutations of columns inside this combination
6: Add every generated permutation to $\mathscr{G}$ } }

7: Initialize chosen group set $\mathscr{C} \leftarrow \{\}$
8: While $\mathscr{C}$ does not cover all columns {
9: Find $G \in \mathscr{G}$ with minimum cost per column, $cost(G)/|G| = \min(cost(G_i)/|G_i|), \forall G_i \in \mathscr{G}$
10: Add G to $\mathscr{C}$, $\mathscr{C} \leftarrow \mathscr{C} \cup \{G\}$
11: Update candidate group set, $\forall G_i \in \mathscr{G}$, if $G_i \cap G \neq \{\}$, $\mathscr{G} \leftarrow \mathscr{G} - \{G_i\}$ }
13: return $\mathscr{C}$

COMPRESSION
1: Reorganize columns as groups
2: Compress each group independently

Figure 3.4: Compression plan generation for intra-packet correlation

Having identified the near-optimal group of columns, the algorithm enters the compression step. In this step, the low-compressible columns in the original data are rearranged into groups found by the plan generation step, i.e., $\mathscr{C}$, and then the given compressor is applied to compress each group independently. The details of the algorithm are shown in Figure 3.4.

Note that to avoid compressing the same column several times in different groups, we remove all groups from the candidate set that overlap with the chosen group, i.e., the groups that contain some columns covered by the chosen group (see line 11). This does not change the algorithm, since all groups from size 1 to k are generated.

This algorithm is not optimal, even if $k = n$, since (i) the greedy algorithm only finds an approximation of optimal solution and (ii) the best grouping for the training set may not be the best one for original set. But in practice, this sub-optimal grouping with $k < n$, can be very efficient, because even in data sets with large number of columns, the number of correlated columns is limited. For example, the port number may be

correlated with layer 7 application, but not layer 2 protocol. Only in rare cases, we have to explore all n columns to find the correlated ones.

The complexity of this algorithm is bounded by $\sum_{i=1}^{k} i!\mathrm{C}_l^i = \sum_i \mathrm{P}_l^i = O(l^k)$, because there are in total $\sum_{i=1}^{k} i!\mathrm{C}_l^i$ number of candidate groups generated. The complexity of finding the minimum set coverage is $O(l^k)$, too, because at each step, there must be at least one column added to C, so the loop at line 8 runs at most l times. With one column covered, the number of candidate groups will be reduced from $O(l^k)$ to $O((l-1)^k)$, and with j columns covered, the candidate group size is only $O((l-j)^k)$. So, the total number of times the candidate groups are visited in order to find the minimum cost coverage is $O(l^k) + O((l-1)^k) + ... + O(1^k) = O(l^k)$.

As can be seen from above, if l is large, the plan generation complexity is large. Hence, in IPzip, this step is done offline and the learnt plan can then be applied against the actual traffic data in real time.

Compression Plan Generation for Inter-Packet Correlation

Recall that the optimal solution would require reordering all the packets and has a super-exponential complexity in terms of the number of payloads ($O(m^2 m!)$). Furthermore, it only returns an optimal ordering of rows, thus an optimal solution from the training data cannot be used on the whole data set.

To address this issue, we introduce a near-optimal algorithm that returns a set of rules that describe how to reorder payloads instead of actual ordering. This algorithm is based on the observation that a packet's payload is typically correlated with its header. For instance, the port number in the packet header indicates the application this packet payload belongs to, e.g., port 80 in the header can be expected to be correlated with the appearance of the string "http" in the payload. Moreover, IPzip exploits the behavior of compressors such as Gzip that are based on the lempel-ziv algorithm, which achieves a good compression ratio when the neighboring bits in the input data stream are highly correlated. Thus, the packet payloads that are correlated to each other such as those that correspond to the same destination port, should all be sent to the same compressor.

The compression plan generation algorithm classifies all the payloads in the training data set into multiple groups, where each group is then compressed via a separate Gzip

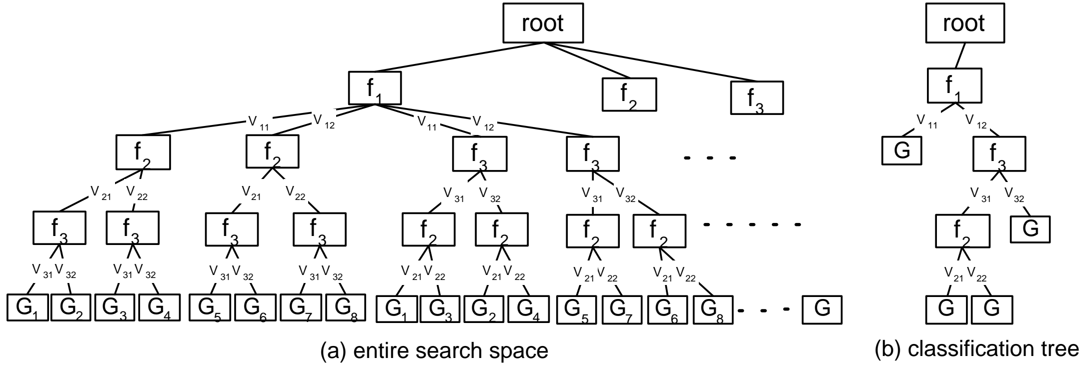


Figure 3.5: Payload compression algorithm

compressor. Information in headers (T) is used in this thesis to generate the best compression plan for the payloads (S). In practice, some fields in T contain little or no information for classification, e.g., the timestamp field in network flow data. Thus, excluding these fields from training can reduce the training time and the classification tree size. F is defined here as the fields in T that are related to S, $F = f_1, f_2, ..., f_n$.

A simple solution to classify the payloads would be to construct a full classification tree, where the values of every field are enumerated. Figure 3.5 (a) shows an example of full classification tree with each node representing a group of payloads. In this example, F comprises of 3 fields (f_1, f_2, f_3), and each field f_i has 2 values v_{i1}, v_{i2}. A record $S[i]$ will be assigned to a group, i.e., the leaf node in the tree, based on the field values in its associated $F[i]$. It is not difficult to find out that several paths lead to the same group (Figure 3.5 (a)). For example, G_2 can be reached from the path $f_1 \stackrel{v_{11}}{\rightarrow} f_2 \stackrel{v_{21}}{\rightarrow} f_3 \stackrel{v_{32}}{\rightarrow} G_2$ or $f_1 \stackrel{v_{11}}{\rightarrow} f_3 \stackrel{v_{32}}{\rightarrow} f_2 \stackrel{v_{21}}{\rightarrow} G_2$. A brute force classification can take all fields one by one, and enumerate all values for each field. The subtree under $root \rightarrow f_1$ in Figure 3.5 (a) is such an example, i.e., the subtree with leaves $\{G_1, G_2, G_3, G_4, G_5, G_6, G_7, G_8\}$ (the left subtree under f_1) and the subtree with leaves $\{G_1, G_3, G_2, G_4, G_5, G_7, G_6, G_8\}$ (the right subtree under f_1) produce the same classification, since their leaves are the same, just in different order. However, compression based on this full classification tree may not be the best solution. First, enumeration of all values for all fields is too expensive. For example, there are 2^{32} possible IP addresses. Second, classifying via all the fields

may not achieve the best compression. To the contrary, for some groups that have very few records, it may be better to combine them with their sibling or parent groups to achieve a better compression ratio. This is because Lempel-Ziv based compressors need a large amount of data to achieve a good compression ratio. If a group does not contain enough data, the algorithm either waits for more data, or compromises the compression ratio.

IPzip proposes a greedy solution to build a classification tree, which may not necessarily be a complete tree. Let the tree *node* represent the group of payloads that have been classified to it by the fields and their values from *root* to itself, where *root* represent the entire data set. The function $cost(node)$ is used to represent the compressed size of the *node*, and $Path(node)$ is used to represent the set of fields along the path from *root* to the *node*, then $F - Path(node)$ is the set of fields not yet used in the classification of the *node*. This algorithm starts to find the best classification field that minimizes the cost of *root* by trying all used fields, then classifying the *root* into subgroups/subnodes according to this best field. Then the above procedure is repeated for each subnode until the cost cannot be minimized anymore. The algorithm for constructing the classification tree is shown in Figure 3.6, in which, Q is the queue of tree nodes need to explore. An example of such a tree is shown in Figure 3.5 (b). In the next step, the incoming data are compressed according the classification tree

As discussed above, if a full classification is used, the order of fields along the path is not important, since they generate the same leaf nodes, just in different order. However, in IPzip's classification tree generation, the tree is trimmed and hence the order of fields makes a difference. Still, IPzip ensures that we create the best path to reach a leaf node, as described in the following lemma.

Lemma 8 *IPzip's classification tree generation algorithm achieves the best order of fields used in classification.*

Proof At step 1, the only leaf node is the root. The algorithm chooses the field f_{i_1} that leads to the smallest compression size to classify the whole data set as described in the algorithm.

```
Build the Classification Tree
1:  Q ← root
2:  while Q is not empty {
3:    node ← first node in Q; minlen ← cost(node)
5:    f_chosen ← NULL
6:    for every f ∈ F − Path(node) {
7:      further classify node to node_1, node_2, ... according to f's value
8:      newlen ← Σ_i cost(node_i)
9:      if minlen > newlen
10:       f_chosen ← f; minlen ← newlen }
11:   if f_chosen = NULL
12:     mark node as leaf
13:   else
14:     add node_1, node_2, ... to Q}

COMPRESSION
1:  for each S[i] {
2:    find its group, i.e., the leaf node can be reached
3:    from the fields in its T[i] }
4:  compress each group individually
```

Figure 3.6: Compression plan generation for inter-packet correlation

Assume that at step k, we had the best order of fields of $f_{i_1}, f_{i_2}, ..., f_{i_k}$ to reach the leaf nodes at level k.

At step k+1, the algorithm expands one of the leaf nodes at level k as a root of a subtree and finds the best field $f_{i_{k+1}}$ for this subtree. Suppose that $f_{i_1}, f_{i_2}, ..., f_{i_k}, f_{i_{k+1}}$ is not the best path for a node at level k+1. There must be at least an $f_j \in f_{i_1}, f_{i_2}, ..., f_{i_k}$ that does not belong to the best path, since $f_{i_{k+1}}$ is the best path for the subtree at level k. However, this violates the assumption that $f_{i_1}, f_{i_2}, ..., f_{i_k}$ is the best path to reach the node at level k. □

For offline compression of packet payloads, the algorithm can train on the entire data set and build the classification tree that will achieve the best compression for the data set. However, for online compression, the classification tree is learnt on a sample training set which may not contain all possible values for some fields. Hence, we also add a value “other” to represent the values not covered for those field in the training set. In the experiments, the top B values in each field are picked up to classify, named *Branch Factor* and all other uncovered values are classified as “other”. For instance, if

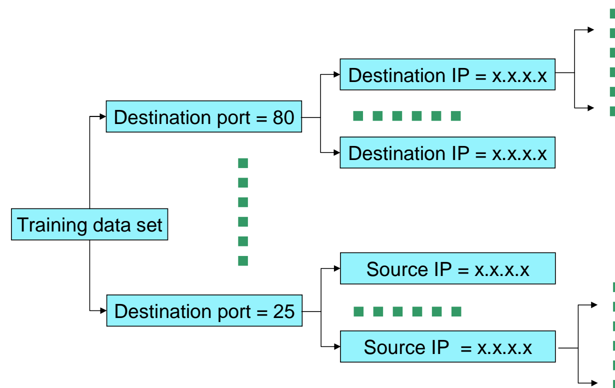


Figure 3.7: Classification tree

there are only the following kinds of layer-4 protocols seen in the training set: *TCP*, *UDP*, *ICMP*, then a value *Other* is introduced to indicate all other layer-4 protocols possible.

IPzip is generic in its definition of compression for unstructured data. Using all the fields in a packet header to build the classification tree gives us a per-packet compression. On the other hand, using only the fields that define a layer-4 flow, i.e, source IP address, destination IP address, source port, destination port and layer-4 protocol, would achieve a per-flow compression. An example of classification tree is shown in Figure 3.7 with its first layer as destination port, and second layer as source/destination IP addresses. In this thesis, implementation and experiments are done on per-flow compression.

3.5 System Architecture

In the previous section, how to learn the intra-packet and inter-packet compression plan, i.e., the order and grouping of the columns for traffic headers, and the classification tree for traffic payloads has been discussed. This section discusses how to apply them to achieve better compression ratios for both online and offline compression. As shown in Figure 3.8, online compression must be high-speed and resource efficient with a goal to be implemented in devices like routers, and offline compression must achieve good compression ratio, since long history of data may require to be stored.

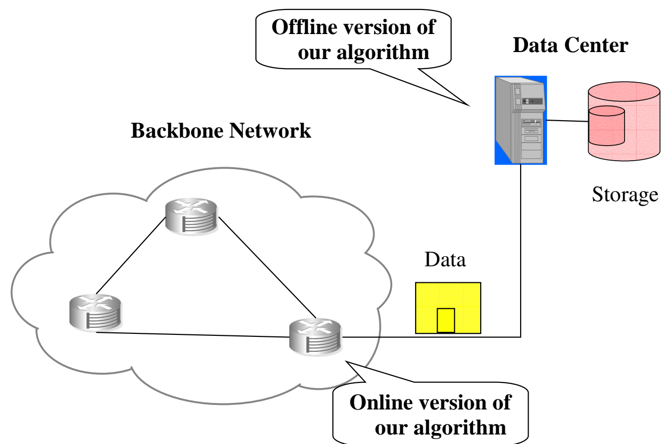


Figure 3.8: Goal of our work

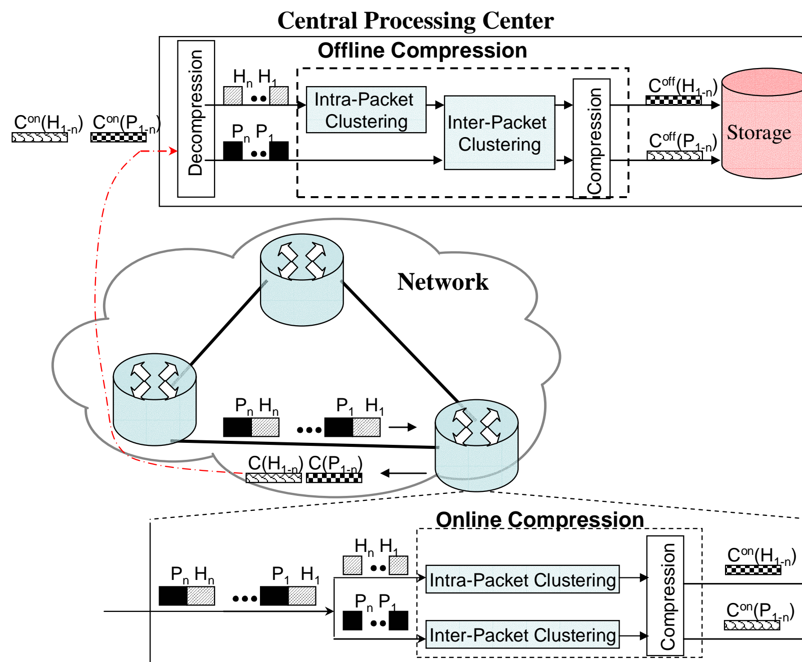


Figure 3.9: System architecture

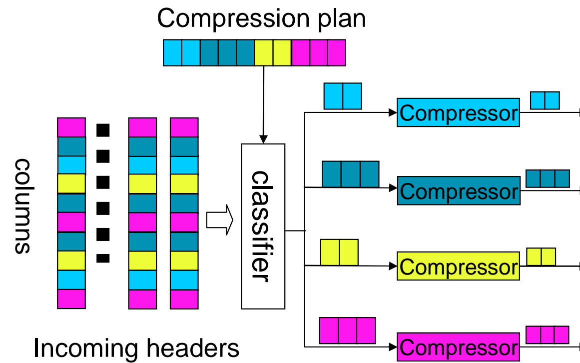


Figure 3.10: Online compression of the headers

Figure 3.9 shows how traffic data are compressed in both online and offline scenarios. In the online case, traffic data are separated to headers (H) and payloads (P). The headers are clustered by the intra-packet compression plan (column rearrangement), and the payloads are clustered by the inter-packet compression plan (row rearrangement), then they are compressed separately and sent to the central processing center. When compressed data arrive at the center, they are decompressed for various analyses. If necessary, they are compressed again by an offline compression algorithm for efficient storage. In offline compression, both the headers and payloads are clustered by inter-packet compression plan (row rearrangement), and headers will be also clustered by intra-packet compression plan (column rearrangement). The details are discussed in section 3.5.1 and section 3.5.2.

3.5.1 Online Compression

Figure 3.10 shows how the headers are compressed in online scenario. The classifier rearranges the order of the columns according to the intra-packet compression plan, which describes which are the high/low compressible columns and which low compressible columns should be grouped together. Then the classifier sends each group to its own compressor. The compressors apply differential encoding to high compressible columns to change them to 0 dominated columns, then compress them by gzip. The low compressible columns are compressed by gzip directly.

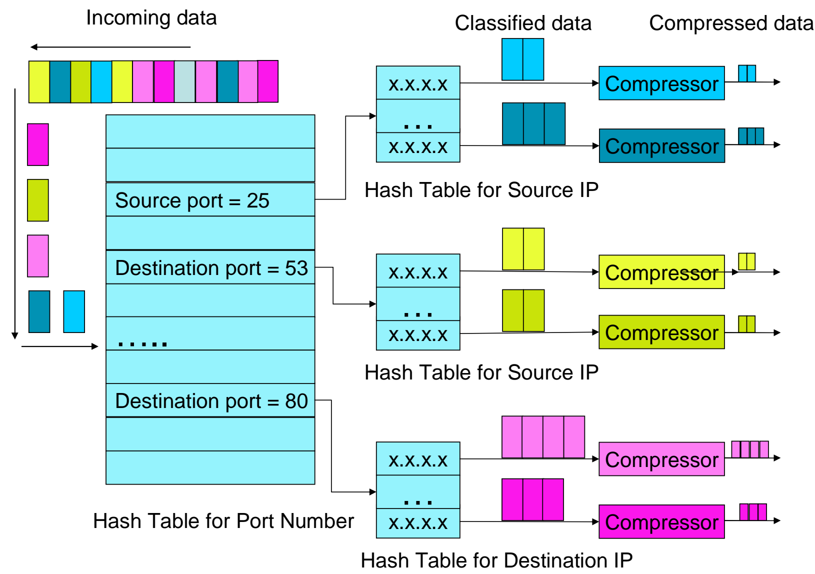


Figure 3.11: Online compression of payloads

Figure 3.11 shows how the payloads are compressed in an online scenario. A two-layer hash table is constructed by keeping the top 2 layers of the classification tree. Figure 3.11 is an example of the hash table. As expected, almost all the top layer classifications are based on either destination port or source port, since lots of applications use fixed port numbers. The second layer may be any other remaining fields, depending on the port number used in the first layer. To make the hash table dynamic, the exact value in the first layer is matched exactly. In the second layer, however, only hashed values are compared. For instance, if the destination IP address is the second layer for destination port 80, then two payloads that have port 80 and the same hashed value of IP addresses are compressed by the same compressor. This fuzzy match dramatically reduces the number of compressors required in online compression with only a little or no decrease in compression ratio. This is because most network traffic has duration, although traffic with different IP address shares the same compressor, most likely they are compressed at different time intervals.

Now it is clear that our online compression consumes very few CPU computations and memory resources. Besides compression, the only computational overhead is the header column grouping and the payload classification. The cost of header column

grouping is only a few operations to rearrange the bytes, and the cost of payload classification is the calculation of two hash values. As a result, our compressor can work at very high speed.

The extra memory required by header grouping is only a few bytes since typical TCP/IP header has only 40 bytes. Since our second layer hash table is dynamic, i.e., no real IP addresses stored, the extra memory required by payload classification is the frequently used port numbers in the first layer, which is only a few hundreds. For those infrequent ones, some default compressors are reserved if their entries cannot be found in the hash table. The dominating cost is the compressors. We choose gzip as our compressor, which needs about 256K memory each. Section 3.6 shows their memory usages.

3.5.2 Offline Compression

The compression ratio of online compression can be further improved. In the online case, only the column correlations inside headers are explored. In the offline case, correlations between headers are also exploited. The same classification is applied to both payloads and headers. An example of offline compression is shown in Figure 3.12. Due to their arriving order, the data look much more random than they should be. Suppose that the correlations inside the data have been identified by the algorithms described in Section 3.4.3, the data are reorganized as follows. First, the correlated columns in headers are rearranged together (in our implementation, columns are bytes. Here header fields are used to demonstrate). Then the headers and payloads are classified by their values in header fields. After that, data are separated in different groups, for example, {timestamp}, {destination ip, destination port} and {payloads}, so that each group are well organized with similar data together. Therefore, better compression ratios can be achieved by compressing each group separately.

In the classification step, instead of the simplified two layer hash table, the classification used in the offline case is the classification tree itself. This finer classification can increase the similarity between neighboring packets to improve the compression ratio at a cost of longer classification time, because the number of comparisons in the

dip-destination IP address, dp-destination port, ts-timestamp, pld-payload

Original Data				Reorder Columns			Reorder Rows			
dip	ts	dp	pld	ts	dip	dp	ts	dip	dp	pld
1.1.1.1	1	80	HTTP/1.1 200 OK..	1	1.1.1.1	80	1	1.1.1.1	80	HTTP/1.1 200 OK..
2.2.2.2	2	25	MAIL FROM:..	2	2.2.2.2	25	5	1.1.1.1	80	HTTP/1.1 200 OK..
3.3.3.3	3	1863	MSG 9294..MIME..	3	3.3.3.3	1863	2	2.2.2.2	25	MAIL FROM: ..
2.2.2.2	4	25	MAIL FROM:..	4	2.2.2.2	25	4	2.2.2.2	25	MAIL FROM:..
1.1.1.1	5	80	HTTP/1.1 200 OK..	5	1.1.1.1	80	3	3.3.3.3	1863	MSG 9294..MIME..
3.3.3.3	6	1863	MSG 9294..MIME..	6	3.3.3.3	1863	6	3.3.3.3	1863	MSG 9294..MIME..

Figure 3.12: Offline compression example

classification may be up to the depth of the tree.

More time and memory are available here because there are less resource constrains in offline compression. In theory, packet classification (intra-packet compression plan) can also be applied before column grouping for online header compression. However, in the online case, we require either a large buffer to rearrange the data or lots of compressors to separate the compression between groups (the number of compressors equals the product of the number of header groups and the number of payload classes), thus the memory usage will be too high to be practical for online devices like routers.

Both online and offline compressed data can be easily recovered. For online case, the headers, which arrive in their original order, are decompressed first. Then from the values in header fields, we can find by which compressor its payload was compressed. Next the data sent from that compressor is decompressed. Data from the decompressed stream are picked up with length specified by the header, and appended to the header. In the offline compression, both header and payload are reordered by the same classification in a similar way as shown by the example in Figure 3.12. In order to recover the data, decompression is done first, then the headers and payloads are sorted together by the timestamp in the headers. If two packets have the same timestamp, their order may not be recovered, which is not important for many applications.

3.5.3 Traffic Pattern Change Detection

Our compression algorithm is able to outperform previous compression algorithms because it exploits in details the inner structure of the data stream under processing. If

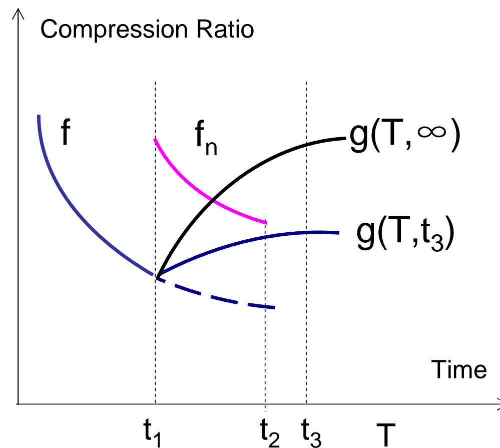


Figure 3.13: Traffic pattern change detection.

the data stream were going to exhibit the same properties over time, then IPzip would never be required to modify its compression plan. Unfortunately, Internet traffic is dynamic and thus IPzip are required to track its performance over time such that it can switch to a more efficient plan as time requires. The following example shows how the simple usage of known predictors, coupled with basic intuitions can effectively solve the above problem. Out approach is validated by using real packet traces in Section 3.6.

Figure 3.13 presents an example that shows the evolution of the observed compression ratio over time. Let's assume that the amount of the original traffic at time t on a generic link is Ct, where C is a constant decided by the rate of the link under consideration. Let's assume that $f(t)$ is a generic function that captures the properties of the Gzip compression algorithm. Hence, the data compressed at any point in time is $f(t)Ct$. If the traffic pattern is stable, then $f(t)$ is a monotonously decreasing function, because the underlying compressor Gzip is asymptotically optimal when the data size is infinitively large (an example is given in Section 3.6).

Let's assume that at time t_1 IPzip observes the compression ratio, denoted as f (solid line) diverging from its expected value (dashed line), as shown in Figure 3.13. The traffic pattern change can be easily detected by using simple predictors such as the ARMA model [5]. At this time, IPzip starts learning a new plan denoted as f_n.

Assume that learning of the new plan is finished at time t_2, and hence the new plan is ready to be applied. The problem now is reduced to determining the time t_3 at which IPzip needs to switch to the new plan. Assume that the interest is in minimizing the overall compression ratio for data within the time range $[0 - T]$, where T represents the time the next traffic pattern change is expected. Note that the parameter T can be computed via time series models [5] that can capture the diurnal and seasonal trends.

The overall compression ratio is defined here as $g(T, t_3)$, which has two parameters: the time range parameter T and the time to switch parameter t_3. Thus, $g(T, \infty)$ represents the decision of not switching to the new plan. Then the compressed size of total data is $S(T, t_3) = Ct_3 f(t_3) + C(T - t_3)f_n(T - t_3)$. The problem now is reduced to finding the optimal value of t_3 that minimizes the overall compressed size $S(T, t_3)$ obtained by solving the differential equation $\frac{\partial S(T,t_3)}{\partial t_3} = 0$. This is equivalent to solving equation $\frac{\partial g(T,t_3)}{\partial t_3} = 0$, where $g(T, t_3) = \frac{S(T,t_3)}{CT} = f(t_3)\frac{t_3}{T} + f_n(T - t_3)\frac{(T-t_3)}{T}$, in which the first and second addition terms represent the compression ratio achieved via the old (f) and new (f_n) plans respectively.

3.6 Experiments

In this section, results of an extensive set of experiments performed on both IP network traffic data and IP network traffic statistics are presented. The results are obtained when using IPzip on real IP datastreams collected from two major ISPs carriers in South America and Asia. The traffic data sets are *truncated packet traces*, comprising of all packet headers and only the first 100 bytes of payload from both ISPs. They are later referred as *Header* and *Payload*. The traffic statistics data set is called *IPVolume* that comprises of the following data records: <StartTime, Duration, TargetIP, BytesIn, BytesOut, PktsIn, PktsOut, FlowsIn, FlowsOut> . IPVolume has a similar structure to the traditional Cisco Netflow. The truncated packet traces are passively collected from OC-48 links (2.5 Gbps) and with 14% and 15% link utilization, respectively. Traffic trace from ISP_1 contains 10 minutes of data and traffic trace from ISP_2 contains 15 minutes of data. The IPVolume data represents a full month of data collected from ISP_1.

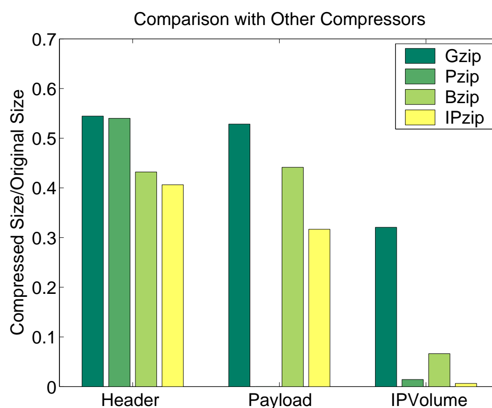


Figure 3.14: IPzip vs Others: Compression rations for flow headers, flow payloads and IPVolume records.

The results are organized into four major sets. First, the overall compression ratios achieved by IPzip is demonstrated when compared to other compressors like Pzip, Bzip and Gzip. Second, we present the results related to the overall performance of IPzip during the training phase in terms of the time required to generate the near-optimal compression plan. Third, we present the results related to the performance of IPzip during the compression phase for both headers and payload data. Fourth, we show how our pattern change detection algorithm works when concatenating together the packet traces collected from the two ISPs; due to the fact that the packet traces exhibit different traffic compositions, at the time of the change, we show how IPzip promptly reacts by generating the new plan and thus is able to keep the performance of compression for dynamic data patterns.

All the results were obtained running tcpdumps on a dual-core 32-bit Linux machine with 2.66GHz CPU and 16GB memory.

3.6.1 IPzip vs Other Compressors: average performance

In Figure 3.14, the performance achieved by IPzip compared with other known compressors like Pzip, Bzip and Gzip, is shown. The *compression ratio* is defined as *(compressed data size)/(original data size)* with smaller values being better. The flow headers, flow payload and IPVolume records collected from ISP1 are used. The truncated flow trace

contains 113 MBytes of header and 290 MBytes of truncated payload, i.e., the first 100 Bytes. The IPVolume records count up to 22 MBytes of data.

As shown in Figure 3.14, for the packet header, IPzip is able to achieve a 40.6% compression ratio, while on the other extreme, Gzip and Bzip can achieve a compression ratio of 54%. Pzip falls in the middle with a compression ratio of 43.2%. For packet payload and IPVolume records, the gain obtained when running IPzip is remarkable. Indeed, for payload data, Gzip cannot do better than 52%, Bzip achieves a 44.1% while IPzip outperforms the previous ones with a 31.7%. Notice that Pzip cannot compress payload data, and thus no results are shown related to this case. Similar results hold true for IPVolume records. The compression ratios achieved by different compressors are 32% for Gzip, 6.6% for Bzip, and 1.4% for Pzip, respectively. IPzip achieves the lowest compression ratio that equals to 0.6%, i.e., over 99% savings of original space. Moreover, due to the fact that the first few bytes of payloads of multiple packets are more likely to be similar than the latter ones, when more bytes in each payload need to be compressed, the performance of IPzip over Gzip decreases. However, even in the limit, the compressed size of IPzip is still 93% of Gzip's size when the entire payload is used (compare that with 74% of Gzip's size when the first 100 bytes of payload are used).

3.6.2 IPzip Training Phase: time required to generate a near-optimal compression plan

In this section, our attention is focused on the performance of IPzip during the generation of compression plans.

As the reader can imagine, the longer the training data set, the better the compression ratio that can be achieved, since more about the structure of the data is learnt. Figure 3.15 shows the compression ratio as a function of the training data size for packet headers and packet payloads. The performance of IPzip when using different group size are shown by the curves. For example, a GX in Figure 3.15(a) refers to the case where no more than X columns are allowed per each column group (see Section 3.4.3). Similarly, a BY in Figure 3.15(b) refers to the case where no more than

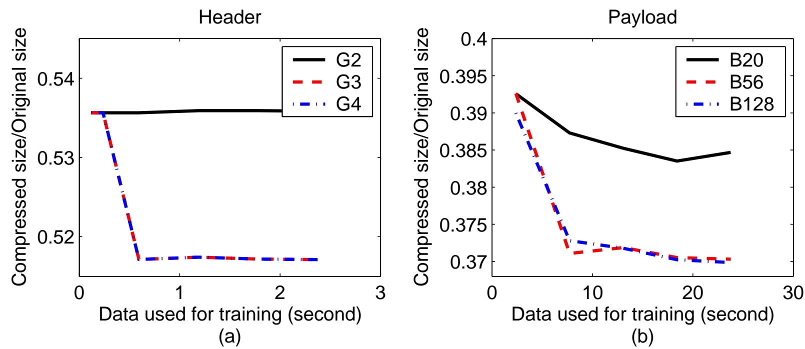


Figure 3.15: Training data size vs Compression ratio

Y number of branches per node are allowed (see Section 3.4.3). First, notice that the compression ratio does not decrease indefinitely, but it reaches a plateau when the training set contains enough data. Indeed, the biggest drop happens at 0.5 seconds for the header and at 7.5 seconds for the payload. The reason for such big difference resides in the fact that IPzip learns intra-packet correlation for headers (correlations between columns inside a packet header) and inter-packet correlation for payloads (correlations between packets) as described in Sections 3.4.3. As a consequence, IPzip has to wait longer in payload training as it needs to receive enough flows to investigate their correlation. Second, notice that the size of compressed data decreases with larger group size for headers and greater number of branches for payloads. However, when the size of groups and number of branches are large enough to capture the hidden correlations, the compressed size does not decrease anymore. For example, for packet headers the compression ratio decreases severely when considering $G3$ instead of $G2$ but does not show any gain when considering $G4$. Similarly for packet payload, the compression ratio improves when using $B56$ compared to $B20$, but no extra saving is achieved when using $B128$.

Next, some results are presented to quantify the cost associated to the generation of the compression plan (Figure 3.16). It can be noticed that the time required to train IPzip for packet headers is super-exponential in terms of group size (as discussed in Section 3.4.3), while the training time for packet payloads increases linearly with the

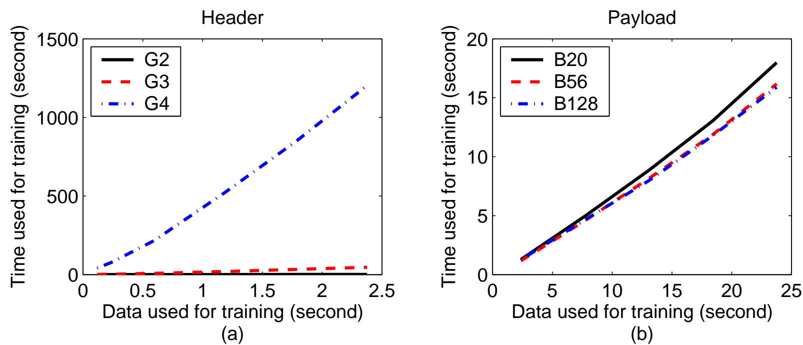


Figure 3.16: Training Phase: Time spent on training

number of branches per node (as discussed in Section 3.4.3). Further, from the previous observation shown in Figure 3.15, the compression ratio does not decrease when the number of groups increases from 3 to 4 or the number of branches increases from 56 to 128, which means that a small group size/branch number has already captured the hidden correlation. Since the training process would be done in central processing station, e.g., in data center, it is possible to buffer several seconds of data for training, and the training costs, i.e., 40 seconds for the header (groups=3) and 16 seconds for the payload (branches=56), would be acceptable in practice.

3.6.3 IPzip Compression Phase: compression ratio and speed

In this section, two important questions are answered. First, how much more IPzip can compress compared with Gzip when used for packet headers and payloads. Second, can IPzip be implemented to compress/decompress data on-the-fly for very high-speed links.

Figure 3.17 shows the compression ratio achieved by IPzip for headers (Figure 3.17(a)) and payloads (Figure 3.17(b)) when applied to a 10 minute trace from ISP_1. The overall compression ratio smoothly decreases over time as more and more about the data under processing is learnt. Compared with Gzip, IPzip can save 4% more for headers and 15% more for payloads when used for online compression. For offline compression, the savings increases even further to 15% and 20% respectively, due to the fact that the

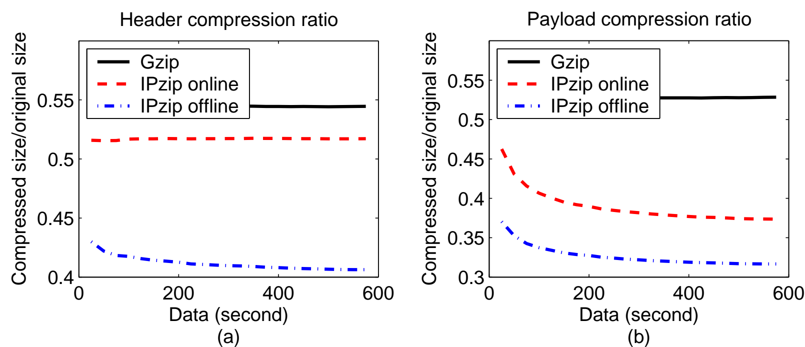


Figure 3.17: Compression ratio over time for Headers (a) and Payloads (b)

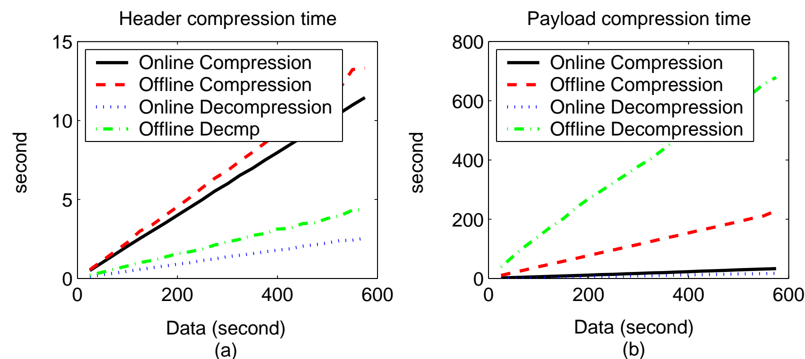


Figure 3.18: Time to compress and decompress Headers (a) and Payloads (b)

offline compression takes into consideration both intra- and inter-packet correlations.

Next, the fact that IPzip is a good candidate algorithm that can run at very high-speed links is shown. Figure 3.18 shows the time required for IPzip to compress/decompress headers and payloads (y-axis of Figure 3.18(a) and Figure 3.18(b), respectively) as a function of the packet arrival rate (x-axis). Notice that to compress the entire 10-minute trace, the online compression time is 12 seconds for headers and 32 seconds for payloads, while the decompression time is 2.5 seconds for headers and 17 seconds for payloads. The offline compression time and decompression time for the payload are longer than the ones observed for the online case. Especially the offline payload decompression, because after decompressing the data, it is still necessary to sort the data according to their timestamp to recover the original order.

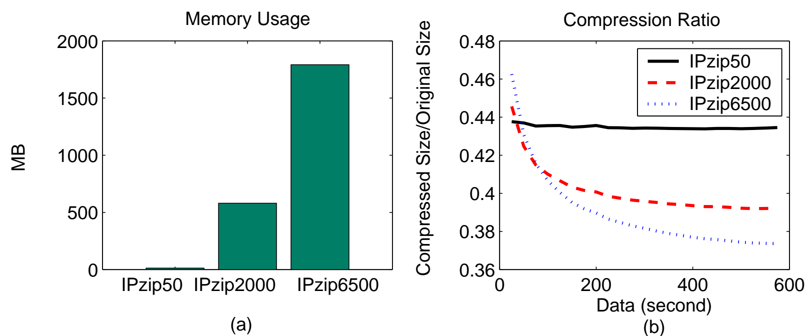


Figure 3.19: Online payload compression: A look at the memory usage

Last in this section, the memory requirement for running IPzip for the online compression of packet payloads is analyzed. In Figure 3.19, we show the memory usage of IPzip as a function of the number of compressors used (see Figure 3.19(a)) and report their associated compression ratio (see Figure 3.19(b)). It can be noticed that the more compressors are used, the more IPzip is capable to compress at the cost of a larger pool of memory. For example, when 2000 compressors are used, IPzip requires 500 MB of memory for the online computation to achieve a compression ratio of 40%. Notice that real systems can be equipped easily today with up to 2 GB of memory, which means that IPzip could be easily implemented using up to 6500 compressors (1.8GB of memory), leading to an overall compression ratio of 37%.

3.6.4 IPzip in a Dynamic Pattern Changing Environment

Until now, the traffic has been assumed to be stationary. In this section, IPzip in a dynamic environment is considered, and our pattern detection algorithm is applied to identify the time at which a degradation of the compression ratio associated to the current plan is observed, and when to switch to the new plan. In order to emulate such scenario, the two packet traces collected from the two ISPs that show strong dissimilarities in their traffic composition (see Figure 3.21) are concatenated. First, the 10 minutes trace collected from ISP_1, and then the 15 minutes trace collected from ISP_2 are used, which means that the traffic pattern change is experienced at a time 300 seconds from the beginning of the entire trace.

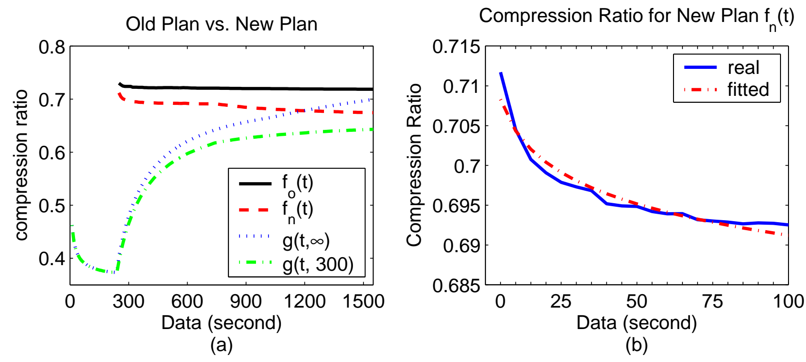


Figure 3.20: IPzip tracking and reacting to traffic pattern changes

Figure 3.21: Traffic pattern change

Figure 3.20(a) presents some results indicating the efficiency of IPzip in adapting to changes in traffic patterns.

Let us define $f_n(t)$ as the function of compression ratio over time t if the newly learned plan is applied to the second packet trace, and $f_o(t)$ as the ratio function if the old plan learned from the first packet trace is applied to the second trace. As discussed in Section 3.5.3, the overall compression ratio $g(t, t3)$ at time t can be computed as $g(t, t3) = \frac{S(t,t3)}{Ct}$, where $t > t_3$ and t_3 is the plan switching time. Thus, the overall compression ratio $g(t, 300)$ and $g(t, \infty)$ represents the two cases that IPzip either switches to the new plan immediately or never.

At the first couple of seconds, although the real compression ratio of old plan $g(t, \infty)$ is still better than the new plan f_n, the trend of the old plan is increasing in ratio values while the new one is decreasing (Figure 3.20 (b)). The fitted function of the new plan obtained with simple curve fitting can be represented as $f_n = -0.0057ln(x) + 0.7175$.

Using the methods described in Section 3.5.3, the final compression ratio $g(1500, t3)$ for every possible switching time $t3$ can be predicated. And the best switching time is the time when traffic pattern changes, i.e., $t_3 = 300$. However, it should be pointed out that this is not achievable because IPzip needs to collect new data points to generate the new compression plan. As a consequence, the actual switching time is represented by the sum of time required to collect the new data samples and the associated training time, which is about 40 seconds in total. Thus the best compression ratio to achieve is 62.4% instead of the optimal $g(1500, 300) = 61.7\%$, as shown in Figure 3.20.

3.7 Summary

Information redundancy dictates the performance of compressors. Our compression algorithm investigates the inner properties residing in IP datagrams, discovers similarities and clusters them in an efficient compression plan that can be easily processed by standard compression algorithms. The effectiveness of our intuitions and data preparation provided by IPzip are deeply analyzed with the usage of real packet traces collected from two large ISPs. IPzip is not dependant to any specific compressor used to process

its output. In this thesis, we use Gzip as our compressor mainly because of the properties of the data analyzed here is characterized by many repeated patterns and thus fits well with the characteristics of Gzip. The future work includes prototyping IPzip in silicon by working with router vendors, since we envision that its better usage will be on compressing IP packets on-the-fly.

Chapter 4

Conclusion

In this thesis, we present the techniques that can be used to perform efficient compression in dynamic systems. Algorithms are proposed to deal with two types of scenarios: the dynamic data and dynamic data patterns. First, the technique of lossy compression of dynamic data is discussed. Investigation is carried out concerning how to build data synopses in databases and how to maintain its approximation accuracy over time-varying data. Algorithms that are capable of keeping the maintenance cost low for both dynamic data and weights are proposed. Second, we study the technique of lossless compression of data with dynamic pattern. New algorithms are invented to solve the problem of traffic compression in IP networks. Our algorithms discover correlation patterns inside traffic packets as well as correlation patterns between packets so that they can be exploited to improve compression ratios. A traffic pattern change detection algorithm is presented to track time-varying patterns and to modify compression strategies accordingly.

In our study, the emphasis is on not only compression ratios, but also their constrains in real world. In the lossy compression in database, we consider non-uniform weights which characterize the real user queries over data. Our algorithms are designed for not only dynamic data, but also dynamic weights. In lossless compression in IP networks, the focus is on both memory usage and compression speed. In online traffic compression, in order to reduce memory cost, the data are classified and they are assigned to different compressors, instead of buffering and reordering the data; in order to improve compression speed, the traffic pattern learning step is carried out offline.

There are other compression scenarios in dynamic systems, such as: (i) lossy compression of data with dynamic patterns, and (ii) lossless compression of changing data.

A typical example of case (i) is the video steam compression, and that of case (ii) is the compression in file system backup, which is called the "deduplication" techniques. The most popular strategies used in these two compression cases are "differential coding", i.e., the current data are compared with the previous data to find their differences, and only the differences are stored.

Although data compression in dynamic environments is a very challenging problem and it has many important applications in the real world, it itself is understudied. Some interesting questions in this area are answered in this thesis through examples from databases and IP networks. There are still more problems to explore. It is expected that the work done in this thesis will inspire more researchers and attract more attentions to discover and solve interesting problems in this field.

References

[1] A. Aboulnaga and S. Chaudhuri, "Self-tuning Histograms: Building Histograms Without Looking at Data," *SIGMOD*, 1999.

[2] S. Babu, M. N. Garofalakis, and R. Rastogi, "Spartan: A model based semantic compression system for massive data tables," *In Proc. of ACM SIGMOD Int'l Conference on Management of Data*, 2001.

[3] N. Bruno, S. Chaudhuri, and L. Gravano, "STHoles: A Multidimensional Workload-Aware Histogram," *SIGMOD*, 2001.

[4] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm", *citeseer.ist.psu.edu/76182.html*, 1994.

[5] C. Chatfield, "The Analysis of Time Series: An Introduction," *Chapman&Hall/CRC* 2004.

[6] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim, "Approximate query processing using wavelets," *VLDB Journal*, Vol. 10, No. 2-3, pp. 199–223, 2001.

[7] S. Chen and A. Nucci, "Dynamic Nonuniform Data Approximation in Databases with Haar Wavelet", *Journal of Computers*, issue 8, 2007.

[8] S. Chen and A. Nucci, "Nonuniform Compression in Databases with Haar Wavelet", *DCC*, 2007.

[9] S Chen, S. Ranjan and A. Nucci, "IPzip: A Stream-aware IP Compression Algorithm", *DCC*, 2008.

[10] V. Chvatal, "A greedy heuristic for the set covering problem," *Mathematics of Operations Research*, vol. 4, no. 3, pp. 233–235, 1979.

[11] C. M. Chen and N. Roussopoulos, "Adaptive Selectivity Estimation Using Query Feedback", *SIGMOD*, pp. 161–172, 1994.

[12] G. Fowler, A. Buchsbaum and D. Caldwell and K. Church and S. Muthukrishnan, "Engineering the Compression of Massive Tables: An Experimental Approach," *In Proc. 11th ACM-SIAM Symp. on Discrete Algorithms* pp. 175-184, 2000.

[13] G. Fowler, A. Buchsbaum and R. Giancarlo, "Improving Table Compression with Combinatorial Optimization," *In Proc. 13th ACM-SIAM Symp. on Discrete Algorithms*, pp. 213-22, 2002.

[14] V. Ganti and M. Lee and R. Ramakrishnan, "ICICLES: Self-Tuning Samples for Approximate Query Answering", *VLDB*, pp. 176-187, 2000.

[15] M. Garofalakis and P. B. Gibbons, "Wavelet Synopses with Error Guarantees", *SIGMOD*, pp. 476-487, 2002.

[16] M. Garofalakis and P. B. Gibbons, "Probabilistic Wavelet Synopses", *ACM Transactions on Database Systems (SIGMOD/PODS'2002 Special Issue)*, pp. 43-90, 2004.

[17] S. Guha and B. Harb, "Wavelet Synopsis for Data Streams: Minimizing Non-Euclidean Error," *KDD*, 2005.

[18] S. Guha and B. Harb, "Approximation Algorithm for Wavelet Transform Coding of Data Stream", *SODA*, 2006.

[19] M. Garofalakis and A. Kumar, "Deterministic Wavelet Thresholding for Maximum-Error Metrics," *PODS*, 166-176, 2004.

[20] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan and M. J. Strauss, "Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries", *VLDB*, 2001.

[21] S. Guha, N. Koudas and D. Srivastava, "Fast Algorithms for Hierarchical Range Histogram Construction", *PODS*, 2002.

[22] M. Garofalakis and R. Rastogi, "Data Mining Meets Network Management: The Nemesis Project," *ACM SIGMOD Int'l Workshop on Research Issues in Data Mining and Knowledge Discovery*, May 2001.

[23] R. Holanda and J. Garcia, "A New Methodology for Packet Trace Classification and Compression Based on Semantic Traffic Characterization," *ITC19*, 2005.

[24] R. Holanda, J. Verdu, J. Garcia and M. Valero, "Performance Analysis of a New Packet Trace Compressor based on TCP Flow Clustering," *ISPASS 05* 2005.

[25] W. H. Hsu and A. E. Zwarico. "Automatic Synthesis of Compression Techniques for Heterogeneous Files," *Software - Practice and Experience,* 1995.

[26] G. Iannaccone, C. Diot, I. Graham and N. McKeown, "Monitoring very high speed links," *In Proc. of the 1st ACM SIGCOMM Workshop on Internet Measurement*, 2001.

[27] H. V. Jagadish, J. Madar and R. T. Ng, "Semantic Compression and Pattern Extraction with Fascicles," *Proc. of VLDB*, pp.: 186 - 198, 1999.

[28] H. V. Jagadish, R. T. Ng, B. C. Ooi, and A. K. H. Tung, "ItCompress: An Iterative Semantic Compression Algorithm," *ICDE* 2004.

[29] N. Koudas, S. Muthukrishnan, D. Srivastava, "Optimal histograms for hierarchical range queries," *PODS*, 2000.

[30] S. Kornexl, V. Paxson, H. Dreger, A. Feldmann and R. Sommer, "Building a time machine for efficient recording and retrieval of high-volume network traffic," *IMC* 2005.

[31] A. C. Konig and G. Weikum, “Combining Histograms and Parametric Curve Fitting for Feedback-Driven Query Result-size Estimation,” *VLDB*, 1999.

[32] Y. Liu, D. Towsley, J. Weng and D. Goeckel, “An Information Theoretic Approach to Network Trace Compression,” *UMass CMPSCI Technical Report 05-03*

[33] Y. Liu, D. Towsley, T. Ye and J. Bolot, “An Information-theoretic Approach to Network Monitoring and Measurement,” *IMC* 2005.

[34] L. Lim and M. Wang, and J. S. Vitter, “SASH: A Self-Adaptive Histogram Set for Dynamically Changing Workloads”, *VLDB*, 2003.

[35] S. Muthukrishnan, “Subquadratic Algorithms for Workload-Aware Haar Wavelet Synopses,” *FSTTCS*, 2005.

[36] V. Markl, G. M. Lohman and V. Raman, “LEO: An autonomic query optimizer for DB2”, *IBM Systems Journal*, Vol. 42, 2003.

[37] S. Muthukrishnan and M. Strauss, “Rangesum histograms,” *SODA*, 2003.

[38] S. Muthukrishnan, M. Strauss and X. Zheng, “Workload-Optimal Histograms on Streams,” *ESA*, 2005.

[39] Y. Matias and D. Urieli, “Optimal workload-based weighted wavelet synopses,” *ICDT*, 2005.

[40] Y. Matias and D. Urieli, “Optimal wavelet synopses for Range-Sum Queries,” *http://theory.stanford.edu/ matias/papers.html*, 2004.

[41] Y. Matias, J. S. Vitter, and M. Wang, “Wavelet-Based Histograms for Selectivity Estimation,” *SIGMOD*, 1998.

[42] Y. Matias, J. S. Vitter, and M. Wang, “Dynamic Maintenance of Wavelet-Based Histograms,” *VLDB*, 2000.

[43] M. Peuhkuri, “A method to compress and anonymize packet traces,” *In Proc. of the 1st ACM SIGCOMM Workshop on Internet Measurement*, 2001.

[44] V. Jacobson, “Compressing TCP/IP headers for low-speed serial links, RFC 1144,” Network Information Center, SRI International, Menlo Park, CA, February, *http://rfc.dotsrc.org/rfc/rfc1144.html*, 1990.

[45] M. Degermark, B. Nordgren and S. Pink, “IP Header Compression, RFC 2507,” *http://rfc.dotsrc.org/rfc/rfc2507.html*, 1999.

[46] RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed, *http://rfc.dotsrc.org/rfc/rfc3095.html*, 2001

[47] A. Shacham, B. Monsour, R. Pereira, M. Thomas, “IP Payload Compression Protocol (IPComp), RFC 3173,” *http://rfc.dotsrc.org/rfc/rfc3173.html*, 2001.

[48] R. Williams, *Adaptive Data Compression*, Kluwer Books, Norwell, United States of America, 1991

[49] N. Thaper, S. Guha, P. Indyk and N. Koudas, "Dynamic multidimensional histograms," *SIGMOD* 2002.

[50] B. D. Vo and K. P. Vo, "Using Column Dependency to Compress Tables," *DCC*, 2004

[51] *http://en.wikipedia.org/wiki/Lossy_data_compression*

[52] *http://en.wikipedia.org/wiki/Lossless_data_compression*

[53] *http://ita.ee.lbl.gov/html/contrib/WorldCup.html*

[54] *http://www.cisco.com/warp/public/116/compress_overview.html*

[55] *http://www.cs.columbia.edu/s̃mb/papers/draft-bellovin-tcpcomp-00.txt*

[56] *http://www.ll.mit.edu/IST/ideval/data/data_index.html*

[57] *http://www.protocols.com/pbook/lan.htm*

[58] *http://www.zlib.net/*

[59] *http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/index.htm*

Vita

Su Chen

Education

May 2008 Ph.D. in Computer Science, Rutgers University
July 2001 M.S. in Computer Science, Fudan University, China
July 1998 B.S. in Computer Science and Engineering, Anhui University, China

Experience

Sep. 2006 - Dec. 2007
Intern, Research Team, Narus Inc., Mountain View, CA
Sep. 2003 - Aug. 2006
Research assistant, Department of Computer Science, Rutgers University, NJ
Jun. 2004 - Sep. 2004
Summer Intern, Telcordia Technologies, Piscataway, NJ
Sep. 2001 - Aug. 2003
Teaching assistant, Department of Computer Science, Rutgers University, NJ

Publications

S. Chen, S. Ranjan, A. Nucci, "IPzip: A Stream-Aware IP Compression Algorithm", *Data Compression Conference*, 2008.

S. Chen, A. Nucci, "Dynamic Nonuniform Data Approximation in Databases with Haar Wavelet", *Journal of Computers*, issue 8, 2007.

S. Chen, A. Nucci, "Nonuniform Compression in Databases with Haar Wavelet", *Data Compression Conference*, 2007.

S. Chen, T. Imielinski, K. Johnsgard, D. Smith and M. Szegedy, "A Dichotomy Theorem for Constraint Satisfaction Problems with Disjoint Domains", *Federated Logic Conference*, 2006.

S. Chen, S. Diggavi, S. Dusad and S. Muthukrishnan, "Efficient string matching algorithms for combinatorial universal denoising", *Data Compression Conference*, 2005.

S. Chen, A. Gaur, S. Muthukrishnan and D. Rosenbluth, "Wireless in loco sensor data collection and applications", *MOBEA II, International World Wide Web Conference*, 2004.