Database Streaming Compression on Memory-Limited Machines

Damon F. Brucoleri
Nova Southeastern University, damon@brucoleri.com

This document is a product of extensive research conducted at the Nova Southeastern University College of Engineering and Computing. For more information on research and degree programs at the NSU College of Engineering and Computing, please click here.

Follow this and additional works at: https://nsuworks.nova.edu/gscis_etd

Part of the Computer Sciences Commons

All rights reserved. This publication is intended for use solely by faculty, students, and staff of Nova Southeastern University. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, now known or later developed, including but not limited to photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the author or the publisher.

NSUWorks Citation
https://nsuworks.nova.edu/gscis_etd/1031.

This Dissertation is brought to you by the College of Engineering and Computing at NSUWorks. It has been accepted for inclusion in CEC Theses and Dissertations by an authorized administrator of NSUWorks. For more information, please contact nsuworks@nova.edu.
Database Streaming Compression on Memory-Limited Machines

by

Damon Bruccoleri

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

College of Engineering and Computing
Nova Southeastern University

2018
We hereby certify that this dissertation, submitted by Damon Brucoleri, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.

Junping Sun, Ph.D.
Chairperson of Dissertation Committee

Wei Li
Wei Li, Ph.D.
Dissertation Committee Member

Jaime R. Raigoza, Ph.D.
Dissertation Committee Member

Approved:

Yong X. Tao, Ph.D., P.E., FASME
Dean, College of Engineering and Computing

3/9/2018
Date

3/9/2018
Date

3-9-2018
Date

College of Engineering and Computing
Nova Southeastern University

2018
Abstract

An Abstract of a Dissertation Submitted to Nova Southeastern University
In Partial Fulfillments of the Requirements for the Degree of Doctor of Philosophy

Database Streaming Compression on Memory-Limited Machines

by

Damon Bruccoleri
March 2018

Dynamic Huffman compression algorithms operate on data-streams with a bounded symbol list. With these algorithms, the complete list of symbols must be contained in main memory or secondary storage. A horizontal format transaction database that is streaming can have a very large item list. Many nodes tax both the processing hardware primary memory size, and the processing time to dynamically maintain the tree.

This research investigated Huffman compression of a transaction-streaming database with a very large symbol list, where each item in the transaction database schema’s item list is a symbol to compress. The constraint of a large symbol list is, in this research, equivalent to the constraint of a memory-limited machine. A large symbol set will result if each item in a large database item list is a symbol to compress in a database stream. In addition, database streams may have some temporal component spanning months or years. Finally, the horizontal format is the format most suited to a streaming transaction database because the transaction IDs are not known beforehand. This research prototypes an algorithm that will compresses a transaction database stream.

There are several advantages to the memory limited dynamic Huffman algorithm. Dynamic Huffman algorithms are single pass algorithms. In many instances a second pass over the data is not possible, such as with streaming databases. Previous dynamic Huffman algorithms are not memory limited, they are asymptotic to $O(n)$, where $n$ is the number of distinct item IDs. Memory is required to grow to fit the $n$ items. The improvement of the new memory limited Dynamic Huffman algorithm is that it would have an $O(k)$ asymptotic memory requirement; where $k$ is the maximum number of nodes in the Huffman tree, $k < n$, and $k$ is a user chosen constant. The new memory limited Dynamic Huffman algorithm compresses horizontally encoded transaction databases that do not contain long runs of 0’s or 1’s.
Acknowledgements

I would like to thank my dissertation committee, Dr. Junping Sun, Dr. Wei Li, and Dr. Jaime Raigoza, for their excellent guidance and help in editing and guiding this manuscript. Their input was significant on several levels, including the challenges they presented, feedback and direction. Finally, I would like to specifically thank Dr. Sun for his mentoring and guidance that started in his Database Management Systems class. His encouragement is greatly appreciated.

I would like to thank my family; my wife Olivia and my three children, Darian, Dalton and Jasmine. Thank you for all your patience, understanding and sacrifice while I conducted my research. It has been several difficult years for all of us while I pursued this degree.

I would like to thank the New York Transit Authority for their employee education program and encouragement while I pursue this degree.
1. Introduction 1
   Background 1
   Problem Statement 10
   Why Streaming? Why Huffman? 12
   Dissertation Goal 13
   Research Questions 16
   Relevance and Significance (Benefit of Research) 17
   Barriers and Issues 18
   Measurement of Research Success 19
   Definition of Terms 20

2. Review of the Literature 25
   The Data Stream 25
   Introduction to Compression 29
   Two Pass Compression of a Transaction Database 35
      Run Length Encoding (RLE) Compression. 36
      Huffman Compression. 40
   Canonical Huffman Codes 43
   Prefix Codes 45
   Adaptive (Dynamic) Compression 46
   Dynamically Decreasing a Node Weight 65
   Frequent Item Counting in Streams 67
   Lossy Compression 73
   Frequent Item-Set Stream Mining 76
   Transaction Database Compression 78
   Other Compression Algorithms 85
   Initial Investigation (Prior Research Work) 88
      Overview 88
      Compression Algorithms Used to Achieve Results in the Initial Study 89
      Conclusion From the prior research 96

3. Methodology 98
   Approach 98
      Discussion of the Proposed Memory Limited Dynamic Huffman Algorithm 98
      Space and Time Asymptotic Complexity 110
      Expansion of the Compressed File 113
      Swap Maximum Bound Analysis 115
“Tail” Items 118
Relationship of distribution and compression ratio 119
Swap Minimum Bound 119
Proposed Work 119
Resources 127

4. Results 128
Verification of Algorithm Coding 128
Performance 130
Optimization of Algorithm 136
Characteristics of Benchmark Transaction Data 140
Database Compression Results 146
  Accidents Benchmark Transaction Database Summary 146
  BMS1 Benchmark Transaction Database Summary 150
  BMS-POS Benchmark Transaction Database Summary 153
  BMS-Webview2 Benchmark Transaction Database Summary 155
  Kosarak Benchmark Transaction Database Summary 159
  Retail Benchmark Transaction Database Summary 161
  T40I10D100K Benchmark Transaction Database Summary 165
  T10I14D100K Benchmark Transaction Database Summary 168
Discussion of Benchmark Compression Results 171

5. Conclusions, Implications, Recommendations 174
Conclusions 174
Implications 178
Recommendations 179

Appendix A Raw Data 181
References 188
Certification of Authorship 195
List of Tables

Tables
Table 1 Benchmark Databases 15
Table 2 Transaction Database Formats 35
Table 3 Golomb Codes for m = 2 to 13 39
Table 4 Huffman and Canonical Huffman Codes 43
Table 5 Illustration of Three Possible Prefix Codes 46
Table 6 Final Huffman Codes After Input String 'Engineering' 59
Table 7 Vertical Versus Horizontal Formats 80
Table 8 Sample Database for Diff-Bits Algorithm 83
Table 9 Calculation of Transactional ID (TID) Differences 83
Table 10 Calculation of Diff-Bits in Bit Vector 84
Table 11 Comparison of Compression Ratio (c/u) Results from prior research 95
Table 12 Comparison of Asymptotic Encoding Time for Compression Schemes 96
Table 13 Comparison of Asymptotic Time/Memory Complexity 117
Table 14 Structure of Benchmark Databases 121
Table 15 Algorithm Verification to Knuth’s Original Grimm Fairy Tale Results 130
Table 16 Bits Produced “3 at a Time” 133
Table 17 Bits Produced “4 at a Time” 135
Table 18 Bits Produced “Word at a Time” 136
Table 19 Measured Data for Modified Frequent Item Identification Algorithm 140
Table 20 Description of Benchmark Transaction Database source data 141
Table 21 Accidents Produced Bits and Minimum Weighted Path Length 147
Table 22 BMS1 Produced Bits and Minimum Weighted Path Length 150
Table 23  BMS-POS Produced Bits and Minimum Weighted Path Length 153
Table 24  BMS-POS Produced Bits and Table Minimum Weighted Path Length 156
Table 25  Kosarak Produced Bits and Minimum Weighted Path Length 159
Table 26  Retail Produced Bits and Minimum Weighted Path Length 162
Table 27  T40I10D100K Produced Bits and Minimum Weighted Path Length 165
Table 28  T1014D100K Produced Bits and Minimum Weighted Path Length 168
Table 29  Comparison of Actual 20% Compression Results to Pareto 173
List of Figures

Figures

Figure 1. Elements of a data stream processing system. 4
Figure 2. Sibling property. 5
Figure 3. FPGA/CPU Architecture for database applications. 6
Figure 4. Glacier source code and circuitry examples. 8
Figure 5. An FPGA data mining architecture. 9
Figure 6. Streaming transactions horizontal format. 10
Figure 7. Two models of compression adaptation. 33
Figure 8. Huffman tree. 42
Figure 9. Pseudocode for static Huffman compression. 42
Figure 10. FGK algorithm tree update pseudocode. 49
Figure 11. Detailed update procedure. 50
Figure 12. Move q to the right of its block. 50
Figure 13. Exchange procedure. 51
Figure 14. Transfer q to the next block subroutine. 52
Figure 15. Encode procedure. 53
Figure 16. FGK algorithm example, 'e' input to tree. 54
Figure 17. FGK algorithm example, ‘n’ and ‘g’ input to tree. 55
Figure 18. FGK algorithm example, 'i' and second 'n' input to tree. 56
Figure 19. FGK algorithm example, input of the two e's in the string 'enginee.' 57
Figure 20. FGK algorithm example, adding the ‘r’ (a) and the final ‘ing’ (b). 57
Figure 21. Sibling property illustration. 60
Figure 22. Core pseudocode for Vitters algorithm. 64

Figure 23. Decreasing the symbol "e" weight by one, to 2. 66

Figure 24. Decreasing a node ‘e’ weight by one, to 1. 66

Figure 25. Decreasing a node ‘e’ weight by one, to 0. 67

Figure 26. 'e' is absorbed by NYT node. 67

Figure 27. Frequent-k algorithm. 69

Figure 28. SpaceSaving algorithm. 70

Figure 29. Frequent-k algorithm example, k = 5. 71

Figure 30. SpaceSaving algorithm example, k = 5. 72

Figure 31. Pseudocode for static Huffman compression. 91

Figure 32. Pseudocode to write Golomb RLE database. 93

Figure 33. Memory limited dynamic Huffman algorithm pseudocode. 101

Figure 34. Overview of the memory limited dynamic Huffman algorithm. 102

Figure 35. Memory limited dynamic Huffman algorithm cases. 103

Figure 36. Rebalancing the Huffman tree. 104

Figure 37. Original algorithm FGK modified to be memory limited. 105

Figure 38. Dynamic Huffman tree for string "enginee." 107

Figure 39. Huffman tree for string "engineer." 107

Figure 40. Huffman tree for string "engineeri," memory constrained. 109

Figure 41. Huffman tree for string "engineerin," memory constrained. 109

Figure 42. Comparison of the final Huffman trees for string "engineering." 110

Figure 43. NYT node configuration. 113

Figure 44. Histogram of item distribution in a database depicting tail items. 118
Figure 45. Expected dynamic versus static compression ratio. 125
Figure 46. One- versus 2-at-a-time compression ratio comparison. 131
Figure 47. Two- versus 3-at-a-time compression ratio comparison. 133
Figure 48. Three- versus 4-at-a-time compression ratio comparison. 134
Figure 49. “4 at a time” versus “word at a time” comparison. 136
Figure 50. Frequent item identification pseudocode. 137
Figure 51. Frequent item identification pseudocode with “Option B.” 138
Figure 52. “Option B” performance. 139
Figure 53. Frequency of items in the accidents database. 141
Figure 54. Frequency of items in the BMS1 database. 142
Figure 55. Frequency of items in the BMS-POS database. 142
Figure 56. Frequency of items in the BMS-Webview2 database. 143
Figure 57. Frequency of items in the Kosarak database. 143
Figure 58. Frequency of items in the Retail database. 144
Figure 59. Frequency of items in the T40I10D100K database. 144
Figure 60. Frequency of items in the T1014D100K database. 145
Figure 61. Accidents static versus memory limited dynamic compression ratio. 148
Figure 62. Accidents actual versus calculated max swaps. 148
Figure 63. Accidents actual versus calculated max swaps (semi-log). 149
Figure 64. Accidents actual swaps versus the compression ratio (semi-log). 149
Figure 65. BMS1 static versus memory limited dynamic compression ratio. 151
Figure 66. BMS1 actual versus calculated max swaps. 151
Figure 67. BMS1 actual versus calculated max swaps (semi-log). 152
Figure 68. BMS1 actual swaps versus the compression ratio (semi-log). 152

Figure 69. BMS-POS static versus memory limited dynamic compression ratio. 154

Figure 70. BMS-POS actual versus calculated max swaps. 154

Figure 71. BMS-POS actual versus calculated max swaps (semi-log). 155

Figure 72. BMS-POS actual swaps versus the compression ratio (semi-log). 155

Figure 73. BMS-Webview2 static versus memory limited compression ratio. 157

Figure 74. BMS-Webview2 actual versus calculated max swaps. 157

Figure 75. BMS-Webview2 actual versus calculated max swaps (semi-log). 158

Figure 76. BMS-Webview2 actual swaps versus compression ratio (semi-log). 158

Figure 77. Kosarak static vs. memory limited dynamic compression ratio 160

Figure 78. Kosarak actual versus calculated max swaps 160

Figure 79. Kosarak actual versus calculated max swaps (semi-log). 161

Figure 80. Kosarak actual swaps versus the compression ratio (semi-log). 161

Figure 81. Retail static versus memory limited dynamic compression ratio. 163

Figure 82. Retail actual versus calculated max swaps. 163

Figure 83. Retail actual versus calculated max swaps (semi-log). 164

Figure 84. Retail actual swaps versus the compression ratio (semi-log). 164

Figure 85. T40I10D100K static versus memory limited dynamic compression ratio. 166

Figure 86. T40I10D100K actual versus calculated max swaps. 166

Figure 87. T40I10D100K actual versus calculated max swaps (semi-log). 167

Figure 88. T40I10D100K actual swaps versus the compression ratio (semi-log). 167

Figure 89. T1014D100K static versus memory limited dynamic compression ratio. 169

Figure 90. T1014D100K actual versus calculated max swaps. 169
Figure 91. T1014D100K actual versus calculated max swaps (semi-log). 170

Figure 92. T1014D100K actual swaps versus the compression ratio (semi-log). 170
Chapter 1
Introduction

Background

Streaming databases provide data processing functions for banking, process control, reservation systems, web analytics, the stock market, and market-basket transactions. A streaming database is a real-time demand. It contains data patterns in a data stream. Compression of the data stream is important for the efficient delivery of the stream over the communications channel, for possible storage of the database (or bounded sections thereof), and for the subsequent processing of the database by specialized hardware. Compression of the data-stream can provide pre-processing of the dataset by identifying frequent items and item-sets. Because it is a stream, multiple pass algorithms to process the streaming databases may not always be possible. In some instances, it may be necessary to obtain results in a single scan for various purposes.

In the streaming transaction, stream $S$ consists of $m$ transactions $t_1, \ldots, t_m$, where $t_1$ is the oldest transaction in the stream, and $t_m$ is the youngest transaction. Each transaction is a set of items. The items in $t$ are drawn from the set of $n$ items $I = \{i_1, \ldots, i_n\}$. Each transaction also includes a transaction id, tid.

Prior research developed several compression algorithms for transaction databases including the run length encoding (RLE) compression techniques using a Golomb code. This is a single pass compression technique. RLE compression using Golomb codes does not require prior knowledge of the item probabilities to achieve good compression (Golomb, 1966). Thus, it is applicable to stream compression. The prior research also
developed a Huffman compression algorithm (Huffman, 1952) for a static database and dataset. The research applied the Huffman static compression to the item IDs in several benchmark transaction datasets. The benchmark datasets had a bounded item list. Multiple passes over the database were possible. In a first pass over the data item ID frequencies were determined. A second pass compressed the dataset. Several of the datasets had large sets of item IDs.

The proposed new research is a progression of the previous research to streams with a very large alphabet. The constraint of compressing a stream means the algorithm needs to be a single pass algorithm. The constraint of a large alphabet is identical to the constraint of compression using a memory-limited machine. Memory here refers to either primary memory or secondary storage. If a machine had enough memory then it would be able to contain the complete Huffman tree, or list, of all the symbols in the alphabet. Thus, the constraint of a large alphabet is identical to the constraint of a memory limited machine. Field programmable gate arrays (FPGAs) are an example of memory limited computing hardware capable of massive and high-speed parallel operation. In the literature, the term *alphabet* is often interchanged with the terms *symbol* or *character*.

This new research proposes to extend the Huffman compression algorithm (Huffman, 1952) to achieve dynamic compression of database streams with a large alphabet on memory constrained hardware. Previous research compared the static two pass Huffman compression to the RLE compression. The static Huffman algorithm compressed a horizontally encoded transaction database. The RLE compression algorithm (using Golomb codes) was used to compress the same transaction databases,
but they were vertically encoded. The new proposed memory limited dynamic Huffman compression algorithm will be used to compress horizontally encoded transaction databases. A transaction data stream will be horizontally encoded since a vertically encoded data stream would require prior knowledge of all transactions. The horizontal encoded transaction databases are commonly structured as a transaction ID followed by one or more item IDs. The vertically encoded databases are commonly encoded as a bitmap. Each row of the bitmap represents an item. Each row is a sequence of 1s and 0s that represent the presence or absence of that item in a transaction. Thus, the complete list of items in a transaction would be assembled by noting the presence of a 1 bit in its column for each item row. Table 7 summarizes several common transaction database formats.

Because a transaction database stream may be responding to real time events, the horizontal format is most commonly used for these systems. The vertical format is more suited for a static transaction database. In the streaming format, as transactions are occurring in real time, each transaction ID and associated item IDs could appear in the stream. Typical elements of a stream processing system are depicted in Figure 1.

The stream processing systems may have several asynchronous input streams and one or more output streams. Because, by definition, the streams are unbounded, portions of the data, aggregations, or queries over the data would be stored. Not the complete dataset.
Dynamic Huffman Compression algorithms exist to compress data streams with bounded item (or token) list (Vitter, 1987; Knuth, 1985; Gallager, 1978; Faller, 1973). They incrementally calculate the Huffman tree from streaming data and compress them dynamically. A key to updating the Huffman tree is that the Huffman tree maintains the ‘sibling property.’ Although the static Huffman algorithm does not maintain the sibling property across all nodes, it is important to the dynamic algorithms of Knuth and Vitter to keep the all the nodes in order of weight (sibling property) to balance the tree. A binary tree has the sibling property “if each node (except the root) has a sibling and if the nodes can be listed in order of non-increasing probability with each node being adjacent in the list to its sibling” (Gallager, 1978). The sibling property is illustrated in Figure 2. Nodes A, B and C may be part of a larger tree. Nodes B and C are siblings. This tree is a Huffman tree if all siblings in the tree can be listed in order of non-increasing probability. Nodes B and C meet that requirement.
An algorithm for adaptive Huffman coding was conceived and proposed by Faller (1973), and Gallager (1978) independently. It was improved by Knuth (1985). Knuth described an efficient data structure for the tree nodes, and an efficient set of algorithms to process the dynamic tree to maintain the sibling property. In the literature this is known as algorithm Faller-Gallager-Knuth or the FGK algorithm (Knuth, 1985). This algorithm is similar to the original Huffman algorithm in that both sender and receiver build the same tree to compress and decompress the stream. The sender performs the compression function and the receiver performs the decompression function. There must be coordination between the sender and receiver to properly restore the data to its uncompressed state.

The FGK algorithm builds the tree dynamically from the frequency of items in the stream. The algorithm will be illustrated and described later in this paper. The node frequencies change as new items arrive in the stream. Both sender and receiver need to update their tree synchronously and dynamically to maintain the sibling property. An aspect of the FGK algorithm is that a new node is added to the tree as each new item arrives, and in certain cases, nodes get exchanged. Nodes are never removed from the
tree. The space complexity of FGK (Knuth, 1985) is $O(n)$, where $n$ is the number of symbols in the alphabet to compress. Thus, one of the challenges of these compression algorithms is in memory-limited machines with many items, or a stream with no bound on the number of items.

An example of a memory limited machine is the FPGA. FPGAs have been proposed for database processing (Mueller, Teubner & Alonzo, 2009a). In Figure 3, three possible architectures are presented. The researchers suggest that data tuples from a network connection could be streamed through the FPGA, and only the results of the query output to the CPU. Similarly, in Figure 3 (b), a stream from a secondary storage device could be processed. The advantages of these two architectures is that tuple processing on the order of hundreds of thousands of tuples per second could be achieved without applying that loading to the main CPU.

As an example of how the circuitry in Figure 3 could be applied, researchers (Mueller, Teubner & Alonso, 2010) have developed a cross compiler that inputs SQL statements and a database schema, and outputs the FPGA circuits. This is depicted in Figure 4. Here, in section (a), is a declared the schema for a database stream. The attributes, datatypes and order in the stream are defined. In Figure 4(b) a SQL statement is declared by a user to filter the database stream. The user is interested in tuple selection of stock transaction trades with a volume larger than 100,000 whose ticker symbol matches the symbol “USBN”. Finally, the user is interested in tuple projection of only the price and volume attributes.

In Figure 5 an architecture for a data mining application is presented using FPGA (Baker & Prasanna, 2009). Here the researcher proposes implementing an FPGA configured as a set of systolic processors to implement the Apriori algorithm for frequent item data mining. In their concept, the transaction database is streamed from some source through the systolic processors and the candidate item sets are built up. First the L2 item sets are generated. Next, the L2 candidate item sets need to be streamed through all the systolic processors to determine the L3 sets. As with the Apriori algorithm, the complete transaction database needs to be streamed (again) through all the systolic processors to prune the L3 candidate item sets. This operation continues until the maximal frequent itemset(s) is determined.
CREATE INPUT STREAM Trades (  
  Seqnr int, -- sequence number  
  Symbol string (4) -- stock symbol  
  Price int, -- stock price  
  Volume int ) -- trade volume

(a) Stream Declaration

SELECT Price, Volume  
FROM Trades  
WHERE Symbol =" USBN" AND  
  Volume > 100,000  
INTO LargeUBSTrades

(b) Textual Query

Large UBSTrades

\[ \Pi \text{Price, Volume} \]
\[ \sigma \text{c:(a,b)} \]
\[ \bowtie \text{b:(Volume,100,000)} \]
\[ \bowtie \text{a:(Symbol,"USBN")} \]

Trades

(c) Algebraic Plan

(d) FPGA Hardware Circuit

Figure 5. An FPGA data mining architecture. Adapted from “Efficient Hardware Data Mining With the Apriori Algorithm on FPGAs” by Z. Baker and V. Prasanna, 2009, Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 3-12.

The research in Figure 3, Figure 4, and Figure 5 are significant for three reasons. The first is it identifies the FPGA, a memory limited hardware element which is the object of database architecture research. Secondly, these architectures might benefit from compression of the data to enable higher speed processing. Finally, they all assume a horizontal encoding scheme of the database. The encoding scheme of a real-time database will be the horizontal transaction format as depicted in Figure 6. In this figure a transaction is created in real time at a cashier. The transaction is inserted into the transaction stream. A vertical format for the transaction database would not be a natural representation since the vertical format requires keying on the item IDs rather than the transaction IDs. The item IDs are a static list of all items in the store. Transaction IDs are created in real time. Keying on the Item IDs would require listing all the Transaction IDs in items list and this would not be possible since all the Transaction IDs may not
have been created yet. *Thus, the Horizontal transaction database format is a natural representation of a real-time streaming database.*

![Real Time Transaction stream](image)

*Figure 6. Streaming transactions horizontal format.*

An algorithm that is proposed to compress a horizontally encoded streaming transaction database on FPGAs is the dynamic Huffman compression algorithm (Knuth, 1985). In the case of item compression on FPGAs or specialized hardware using the FGK algorithm, the space complexity register requirements are $O(n)$ since a node must exists for every item in $I$ (the set of items). This proposal is for a new type of compression algorithm, or variation of the FGK algorithm. This new algorithm could dynamically compute the Huffman tree of only the most $k$ frequent items without needing memory capable of holding all $n$ items, where $k$ is defined as:

$$k < n$$

For instance, $k$ can vary from 1 to $n$ and might be chosen based on available memory on memory limited hardware.

**Problem Statement**

A problem related to compression of transaction database streams on memory-limited machines is identifying the frequent items in a data stream on memory-limited
machines. This is the objective of several algorithms and research (Charikar, Chen, & Farach-Colton, 2002). For instance, the Frequent-k algorithm dynamically finds the $k$ most frequent items in streaming data base $S$ (Demaine, López-Ortiz, & Munro, 2002; Karp, Shenker, & Papadimitriou, 2003). A frequent item identification algorithm (Metwally, Agrawal, & El Abbadi, 2005) is another algorithm used to find the list of most frequent items in streams. Their memory complexity is $O(k)$, rather than $O(n)$. Here $k$ is chosen so that

$$k < n,$$

The number of different items, $i$, in the stream $S$ is $n$ (as previously defined). The number of items to be held in memory is $k$.

A transaction database data stream can be compressed by applying a dynamic Huffman compression on the resulting stream’s items. Algorithms exist for updating a dynamic Huffman tree as a single item arrives in a bounded stream with a bounded item list. The dynamic Huffman algorithms are not designed for memory-constrained machines or to process streams with very large item lists. They expect the tree to grow to accommodate all items in the stream.

The dynamic Huffman compression algorithm as proposed by Knuth (1985) will be extended to accommodate operation on memory-limited machines or to compress database streams with a large set of items.

The new algorithm will be able to limit the required memory size of the dynamic Huffman algorithm. Dynamic Huffman algorithms are single pass algorithms. Conventional, static, Huffman algorithms require two passes over the data to be compressed. The first pass is used to tabulate the frequency of the symbols. The second
pass compresses the data. A single pass compression algorithm is applicable to streaming databases because a second pass may not be possible. The complete stream may not fit into available memory. Additionally, there may be many symbols to compress in the stream. For instance, assume that each item ID in a streaming transaction database is represented as a 32-bit word, and each item ID in the stream is considered a symbol to compress. A large Huffman tree would result if some method to moderate the Huffman tree is not employed.

The work proposed is different than the prior dynamic Huffman algorithms (Knuth, 1985) because the prior work assumes a bounded item list and that the dynamic Huffman tree will fit into memory. A new algorithm will build the Huffman tree, update the Huffman tree item frequencies as new items are added to the tree, or as they become old. It must maintain the sibling property of the Huffman tree and moderate the size of the Huffman tree. Node frequencies will be determined from the frequent item algorithm.

A recognized benefit of the prior dynamic Huffman compression algorithms use on a data stream is that the algorithm is adaptive to temporal changing statistical frequencies of the symbols. Enhancing the algorithm to manage the maximum size of the stored data structure will benefit compression of transaction data streams with large symbol lists. This new algorithm to be researched is called the memory limited dynamic Huffman algorithm.

**Why Streaming? Why Huffman?**

Streaming implies single pass. Reading the database multiple time may not be possible, or may be slower.
Streaming is a real-time demand. A horizontal encoded transaction database may be more natural for a real-time stream. Existing compression schemes for vertically encoded bitmap or tidset transaction database schemas may not be applicable to a real-time stream because they require the transaction IDs to be known beforehand.

Several datamining algorithms exist for horizontal encoded transaction database formats. A Huffman compression algorithm could compress the frequently used item IDs in a transaction database stream.

**Dissertation Goal**

High speed and large throughput data stream mining will require specialized computing hardware to analyze, summarize, monitor and tabulate user queries, perform algorithmic trading, and secure networks. To this end reconfigurable hardware has been used to process the data stream using algorithms realized on a massively parallel scale. For instance, Muller, Teubner, and Alonzo (2009a, 2009b, 2009c, 2010, 2011a, 2011b) have published much research on mining streaming databases with algorithms implemented on a highly parallel scale. In their research, they present a variety of algorithms for *frequent item computation, stream queries, and stream joins* using reconfigurable computing. Other researchers using reconfigurable computing to mine streaming databases are Baker and Prasanna (2005, 2006). Some of this research centers on computing frequent item-sets on transaction databases using systolic arrays. The systolic arrays are implemented using reconfigurable computing hardware. Other researchers are using the reconfigurable hardware to filter XML data streams (Mitra et al., 2009). Other previous work in this research project explored and implemented algorithms for association rule mining using reconfigurable computing. In this previous
research, the algorithms were designed to be massively parallel and fine grained. The algorithm was scalable.

Technology is enabling the implementation of the reconfigurable compute function. Data compression techniques can increase the effective throughput of data that is transferred on a communications channel and the computer hardware. Compression of the data can potentially make use of memory more effectively. Typically, this would be secondary storage. It is also used to more efficiently use primary memory. Similarly, compression can be used to more effectively use the logic gates and interconnects in the reconfigurable computer hardware.

Effective use of the computing hardware can be achieved by compressing the data at the source, and keeping the data compressed during processing. For instance, Baker and Prasanna (2005) propose using an FPGA to implement the Apriori algorithm. They propose a systolic array architecture that might benefit from compression of the data stream between the individual systolic processors. The Viper algorithm (Shenoy, Haritsa & Sudarshan, 2000) proposes compression of the database stream.

This research work will develop a dynamic Huffman compression algorithm for memory-constrained machines. A memory-constrained machine is defined as one where the size of the database to be held in memory, approaches or exceeds the size of the memory. It will benchmark the algorithm using several popular benchmark databases as summarized in Table 1.
Table 1  
*Benchmark Databases*

<table>
<thead>
<tr>
<th>Database</th>
<th>Database source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accidents</td>
<td>Traffic accident data&lt;sup&gt;b&lt;/sup&gt;</td>
</tr>
<tr>
<td>BMS1</td>
<td>KDD CUP 2000: click-stream data from a webstore named Gazelle&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td>Kosarak</td>
<td>Click-stream data of a Hungarian on-line news portal&lt;sup&gt;b&lt;/sup&gt;</td>
</tr>
<tr>
<td>Retail</td>
<td>Market basket data from an anonymous Belgian retail store&lt;sup&gt;b&lt;/sup&gt;</td>
</tr>
<tr>
<td>T10I4D100K</td>
<td>Synthetic data from the IBM Almaden Quest research group&lt;sup&gt;c&lt;/sup&gt;</td>
</tr>
<tr>
<td>T40I10D100K</td>
<td>Synthetic data from the IBM Almaden Quest research group&lt;sup&gt;c&lt;/sup&gt;</td>
</tr>
<tr>
<td>BMS-POS</td>
<td>KDD CUP 2000: click-stream data from a webstore named Gazelle&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td>BMS-WebView2</td>
<td>KDD CUP 2000: click-stream data from a webstore named Gazelle&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

<sup>b</sup>Retrieved from [http://fimi.ua.ac.be/data/](http://fimi.ua.ac.be/data/),  
<sup>c</sup>Agrawal and Srikant (1994)

The dynamic compression results will be compared to the static database compression results that were obtained in the previous experiments (see Chapter 2, “Initial Investigation,” for results). The important metrics to collect are the compression ratio of prototyped algorithms compared to a two pass Huffman Compression (Huffman, 1952) and the RLE compression techniques (Golomb, 1966). The compression ratio will be calculated from the measurements of uncompressed and compressed file bit lengths. The performance of the Dynamic Huffman compression using limited memory algorithm developed in this research will depend on the amount of memory allocated to the algorithm. The algorithm will be run multiple times on the benchmark databases to collect insight on how the allocated memory affects real world compression ratios. A complete list of metrics and measurements will be detailed in Chapter 3, Methodology.

The proposed work prototypes a compression algorithm using a frequent item algorithm to determine item frequencies. A frequent item identification algorithm
(Metwally, Agrawal, & El Abbadi, 2005) will be implemented and integrated into the FGK algorithm.

The overall goal of this research is to facilitate knowledge discovery in streaming databases using reconfigurable computing. This research will facilitate compression of the data stream with a large item list on memory limited devices.

**Research Questions**

A question this research will answer is how the frequent item identification algorithm affects the compression algorithms ability to compress different types of streaming data. The benchmark databases represent several varieties of transaction databases and are used to represent the streaming data. This research will compare the results of the streaming database algorithms with the static compression techniques previously developed and benchmarked.

Secondly, the benchmark databases results will provide insight on how the new algorithms compression ratio varies with the type of data compressed. For instance, it was determined in the initial study that artificial (computer generated) transaction databases compress poorly using the Huffman static compression. Huffman techniques compress well when the data has a probability distribution function that is not uniform. In probability theory, a uniform probability distribution, $f(x)$, has a constant value

$$f(x) = \frac{1}{a - b}$$

over interval $a \leq x \leq b$ and $f(x) = 0$ otherwise (Mendenhall, Beaver, & Beaver, 2012). For example, when a single dice is rolled, there is a uniform probability of rolling any one of the six outcomes. It was determined that for synthetic datasets with a more nearly uniform probability distribution a bit map representation of the database with an
RLE compression technique might offer better compression results. It is expected that the memory limited dynamic Huffman compression will provide similar, poor, compression results as was obtained with the static Huffman algorithm for a synthetic database. This will be verified using the benchmark datasets.

A question this research will answer is what should be the criteria for selecting $k$, the number of nodes to hold in memory. This question has not been found to be explored in the literature. Certainly, this should be related to the total number of symbols in the input stream, $n$. The metric used by most compression techniques to compare algorithms is the compression ratio. The compression ratio is defined as the compressed data length divided by the uncompressed data length. For streaming data, often the compression ratio is defined as the compressed data rate divided by the uncompressed data rate.

**Relevance and Significance (Benefit of Research)**

Because of the explosion of data, stream mining has come to the forefront. In stream mining, the frequency of items $t$ in $T$ (as previously defined) may not be known beforehand and thus would require single pass compression. Additionally, there is a push to perform the mining and queries using specialized hardware and reconfigurable computing (such as FPGAs) for high speed parallel processing to handle the high speed data streams. The challenge with FPGAs is to create massive parallel high speed algorithms with the limited resources on chip. Although algorithms scalable to a number of FPGAs are possible, compression of the data will reduce the number of registers and logic necessary.

This research proposes to apply a single pass dynamic Huffman compression to the transaction database stream to compress the $k$ most frequent items in the database, on
memory limited machines, where \( k \) is a user chosen constant chosen to limit the size of the Huffman tree. It will provide a basis to make the tradeoff between computing memory/time required and the resulting compression ratio.

This research may be beneficial to database storage. Successfully compressing streaming items, \( i \), in \( T \) (as previously defined) could result in a reduction in required primary storage. Although a streaming database is not capable of being stored in primary memory, compression would allow larger sections, or windows, of the database to be stored than could be stored without compression.

This research will be beneficial to the dynamic Huffman compression of data with a large symbol list, or a symbol list where the memory structure is larger than can be fit into available memory.

**Barriers and Issues**

In frequent item stream mining an in-memory data structure holds a list of items and their frequencies. The data memory is limited and as it fills up, less frequent items are pushed out to make way for new items. Thus, the main issue is to update the dynamic Huffman tree using the results from the frequent item-set mining.

Specifically, incrementally updating the Huffman tree and keeping the sibling property using Vitter’s algorithm (Vitter, 1987), as the frequencies change poses problems. Vitter provides a way to update the Huffman tree with a single new item and maintain the sibling property. Vitter does not provide a way to remove items from the tree and rebalance the tree to maintain the sibling property. Knuth (1985), in his algorithm, discusses how to add and remove items from the tree and perhaps by studying Knuth’s method, Vitter’s algorithm (1989) can be extended. Another possible solution is a
brute force approach (Pigeon, 2003). The brute force approach recalculates the complete Huffman tree as new frequent items are found or removed. Pigeon also discusses using a fixed table as previously discussed in the “Questions” section. A fixed table provides some set of fixed Huffman prefix codes. These prefix codes are not calculated dynamically. They are pre-calculated and fixed. In this approach the shortest prefix codes are simply assigned to the most frequent items.

Because of these issues, the method that will be prototyped in this research proposal is to use the FGK dynamic Huffman algorithm and prune the tree using a frequent item algorithm (Metwally, Agrawal, & El Abbadi, 2005). This algorithm will allow reuse of old tree nodes. The deletion of a tree’s node, or decrementing of a node’s weight, will not be required, although it is possible as discussed by Knuth (1985).

**Measurement of Research Success**

There are several measurements that should be met to determine research success. The measurements should be quantitative rather than qualitative to avoid subjective measurement. For instance, some metrics might be: Benefits, Value, Goals/milestones. Value may be a subjective measurement. For this research, the following framework should be used to determine final success:

- The coding of a memory limited dynamic Huffman algorithm that:
  - can compress/decompress a file and is verified against Knuth’s original results.
  - Consumes comparable memory and time as Knuth’s algorithm.
  - provide a ‘dial’ to control the amount of memory consumed.
• Compression ratio comparable to Knuth’s original algorithm when memory is not limited. If not, why?
  
  o Verification of the ability to compress real life transaction database files formats.
  
  o A method to predict or estimate the compression ratio/memory tradeoff for a database application, perhaps based on the distribution of items.

This last item is important because if this algorithm is to be included in a design, then some expectation of the results should be able to be determined in the design stage.

**Definition of Terms**

Adaptive Compression – A type of single pass compression where the algorithm change based on the data being compressed. The algorithm may automatically learn or adapt to the type of data with the goal of increasing the compression ratio, or other metric.

Apriori [algorithm] – A computer algorithm for knowledge discovery in databases. It finds association rules in the data based on a support and confidence. It does this by successively pruning larger supersets of data patterns based on the frequency of its subsets. This bottom up algorithm significantly reduces the number of item sets that are considered.

Canonical [Huffman Code] – It means a ‘useful’ code, given the many different Huffman Code sets that could result depending on the arrangement of the Huffman tree. Canonical Huffman codes are lexicographically ordered by length of the code.
Compression – the act of representing a larger set of data with a smaller set of data. Decompression would be the restoration of the larger set of data, or the approximate restoration of the data, from the smaller compressed set.

Dynamic Compression – see Adaptive Compression

Entropy – Originally defined in the field of thermodynamics, it is a measure of the degree of randomized energy in the system and its ability to produce work. A system with higher entropy is more randomized and has less ability to do useful work. In information science, higher entropy refers to a more randomized signal. Shannon (1948b) reused the word and defined it to be negative the log of the expected value of the probability of an event. The event in this case would be the symbol or bit in a message.

Entropy coding – A lossless Compression technique which is accomplished by removing redundancy in the data.

FPGA – Field Programmable Gate Array. This is a chip that has I/O pins and an internal ‘sea of gates’ whose connections are programmable after manufacture. The internal structure of the FPGA is determined by the manufacturer but can realize any logic function as determined by the users programming. Commonly, FPGA also contain ‘macro’ functions such as memory, phase lock loop clocking circuitry, non-volatile memory, analog conversion, and specialized I/O.

Lossy compression – a compression technique where the reconstituted data only approximates the original signal. The algorithm accomplishes this by partially discarding data. It may also match data as redundant that only
partially matches and then remove the redundancy. Typically used for applications that can tolerate inexactness in the reconstituted data such as digitized photos, video and sound.

Lossless compression – Any compression technique where the reconstituted data is identical to the original data for all data targeted by the algorithm.

LUT – In the context here it refers to Look up Table for an FPGA. From Boolean logic, it is well known that any logic function can be implemented using a NAND gate. FPGA manufacturers do not use the NAND gate as the basic building block of the logic that can be implemented on the FPGA. Rather, they use the LUT. Each LUT typically may have four input bits and a single output bit. Sixteen memory cells hold the logic mapping between input and output. Thus, any Boolean logic function is realizable. Rather than using LUTs to implement flip-flops and registers, manufacturers will include a few flip-flops with each LUT on the chip. Different manufacturers may call these LE’s (logic elements), macrocells, or gates.

Memory constrained – Any real computer system is memory constrained when the application or algorithms requirement for data exceeds the memory limits. This could refer to either secondary storage or main memory constraints.

Mining – The term data mining is a misnomer. A more apt term, and the industry-accepted term, is Knowledge Discovery in Databases. The goal of Mining is the discovery of useful patterns and relationships in the data and not the data itself.
Prefix code – A prefix code can be uniquely (and instantaneously) decoded in the input stream. It is usually a variable length code. Unary coding is a type of prefix code.

Reconfigurable computing – This is the ability of the computing hardware to change its hardware connections either dynamically (at runtime), or more commonly, to adapt itself to the application after its manufacture. Reconfigurable computing was made possible by the development of FPGAs with high density. Reconfigurable computing has the unique characteristic of being able to reconfigure itself during or before runtime to implement a variety of fine grained, massively parallel, algorithms.

RLE – Run length encoding is a simple form of lossless data compression where long sequences of identical symbols (or data patterns) are coded as a data count and symbol (or pattern). A familiar application that uses RLE compression is facsimile transmission. In this application, it is common to scan pages of text which have large sections of white space. If the pixels that represent the white space were encoded as a ‘0’, and the black text pixels as a ‘1’, the RLE model would be tuned to encode the large ‘runs’, or sequences of 0s as a count and a single 0 bit.

Streaming – In the context of this document, refers to an unbounded real-time data transmission.

Transaction database – a database whose records (or tuples) contain the presence, or absence, of items. Each tuple also has a key, also called the transaction id.
Truncated binary coding – a binary coding that is used because of its entropy efficiency. To encode n symbols requires between $k$ and $k+1$ bits, where $k = \lfloor \log_2 n \rfloor$. In this coding the first few symbols can be transmitted with $k$ bits, while the remainder of the symbols require the full $k+1$ bits.

Unary coding – This is a type of prefix code. In this system the number $n$ is represented by $n$ 1s followed by a 0 (or the opposite). Note that the number of bits required to transmit this code increases with the number to transmit. Thus, it is an entropy encoding where the probability of each symbol is given by $P(n) = 2^{-n}$. It is similar to tallying where a mark is drawn for each item to be represented.

XML – Extensible Markup Language is a set of rules for encoding documents. It is text based and designed to be readable by humans, but efficient for machine processing as well. It is the basis for many formats including RSS, SOAP and ATOM. XML formats have become the basis for many applications including Microsoft Office. Part of the format is textual data delimited by text tags.
Chapter 2
Review of the Literature

The Data Stream

Advances in computing have facilitated the collection of continuous data (Aggarwal, 2007). Much of this type of data is generated from simple transactions, such as using a credit card on the telephone, browsing a website with a browser, a stock transaction or the daily itinerary of commercial aircraft. Much of the data flows across IP networks. This data can be mined for interesting relationships for many different applications. When the volume of data is large there are significant challenges:

1. Because of processing time cost constraints, it may no longer be possible to mine the data in multiple passes. A single pass of the data for its processing may be desirable. This will define the algorithm chosen to process the stream. Stream mining algorithms process the data in a single pass.

2. In many cases, there is a temporal component to the data. The data may change with some periodicity given the time of day, season, or perhaps it may evolve apparently randomly given the political situation of the time. Item frequencies may change over time. There is a need for stream mining algorithms to accommodate this temporal or ‘time varying’ component.

This second bullet seems to corroborate this research. The author also notes that stream mining is often accomplished with distributed algorithms/hardware (Aggarwal, 2007).
S. Muthukrishnan (2011) presents some further thoughts on the progress and direction of stream computing. He presents an alternate view of stream computing. He notes that computing capacity, memory and communications have been growing steadily. With it the amount of generated data has also grown and it needs to be analyzed. The generated data streams are created in “massive” rates far higher than can be captured and stored. It arrives at a faster rate than can be sent to a central database without overwhelming the communication’s channel and faster than can be computed. The assumption is that all data can be captured, processed, and stored. For instance, digital signal processing starts with the Nyquist theorem (Nyquist, 1928; Shannon, 1948) that states the sample rate should be twice the highest signal frequency for full reconstruction. Database theory leads to a relational algebra that is continuously applied to the data and is provable to be correct on its results. Communications theory incorporates the thought that there is a minimum number of bits required to transfer the information content. This is Shannon’s concept of self-information (Shannon, 1948). As a future direction of research, Muthukrishnan (2011) notes:

Streaming and compressed sensing brought two groups of researchers (computer science and signal processing) together on common problems of what is the minimal amount of data to be sensed or captured or stored, so data sources can be reconstructed, at least approximately. ...This is however just the beginning. We need to extend compressed sensing to functional sensing, where we sense only what is appropriate to compute different functions and SQL queries (rather than simply reconstructing the signal) and furthermore, extend the theory to massively
distributed and continual framework to be truly useful for new massive data
applications above. (p. 319)

In “Data Streams: Algorithms and Applications” (Muthukrishnan, 2005), the
concept of transmit, compute and store, or TCS, capacity is outlined to differentiate data
stream processing from other ‘normal’ compute data flow. The data stream is data that
occurs as an input to some program at a very high rate. At this rate it may be difficult for
the computing hardware to transmit (T) the entire data to the program. The program may
have limitations on its ability to compute (C) the algorithms and processing necessary on
all the large chunks of data. The program may not be able to store (S) either temporarily
or to archive the data.

This view defines data stream processing as relating to the stress on these
resources.

Another interesting definition of a data stream, that is relevant to this research,
The author demonstrates quite a bit of knowledge and the writings do provide
confirmation of some basic concepts to understand a data stream. Here Collet (2011)
says,

At its most basic level, a file satisfies the definition of a stream. These are
ordered bytes, with a beginning and an end. File is in fact a special stream with
added properties. A file size can to be known beforehand. And in most
circumstances, it is likely to be stored into a seek-able media, such as a hard disk
drive. But a stream is not limited to a file. It can also be a bunch of files grouped
together (tar), or some dynamically generated data. It may be read from a
streaming media, such as pipe, with no seek capability. (p. 1)

Babcock, Babu, Datar, Motwani, and Widom (2002) expound on the unique
aspects of the data stream. They note that the database system cannot control the order in
which the data arrives. Either between data streams, or within a stream. They note that
data streams are unbounded in size and that once a data stream element has been
processed or discarded, it cannot be easily retrieved unless the element was stored. But
typical storage is small compared to the size of the stream.

There are several real-world examples of stream processing database systems
(Babcock et al., 2002). *Traderbot* is a web-based financial search engine that processes
queries over streaming financial data. *iPolicy Networks* processes network packet
streams in real time. It performs complex stream processing, table lookups, URL
filtering, and correlates the data across multiple network flows. Large websites such as
Yahoo may coordinate distributed clickstream analysis to track heavily accessed web
pages. Finally, they cite sensor monitoring as a streaming database application. Large
number of sensors may generate data that needs to be processed and analyzed by the
database management system. An example query from a network management system is
presented. This query will compute the averaged load over one minute on link B and to
notify the operator when the load crosses some value $t$:

```
SELECT notifyoperator(sum(len))
FROM B
GROUP BY getminute(time)
HAVING sum(len) > t
```
Introduction to Compression

Both RLE compression (Golomb, 1966) and Huffman Compression (Huffman, 1952) take advantage of a statistical modeling of the data stream to achieve a level of compression. Without some model of the data, its compression may not be possible (Nelson & Gailly, 1996). For instance, consider all possible 1000-bit messages (Blelloch, 2001). It should be obvious that all possible 1000-bit messages (there are $2^{1000}$ of them) cannot be represented by less than 1000-bits, unless some set of those messages are represented by more than 1000-bits. As a more concrete example, take for example an alphabet that consisted of only four symbols. It is impossible to represent 10 different values using only four symbols (unless multiple symbol combinations are used). The ten decimal digits cannot be compressed to only four symbols. Going back to the 1000-bit example, a model of the 1000-bit messages is required that identifies a subset of those messages and/or some redundancy in the representation of information contained in that message subset. It is reasonable to expect to compress only that subset of messages. For instance, identify a model of data that represents a color static picture, then identify redundancies in the structure of the data. From that model, develop an algorithm and code that compress that subset of messages. Data that falls outside of the model might not be effectively compressed. In fact, data that falls outside the model often result in a larger size when run though a compression algorithm not intended for its model.

Some common ways compression algorithms achieve results is to exploit the redundancies in the data (Salomon, 2004). Redundancies in the database can exist at many different levels; from the bit stream level (RLE), to the identification of repeated symbol patterns (Ziv & Lempel, 1977, 1978), up to the taxonomy of attributes/records as
identified by database normalization (Codd, 1970, 1972). Note that neither of these last
two methods require a statistical model of the data. Non-random data has some sort of
structure. Compression takes advantage of that structure to represent that data in a
smaller version. Ideally, the smaller, compressed, version of the data would not have any
noticeable format.

For instance, Huffman compression takes advantage of the statistical frequency
of characters (or tokens) in the message. In this model of the data used by Huffman,
there is a statistical non-uniformity in the frequency of characters. Messages that fall
outside that model, i.e., do not have a non-uniformity in the frequencies of their data
characters, when compressed, would likely result in ‘compressed’ messages that are
larger than the original message.

As an example, assume a message to be compressed was composed of an alphabet
that was 127 characters and these characters were encoded in 7-bit ASCII. Further
assume that all 127 characters appeared in the original message with equal probability.
The proposed message might consist of 500 occurrences of each of the 127 characters in
the symbol set. Further assume that the occurrence of each of the characters is random
within the message. This message falls outside of the Huffman model. In the Huffman
model there is a non-uniformity in the symbol probabilities. The resulting Huffman
compressed message from this equal probability symbol set would consist of the same
sized message as the original, also encoded in 7-bit characters. This can be confirmed by
trying to build a Huffman tree of the 127 characters where each character has equal
probability of 0.79%. The probability of any of the 127 characters occurring is 100%,
then the probability of a single character, if all had equal probability, is \( \frac{100\%}{127} = 0.79\% \).
Each character would be encoded by the Huffman tree using 7 bits, the same as the original ‘uncompressed’ coding. In addition, the compressed message would contain ‘overhead’. The overhead would at least need to contain some representation of the Huffman tree, or if canonical codes are used, a dictionary of the input characters. The result in this case would be a compressed message that is larger than the original message.

Arithmetic compression (Witten, Neal, & Cleary, 1987) builds on Huffman by taking advantage of inefficiencies in Huffman’s representation of the compressed data. This model recognizes that the ideal number of bits to represent the Huffman token is most often not an even integer.

Lempel-Ziv (Ziv & Lempel, 1977) does not look at individual character’s frequencies as in Huffman or arithmetic compression techniques. It looks for repeated sequential patterns of characters in streaming message. It is effective at compression long strings of repeated characters. The Lempel-Ziv model recognizes recurring patterns in streaming data.

A model of a streaming transaction database presents several features that can be exploited to code a compression scheme algorithm. This research is intended to explore and find that redundancy.

Streaming data has a sender, a receiver, and a communications channel. The sender compresses the data. In some literature, this is also called the coder. The receiver, on the other end of the communications channel, decompresses the data. In some literature, this is called the decoder. Streaming data compression algorithms need to pay
attention to this decompression algorithm as well. Both the sender and receiver need to stay synchronized over the data stream.

The model of compression may be fixed or adaptive. An example of a fixed compression algorithm might be one which compresses ASCII characters and the relative frequencies of the various ASCII character are determined beforehand. Both the sender and receiver require the same model of the data for successful compression to occur. With an adaptive model, both the sender and receiver would respond to changes in the frequencies of the characters as the data is processed.

Adaptive, often called dynamic, compression is a type of single pass compression. Single pass compression algorithms do not need to know the frequency of items to be compressed in advance. They may have ‘meta-information’ about the model of the data but not be tuned to specific probabilities of redundancies in the data. These techniques are applicable to streaming databases because the database may have a temporal component and the frequencies may vary over some period.

Figure 7 shows two models for adaptation. The forward model assembles a packet from the stream. Statistics on the item frequencies or other redundancies are computed over the packet. The compressed stream is transmitted along with the adaptation information. For instance, in the Huffman two pass compression the source transmits the table necessary to reconstruct the Huffman tree.

In the backward adaptation model, encoding is immediately performed using stored information from previous conversions. As new symbols enter the source stream, they are immediately transmitted into the compressed stream (since there is no adaptation
information on them yet). Both the source and destination sides compute the adaptation information synchronously.

![Forward Adaptation Model](image)

**Forward Adaptation Model**

![Backward Adaptation Model](image)

**Backward Adaptation Model**

*Figure 7. Two models of compression adaptation.*

Transaction databases occur in several applications. Finding association rules in market basket analysis is introduced by Agrawal, Imieliński, and Swami (1993). The data typically consists of marketing and transaction information. This might include the date of purchase, a customer ID, a transaction ID, and most importantly the list of items and quantities. In this application, the data may come from a web store, or it may come
from a supermarket. It may come from the register at the checkout counter. Association rules identify those items frequently purchased with other items at the checkout. For example, the association rule might state that 80% of customers who purchased a tablet computer also purchased a black carrying case and a mouse. This type of information may determine the advertisements that appear on a web store checkout page, product placement within a retail store, layout of a mailing flyer, target marketing...

Table 2 illustrates popular and proposed transaction database formats. In the horizontal format illustrated in (a), a transaction id (tid) is followed by a variable length list of item identifiers. The record, or tuple, is variable length. For the purposes of a compression algorithm, each of these ids would be considered a token in the input stream. The tid would be a unique identifier and would have a frequency of 1 in the stream. The item identifiers might not and items that are popular and sold frequently might would have a higher probability. Some algorithms for association rule mining, such as the popular Apriori algorithm, assume the horizontal format for the transaction database. This database format is also assumed by the MaxMiner (Bayardo, 1998) and DepthProject (Agrawal, Aggarwal, & Prasad, 2000) algorithms.

In contrast is the vertical format as illustrated in Table 2 (c). In this format, each tuple is keyed to the item ID. Each tuple contains the item tidset. The tidset is the set of all tids in which the item occurs. Table 2(d) illustrates the vertical bitmap format. This format is used by the Mafia (Burdick, Calimlim & Gehrke, 2001), Viper (Shenoy et al., 2000), Eclat (Zaki, 2000), Charm (Zaki, 2000), and Partition (Savasere, Omiecinski, & Navathe, 1995) algorithms. Compressed vertical bit vectors are used in this format.
Table 2
Transaction Database Formats

(a) Horizontal format

<table>
<thead>
<tr>
<th>ID</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>A, B</td>
</tr>
<tr>
<td>200</td>
<td>D, E</td>
</tr>
<tr>
<td>300</td>
<td>A, C</td>
</tr>
<tr>
<td>400</td>
<td>A, C, E</td>
</tr>
<tr>
<td>500</td>
<td>C</td>
</tr>
<tr>
<td>600</td>
<td>D, E</td>
</tr>
</tbody>
</table>

(b) Horizontal bitmap format

<table>
<thead>
<tr>
<th>ID</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>200</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>300</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>400</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>600</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(c) Vertical tidset format

<table>
<thead>
<tr>
<th>Item</th>
<th>Transaction ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100, 300, 400</td>
</tr>
<tr>
<td>B</td>
<td>100</td>
</tr>
<tr>
<td>C</td>
<td>300, 400, 500</td>
</tr>
<tr>
<td>D</td>
<td>200, 600</td>
</tr>
<tr>
<td>E</td>
<td>200, 400, 600</td>
</tr>
</tbody>
</table>

(d) Vertical bitmap format

<table>
<thead>
<tr>
<th>Item</th>
<th>100</th>
<th>200</th>
<th>300</th>
<th>400</th>
<th>500</th>
<th>600</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Two Pass Compression of a Transaction Database

Data mining, or knowledge discovery in databases (KDD), attempts to find interesting relationships in a database. Association rule mining is a machine learning
algorithm used for KDD. Association Rule learning finds frequent item-sets in a
transaction database, $D$ (Agrawal, Imieliński, & Swami, 1993). Let the set of
transactions in $D$ be

$$ T = \{ t_1, t_2, \ldots, t_j \} $$

where $t_1$ is the first transaction and $t_j$ is the last transaction in the
database. Let the set of items in the database be

$$ I = \{ i_1, i_2, \ldots, i_n \} $$

Thus, $n$ is the number of different items in the database, and $j$ the number of
different transactions in the database. The transactions each have a TID, or transaction
ID. An association rule is defined as

$$ A \Rightarrow B; \text{ where } A \subseteq I, B \subseteq I \text{ and } A \cap B \neq \emptyset. $$

An example of an association rule is $\{ \text{Bread, Eggs} \} \Rightarrow \text{Milk}$. The task is to find
all the frequent association rules in the transaction database, $T$. A maximal frequent item-
set, of length $k$, is defined as

$$ \{ i_1, i_2, i_3, \ldots, i_k \} $$

A maximal frequent item-set is not a subset of a frequent item-set. Finally, a
transaction $t_i$ in $D$ will contain some of the items in $I$. Let the cardinality of items in a
transaction $t_i$ in $T$ be $|t_i|$. 

*Run Length Encoding (RLE) Compression.*

RLE compression notes that in sparse matrices, there will be long runs of 0 bits in
each row. The probability of a 1 bit in the database will be the number of items in the
transaction database $D$, divided by the total number of transactions times the number of
different items in item list.
\[ N = \sum_{i=1}^{j} |t_i| \]

\( N \) is the total number of items in the database. The probability of a 1 bit becomes

\[ p = \frac{N}{j \cdot n} \]

where \( j \) is the number of transactions in the database and \( n \) is the number of different items in the database.

The probability of a 0 bit will be \( 1 - p \).

An RLE algorithm using Golomb prefix codes will compress a long string of 0 or 1 bits with a minimal entropy (Golomb, 1965). Golomb codes first require computation of a factor, \( m \), based on the probability, \( p \), of a 0 bit in the string to be encoded. This quantity is computed as

\[ p^m \approx \frac{1}{2} \]

or,

\[ m \approx -\frac{\log_{10} 2}{\log_{10} p} = \frac{1}{-\log_2 p} \]

Based on work by Gallager and van Voorhis (1975), Salomon (2007) refines this equation. Salomon more accurately obtains:

\[ m = \left[ -\frac{\log_2(1 + p)}{\log_2 p} \right] \]

A larger \( m \) value means a higher probability of a long run of 0 bits. This will be used to calculate a Golomb prefix code whose length is shorter for runs of 0 bits around the ideal mean run length.
A Golomb code consists of two concatenated parts; a $q$ value coded in unary, and an $r$ value with a truncated binary coding. Let the run of 0 bits be $n$ in length. The first step, after computation of $m$, is to compute the three values:

$$q = \left\lfloor \frac{n}{m} \right\rfloor$$

$$r = n - qm$$

$$c = \lceil \log_2 m \rceil$$

The case where $m$ is a power of two results in $c = 0$. This is a special case of the Golomb code and is easier to encode/decode. These are the Rice codes (Rice, 1979). To code the truncated binary quantity $r$, unsigned integers are used to encode the first $2^c - m$ integers using $c - 1$ bits. The rest are encoded using $c$ bits. The Rice code do not require the first $c - 1$ bit codes. See Salomon (2007) for a complete description of Golomb encoding/decoding.

Table 3 summarizes some typical Golomb codes. To use the table, calculate the $m$ value. The length of the run of 0 bits is the $n$ value, then lookup the compression code in the table. If the average number of items in a transaction $T$ can be computed, then RLE using Golomb codes can be used to provide a maximal compression of the database in Table 2 (b) and Table 2 (d) without knowing the individual probability of each item (as is required using Huffman coding.) For an exact value of $P$, the database must be read prior to compression to determine the average number of items in a transaction. Fortunately, an exact value of $P$ not usually required and an approximate value is usually sufficient (Golomb, 1966). This characteristic enables it as a single pass compression algorithm.
### Table 3

Golomb Codes for $m = 2$ to $13$

<table>
<thead>
<tr>
<th>$m$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$2^c - m$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$n$ (number of 0s to compress)

<table>
<thead>
<tr>
<th>$m$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>00</td>
<td>01</td>
<td>100</td>
<td>101</td>
<td>1100</td>
<td>1101</td>
<td>11100</td>
<td>11101</td>
<td>111100</td>
<td>111101</td>
<td>1111100</td>
<td>1111101</td>
<td>11111100</td>
</tr>
<tr>
<td>3</td>
<td>00 010 011 100 1011 1100 11011 11100 111011 111100 1111011 11111000 11111011 11111100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>000 001 010 011 1000 1001 10100 10101 11000 11001 111000 1110011 1111000 1111011 11111000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>000 001 010 0110 10000 10001 101000 1010011 110000 1100011 1110000 11100111 11110000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>000 001 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111 111000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>000 0010 0011 0100 01010 10000 100010 10100 101010 11000 110010 111000 111010 111100 111110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>000 00100 00101 00110 01000 010010 01100 011010 100000 1000010 10100 1010010 110000 1100010 111000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>000 001000 001001 001010 001100 010000 0100010 0110000 01100010 10000000 10000010 10100000 10100010 1100000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>000 0010000 0010001 0010010 0010100 0011000 01000000 01000010 01100000 011000010 1000000</td>
<td>01000001</td>
<td>10100000</td>
<td>11000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>000 00100000 00100001 00100010 00100100 00101000 00110000 010000000 0100000010 0110000000</td>
<td>0110000001</td>
<td>1000000000</td>
<td>1000000001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>000 001000000 001000001 001000010 001000100 001001000 001010000 001100000 0100000000 01000000010</td>
<td>01100000010</td>
<td>0111000000</td>
<td>10000000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>000 0010000000 0010000001 0010000010 0010000100 0010001000 0010010000 00101000000 001100000000</td>
<td>010000000000</td>
<td>011000000001</td>
<td>100000000001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note.** The vertical bar (|) indicates the split between the $r$ and $q$ values.

As an example of compression of a string, assume the following string is to be compressed, 000011001000000001. Assume an exact solution is required and the $m$ value is to be calculated. If an $m$ value is not necessary, then an approximate value can be used. There are 14 zero bits in this sequence of 18 bits. The probability of a 0 bit in the sequence is determined as $p = 14/18 = 78\%$. An exact value of $m$ is

$$\left\lfloor -\log_2 \frac{1.78}{\log_2 0.78} \right\rfloor = \left\lfloor 2.3 \right\rfloor = 3$$

The run of 0 bits in the string are 4, 0, 2, 8. Therefore the string can be compressed as 1010 | 00 | 011 | 11011. The compression ratio is 78%. As a second example, encode the string 0000000000010000000000010000000001. This example has 29 zeros in the sequence of 32 bits. $p = 29/32 = 91\%$. $m$ becomes 8. The compressed string is 10010 | 10010 | 10001. The compression ratio here is 45%. In this second example,
there are many runs of 0 bits about the median run length and the sequence compresses better.

In contrast to the RLE using Golomb prefix codes, the Huffman compression scheme requires a value for each of the item probabilities. A static Huffman compression algorithm requires two passes over the data. If applied to a streaming database, it would introduce a delay in the data as the statistical frequencies of the symbols are determined for the packet. It would fall under the backward adaptation model.

**Huffman Compression.**

Huffman codes provide a minimum entropy-encoding scheme for items (or any tokens) (Huffman, 1952). Huffman codes require knowing the probability of each item’s occurrence in $I$. The total number of items in all transactions in $D$ is given by:

$$\mathbb{N} = \sum_{i=1}^{j} |t_i|$$

If a transaction, $t_i$, contains an item, $i_k$, then $|t_i \cap i_k| = 1$. Given an item $i_k$ in database $D$, the probability, $H_k$, of that item symbol will be

$$H_k = \frac{\sum_{i=1}^{j} |t_i \cap i_k|}{\mathbb{N}}$$

Creation of the Huffman encoding table will require a separate pass over the database to count the number of times each item appears in the transactions to compute each of the $H_k$’s. Other options to reading the entire database would be to sample the database to determine the item probabilities, or to update the probabilities as the transactions in $T$ are written to the database.

To build the Huffman codes the algorithm first creates a list of all the tokens with their associated probabilities. Each item in the list should be a node of a tree. Each of
these nodes are initially unlinked and free. The Huffman algorithm then builds a binary tree bottom-up. It first selects the two nodes with the least probable tokens from the list. It links these two nodes together with a new parent and returns this subtree to the list. The probability of this parent node is the joint probability of its two children. Next, from the list containing the remaining free nodes, and the subtrees, the algorithm chooses the two next least probable items. It links these together with a parent node. This continues until it builds the complete tree, with all the items. The root node of the Huffman tree is the only node left in the list.

The Huffman algorithm labels each of the tree branches and enumerate the Huffman codes in a dictionary or list of input/output symbols. When labeling the Huffman tree, a consistent approach would be to label all left branches a 1 and right branches a 0. Different labeling schemes will result in different Huffman code mappings. The compressed output symbol is the path back to the root. This is the first pass of the algorithm.

In the second pass, the input file is processed again and the corresponding compressed output symbol found in the dictionary to achieve the compression.

An example of a Huffman mapping is presented in Figure 8 and Table 4. In this example an imaginary transaction database, D, has five items in its set of items I, Beer, Butter, Diapers, Eggs and Milk. The probabilities are listed in Table 4; each of the Hk’s, were determined by counting the occurrences of items in a database, D. In this mapping Beer would have the Huffman code of 00 and Milk would have the Huffman code of 110 as shown in the “Huffman Code” column of Table 4. Beer has a shorter prefix code because it has a higher probability of occurring in the database.
Figure 8. Huffman tree.

Create Huffman tree
Input: list of items and probabilities
Output: Huffman tree

1. Create a node for each item. Each node contains item ID and probability.
2. Initialize a list of the nodes
3. Repeat
4. Sort the list of nodes by probability
5. Select two nodes from list containing least probabilities.
6. Link the two nodes with a parent node. The probability of the parent node is the joint probability of the child nodes.
7. Add parent node back to priority list.
8. Until list contains single node.

Figure 9. Pseudocode for static Huffman compression.
Table 4

*Huffman and Canonical Huffman Codes*

<table>
<thead>
<tr>
<th>Item</th>
<th>Huffman code</th>
<th>A canonical Huffman code</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beer</td>
<td>00</td>
<td>00</td>
<td>0.3</td>
</tr>
<tr>
<td>Diapers</td>
<td>10</td>
<td>01</td>
<td>0.3</td>
</tr>
<tr>
<td>Butter</td>
<td>01</td>
<td>10</td>
<td>0.2</td>
</tr>
<tr>
<td>Eggs</td>
<td>111</td>
<td>110</td>
<td>0.1</td>
</tr>
<tr>
<td>Milk</td>
<td>110</td>
<td>111</td>
<td>0.1</td>
</tr>
</tbody>
</table>

**Canonical Huffman Codes**

In general, a Huffman code mapping results in a code that is seemingly arbitrary (they are based on probabilities of course) since it is not unique and is just one code set from many possible mappings. Decoding of an arbitrary Huffman mapping can be more difficult because the decoder needs to replicate the Huffman tree. One way to make the decoding simpler is to use a canonical Huffman code (Schwartz & Kallick, 1964). The canonical Huffman codes add an additional constraint on the generated codes.

An interesting insight into generating the Huffman code can be gained by looking at the Huffman tree. For the purposes of this research, assume that a left branch from a node creates a “0” bit in the resulting compressed code and the right branch a “1” bit. Assume for a moment that the “0” and “1” labeling of one of the left and right child branches are swapped. This is a simple swap that the reader should verify does not change the Huffman tree. What it does change is the generated code. The number of bits that encode each symbol does not change with this swap. Thus, one can conclude, a carefully crafted constraint has the latitude to change the generated Huffman code, as long as the number of bits remains the same, and still be a Huffman tree. This constraint must also maintain the constraint that each compression code be a prefix code. To be a
prefix code, each generated code cannot be a prefix of any other generated code. If the resulting tree is a Huffman tree, it will automatically also generate prefix codes. The algorithm presented by Schwartz and Kallick maintains these features.

Table 4 lists a canonical Huffman code. The first step to building canonical Huffman code is to sort the Huffman table by number of bits in the Huffman code. Next, each group of in the list that have identical number of bits are sorted alphabetically. In this example Beer, Butter, and Diapers are first because they each have a two-bit code. They appear alphabetically.

Each of the canonical codes will be the same length as the original code. The first symbol gets assigned the code a code of all 0’s with the same number of bits as the originally symbol. To determine the canonical code of the next symbol in the list, simply increment the previous symbols code. Do this for all equal length codes. When the next symbol in the list has a longer code word, the previous canonical code is incremented and an additional ‘0’ bit is appended to the least significant bit. Append more than one ‘0’, as is necessary, to maintain the same number of bits as the original Huffman code. A straightforward algorithm re-encodes the Huffman mapping into new codes (Schwartz & Kallick, 1964).

Decoding of a canonical Huffman code is simple and is algorithmically programmable. The original Huffman tree is not required. All that is needed by the decoder is a list of the items and their Huffman code length. The structure of the tree, or the complete list of Huffman code is not required. The decoder can build the canonical code list from the list of items and their Huffman bit length since the codes are ordered. Algorithms (non-table or tree based) exist for decoding canonical Huffman codes. The
canonical code both simplifies the decoding of the compression code by not requiring the original tree, but also decreases the amount of data transmitted. The canonical code decreases the ‘overhead’ that should be considered as part of the transmitted data when calculating the compression ratio.

Prefix Codes

Huffman and RLE compression techniques represent the compressed token as unique characters using a prefix code (Blelloch, 2001). A Huffman code is a particular type of prefix code. In a transaction database stream, where all transaction and item IDs are 32 bits, and assuming no errors in the stream, each unique item ID can be identified. But assume an entropy encoding were to encode the item IDs as a variable number of bits. It may not be possible to identify where an ID ends, and the next one begins, unless some sort of unique stop pattern is used between IDs. The stop pattern would lower the compression ratio. Another solution is to use prefix codes.

To appreciate the feature of a prefix code, refer to the three sets of possible prefix coding in Table 5. Code 1 encoding of the characters has a problem. Assume the following sequence of characters is to be encoded, $x_1x_2x_3$. The encoded bit stream would be 0100. A decoder could interpret this as $x_1x_2x_1x_1$ or even as the sequence $x_4x_3$. The coding does not offer a uniquely decodable sequence. Contrast this with the compression coding of Code 2. Assume the decoder has the simple algorithm of reading bits until either 3 one bits are read, or a 0 is read. With this algorithm, the decoder can uniquely determine the original token as soon as the last bit of the compression code is received. Now consider code set 3. The encoding of Code 3 also offers a unique coding for a bit stream. An algorithm the decoder might use for this coding is to read characters in the
input stream until the second 0 is received. The difference with this is that the decoder cannot determine the original token until after the first bit of the next code word is received (the second 0). Code 3 is not an instantaneous code (Salomon, 2004).

Table 5  
Illustration of Three Possible Prefix Codes

<table>
<thead>
<tr>
<th>Token</th>
<th>Probability</th>
<th>Code 1</th>
<th>Code 2</th>
<th>Code 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x₁</td>
<td>0.500</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x₂</td>
<td>0.250</td>
<td>1</td>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td>x₃</td>
<td>0.125</td>
<td>00</td>
<td>110</td>
<td>011</td>
</tr>
<tr>
<td>x₄</td>
<td>0.125</td>
<td>01</td>
<td>111</td>
<td>0111</td>
</tr>
<tr>
<td>Average weighted length</td>
<td>1.125</td>
<td>1.75</td>
<td>1.875</td>
<td></td>
</tr>
</tbody>
</table>

Code 2 is the only prefix code because it is uniquely and instantaneously decodable in the input stream. There are many different possibilities for prefix codes other than the sequence presented as Code 2.

A code is uniquely decodable iff for each source symbol, \( s \in S \), a valid coded representation \( b \) exists, and the representation \( b \) is unique for every possible combination of source symbols \( s \) in \( S \), where \( S \) is a stochastic process.

It is a simple matter to build up other prefix codes. For instance, following the procedure given by Huffman, an infinite variety of prefix codes can be generated. As another example, all fixed length codes are prefix codes; such as ASCII codes. Because fixed length codes are all the same length they are uniquely decodable in the input stream.

Adaptive (Dynamic) Compression

Adaptive Huffman coding was first introduced by Faller (1973), independently introduced by Gallager (1978), and then further refined by and Knuth (1985). It is known
as the FGK algorithm. In the traditional Huffman algorithm, the source is read (in a first pass over the data) to determine the symbol frequencies. The algorithm reads the source again (in the second pass) to compress the data. Often, such as the case with a streaming database, this is impractical. Another example where this would be impractical is where a very large dataset is stored in secondary storage and the time to read it in a first pass is deemed impractical. Dynamic or adaptive compression identifies the data source and a destination to achieve the compression in a single pass over the data. Both the source and destination work together, they mirror each other. Both start with an empty Huffman tree and build it dynamically, as new symbols in the stream arrive, and the tree must be identical on both ends. On both ends, as symbols are added to the tree, the tree must be examined to see if it is still a Huffman tree, and rebalanced if it is not. Notice that since both sides build the tree dynamically there is some savings in data transmission since the tree does not have to be initially transmitted as is required in the two pass algorithm. On the other hand, the source symbol frequencies have to be learned, and there is some inefficiency as each side asymptotically reaches the ideal source entropy. The FGK algorithm updates the frequencies in the Huffman tree dynamically as new items arrive in the stream. It also rebalances the tree to maintain the sibling property. A key point, and the key to keeping the receiver and transmitter Huffman trees in synchronism, is what happens when a symbol that neither the transmitter nor receiver have seen yet is received in the stream. In this case, a special escape symbol, the NYT (Not Yet Transmitted) symbol, is transmitted with the new uncompressed symbol, so each side can build the identical tree with the new character. The NYT symbol is defined to have a frequency of 0. It is a node in the Huffman tree. This is the longest code. As new symbols arrive in
the stream, their frequencies in the tree are updated. If the character is not seen before, the NYT node become a parent node and is split into a new NYT node with frequency 0, and a new node is added to the tree that represents the new character with a frequency of 1. Next, the tree may need to be rebalanced and all the parent nodes may have to be incremented.

The first option in rebalancing the tree is to simply rebuild the whole tree when the tree is no longer a Huffman tree. This is neither Vitter’s (1987) algorithm nor FGK (1985), but it is an option for a dynamic compression scheme. To tell if the tree is a Huffman tree, scan the nodes, from left to right and bottom to top, each leaf and parent node. The node frequencies should be in sorted, non-descending order. This is referred to as the sibling property. Rebuilding the whole tree from scratch every time can be a lengthy process (Pigeon, S., 2003). A second option would be to completely rebuild the Huffman tree after some ‘arbitrary’ number symbols are received in the input stream, say 100. This option could result in non-optimal compression ratios, but would reduce the required processing time. A third option (Pigeon, 2003) is to rebuild the tree when the symbols rank has significantly changed. In the implementation proposed by Pigeon a table is kept with the list of input symbols and frequencies. As new symbols arrive in the input stream the frequencies are updated and the table sorted by frequency. When a swap occurs due to sorting, the Huffman tree is rebuilt. Pigeon points out that the table operations coding is more efficient than Vitter’s algorithm, but on the other hand the tree rebuilding is costly.

The FGK algorithm rebalances the tree more compute efficiently for incremental updating of the frequency of a single symbol using the algorithm as outline in the
pseudocode given in Figure 10. The ‘block’ (in line 2) is defined as the set of nodes with the same weight.

<table>
<thead>
<tr>
<th>Pseudocode FGK Update Huffman Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Huffman Tree, pointer to Node N. N is the node to increment.</td>
</tr>
<tr>
<td><strong>Output:</strong> Updated Huffman Tree.</td>
</tr>
</tbody>
</table>

1. **Repeat**
2. Exchange N with the last (rightmost) node in its ‘block’.
3. Increment the frequency of node N.
4. \(N \leftarrow \text{parent of } N.\)
5. **Until** N is the root
6. Increment frequency of the root

*Figure 10.* FGK algorithm tree update pseudocode. Adapted from “Dynamic Huffman Coding” by D. E. Knuth, 1985, *Journal of Algorithms, 6*(2), 163-180.

As an example, in Figure 17(b) nodes 2, 3, 4 and 5 are in the same block because their weight is 1. More detailed pseudocode for this same algorithm follows. As Knuth (1985) says, “The heart of the dynamic Huffman tree processing is the update procedure.”

*Figure 11* lists pseudocode for the update procedure as presented by Knuth (1985). The input to the procedure is the symbol to encode, \(k.\) A following procedure uses \(k.\) It is not used here. The data structure, \(P,\) is an array of backward links to the parent of the node \(q.\) Line 2 sets \(q\) to be the node whose weight should increase. Note the math when indexing the array \(P.\) \(P\) is the pointer to node parents and has a range of 1 to \(n,\) where \(n\) is the number of nodes (and is a constant). The parent of node \(2j\) and \(2j-1\) is \(P[j].\) When \(q\) becomes 0, the root node is reached. The procedure calls in lines 4 and 5 follow.
Knuth update procedure

1. procedure update (integer k);
2. (Set q to the external node whose weight should increase);
3. while q > 0 do
4.   (Move q to the right of its block);
5.   (Transfer q to the next block, with weight one higher);
6. q ← P [(q + 1) div 2] od;
7. end;


Two new arrays here are “B” and “D”. B is an array of pointers to the blocks. All nodes j of a given weight have the same value of B[j]. The D array is an array of pointers to the largest node number in each block. Both arrays have a range of from 1 to 2n-1. This subroutine moves node q to the right of its block, unless both q and it parent are at the right of its block already. This subroutine uses the ‘exchange’ procedure. The ‘exchange’ procedure in Figure 13 exchanges two subtrees (as long as neither is the child of the other).

Knuth <Move q to the right of its block>=

1. if q< D[B[q]] and D[B[P[(q + 1) div 2]]] > q + 1 then
2.   exchange (q, D[B[q]]); q ← D[B[q]] fi

Figure 12. Move q to the right of its block. Adapted from “Dynamic Huffman Coding” by D. E. Knuth, 1985, *Journal of Algorithms*, 6(2), 163-180.
The subroutine shown in Figure 14 will update the weight of $q$, and it will update the weight of $q$'s parent if it has the same weight (Knuth, 1985). This subroutine introduces arrays $A$, $L$, $G$, and $W$. Array $A$ has a range of 0 to $n$. It is an array of the symbols. Arrays $L$ and $G$ are the left and right pointers to the blocks. They have a range of 1 to $2n-1$. Array $W$ is the weights of each block. Block $k$ has a weight of $W[k]$. Its range is 1 to $2n-1$. 

```
1. procedure exchange (integer $q$, $t$);
2. begin integer $ct$, $cq$, $acq$;
3. $ct \leftarrow C[t]$; $cq \leftarrow C[q]$; $acq \leftarrow A[cq]$;
4. if $A[ct] \neq t$ then $P[ct] \leftarrow q$ else $A[ct] \leftarrow q$ fi;
5. if $acq \neq q$ then $P[cq] \leftarrow t$ else $A[cq] \leftarrow t$ fi;
6. $C[t] \leftarrow cq$; $C[q] \leftarrow ct$;
7. end;
```
Figure 14. Transfer $q$ to the next block subroutine. Adapted from “Dynamic Huffman Coding” by D. E. Knuth, 1985, Journal of Algorithms, 6(2), 163-180.

The Encode Procedure of Figure 15 accepts a symbol to encode, $k$. The symbol is ‘looked up’ in a simple hash table, the $A$ array, in line 3. Typically, the $A$ array stores the pointer to the external node containing the symbol. If the symbol is not stored in the tree, then contents of $A$ contain a value less than $M$ to encode the zero-weight symbol. $M$ is the number of zero weight symbols. The nodes are stored in positions $2M-1$ through position $2n-1$. $M$ is calculated from $E$ and $R$. $M = 2^E + R$, where $0 \leq R < 2^E$. 

Knuth < Transfer $q$ to the next block, with weight one higher) >=

1. begin integer $j$, $u$, $gu$, $lu$, $x$, $t$, $qq$;
2. $u \leftarrow B[q]$; $gu \leftarrow G[u]$; $lu \leftarrow L[u]$; $x \leftarrow W[u]$; $qq \leftarrow D[u]$;
3. if $W[gu] = x + 1$ then
4. $B[q] \leftarrow B[qq] \leftarrow gu$;
5. if $D[lu] = q - 1$ or ($u = H$ and $q = A[O]$) then comment block $u$ disappears:
6. $G[lu] \leftarrow gu$; $L[gu] \leftarrow lu$; if $H = u$ then $H \leftarrow gu$ fi;
7. $G[u] \leftarrow V$, $V \leftarrow u$;
8. else $D[u] \leftarrow q - 1$ fi;
9. else if $D[lu] = q - 1$ or ($u = H$ and $q = A[O]$) then $W[u] \leftarrow x + 1$;
10. else comment a new block appears;
11. $t \leftarrow V$; $V \leftarrow G[V]$;
12. $L[t] \leftarrow u$; $G[t] \leftarrow gu$; $L[gu] \leftarrow G[u] + t$;
13. $W[t] \leftarrow x + 1$; $D[t] \leftarrow D[u]$; $D[u] \leftarrow q - 1$;
14. $B[q] \leftarrow B[qq] \leftarrow t$ fi;
15. fi;
16. fi;
17. $q \leftarrow qq$;
18. end
1. **procedure** encode (integer k);
2. **begin** integer i, j, q, t;
3. i ← 0; q ← A[k];
4. **if** q ≤ M **then comment** encode zero weight;
5. q ← q - 1;
6. **if** q < 2 × R **then** t ← E + 1 **else** q ← q - R; t ← E **fi**;
7. **for** j ← 1 **to** t **do** i ← i + 1; S[i] ← q mod 2; q ← q div 2 **od**;
8. q ← A[0] **fi**;
9. **while** q < Z **do**
10. i ← i + 1; S[i] ← (q + 1) mod 2;
11. q ← P[(q + 1) div 2] **od**;
12. **while** i > 0 **do transmit** (S[i]); i ← i - 1 **od**;
13. **end**;


Line 6 calculates a temporary value, t, that will be used to loop through the symbol to collect the bits to transmit. The bits are put onto a stack, S, in line 7. Line number 8 sets q to point to the NYT node. Lines 9 through 11 traverse q up to the root node and put the code for the NYT node on the stack. Z points to the node that contains the root of the tree. Line 11 sets q to its parent. Line 10 determines if q is an odd or even number. If it’s odd, it’s a right child, if it’s even, it’s a left child, and adds a 1 or 0 to the stack to transmit. Line 12 transmits the bits just stored on the stack.

The NYT node represents all the “as of yet” unseen symbols. It is emitted prior to emitting any new symbol. It is used to keep the encoder and decoder in synchronism. When the decoder sees the NYT symbol, it will be used to indicate a new symbol follows and to split the new symbol out of the NYT symbol in the tree.

As an example of the steps an FGK compression algorithm would take, assume the string ‘engineering’ are the first characters to appears in the input stream. Assume the
character are encoded with an 8 bit ASCII code. The Huffman tree starts out with an empty tree as depicted in Figure 16(a). The NYT node is split, a new node that represents the ‘e’ is assigned to the split node, and a both nodes are assigned to a new parent as depicted in Figure 16(b). This is an example of a new symbol splitting out of the NYT node. The NYT node represent ‘all symbols’ with weight 0. All the symbols that may be received, but as of yet are unseen. When the NYT node is split it is equivalent to pulling one of the symbols out of it. In the figure, the number inside the node represents the node symbol frequency, and the number outside the node represents the node number.

The emitted, or output stream would be the 8-bit ASCII code for the single symbol e:

Input stream : e
Output stream : e

Figure 16. FGK algorithm example, 'e' input to tree.

Note that the NYT symbol for the first symbol is not emitted (transmitted) into the output stream. If it were emitted, then the output stream would contain 0e. It does not have to be emitted because for the first symbol only, it can be assumed to be emitted.

In Figure 17(a) the ‘n’ is added to the tree. In Figure 17(b) the second ‘g’ is input and the tree is updated to reflect the new node weighting. In Figure 17(b) the parent of
the new node (the ‘g’) is interchanged with the last node in the block of nodes that has the same weight as it. The last node in that block is then incremented by 1. Therefore, the leaf node that contains the ‘e’ moves to the left side of the root.

The output stream now has the addition of the two NYT symbols, the n and the g symbols. Notice the NYT code changes dynamically.

Cumulative input stream: eng

Cumulative output stream: e 0n 00g

Figure 17. FGK algorithm example, ‘n’ and ‘g’ input to tree.

In Figure 18 (a) the ‘i’ is added to the tree. Because the parent of the ‘g’ now has a frequency of 2, its subtree will be exchanged with the highest numbered node that has the same weight. In this case, that is the leaf node that represents the ‘e’.

In Figure 18(b) a second ‘n’ character is input. This symbol is not in the tree. Since the ‘n’ node is the highest numbered node in its weight group, it is not exchanged with any node. The ‘n’ node is incremented. Now the algorithm moves up to the parent. That node is also the highest numbered node in its group. In fact, it is the only node in the group with a weight of three. An exchange of this node is not required.
Cumulative input stream: engin
Cumulative output stream: e 0n 00g 100i 11

Figure 18. FGK algorithm example, 'i' and second 'n' input to tree.

The addition of the two e’s results in the trees shown in Figure 19 (a) and Figure 19 (b). Finally, the input of the last part of the string, ‘ring’, causes a few exchanges of parent nodes as the algorithm recursion travels up the Huffman tree. Figure 20 shows the final tree.
Figure 19. FGK algorithm example, input of the two e's in the string 'enginee.'

Figure 20. FGK algorithm example, adding the 'r' (a) and the final 'ing' (b).
Assuming that the input characters are coded with an 8 bit ASCII code, the effective compression ratio would be \((5*8+25)/(11*8) = 74\%\). The compression ratio is defined as the compressed string length divided by the uncompressed string length. In this calculation, each symbol is assumed to be an 8-bit ASCII character. There are 5 input ASCII characters in the output stream and 11 ASCII characters in the input stream. Shannon, in his 1950 paper “Prediction and Entropy of Written English”, calculates the ideal as about 1.5 bits per character in the 27-letter written English. The ideal compression ratio to be asymptotically reached by a dynamic Huffman compression algorithm would be \(1.5/8 = 18.75\%\).

Table 6 summarizes the weights (or frequencies) of each of the leaf nodes and the generated compression code for each symbol at the termination of the input stream. It is interesting to compare the dynamically generated tree in Figure 20(b) to a tree built with Huffman’s original algorithm. To make an equivalent comparison, initially a list is created that contains the NYT node and node’s whose contents are the tuple of the symbol and the frequency as listed in Table 6.
Table 6

*Final Huffman Codes After Input String 'Engineering'*

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Frequency</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>N</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>I</td>
<td>2</td>
<td>00</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
<td>011</td>
</tr>
<tr>
<td>R</td>
<td>1</td>
<td>0101</td>
</tr>
<tr>
<td>NYT</td>
<td>0</td>
<td>0100</td>
</tr>
</tbody>
</table>

The first step by the static Huffman would be to combine the NYT and the ‘r’ leaf node into a subtree because these are the lowest frequency (probability) in the list of Table 6. The resulting subtree has a frequency of 1. Next, this subtree is combined with the ‘g’ leaf node. There is another choice here, the ‘i’ leaf node, because it has the same frequency, but choosing the ‘g’ will eventually lead to the tree determined dynamically. This subtree has a frequency of 3. Next, combine the leaf node ‘i’ and the subtree with the frequency 3 to obtain a new subtree with a frequency of 5.

At this point, there are three items left in the list. These are the subtree with the frequency 5, and the ‘e’ and ‘n’ nodes (each with a frequency of 3). The next step in Huffman’s algorithm is to combine the ‘e’ and ‘n’ nodes to obtain a subtree with frequency 6. Finally, two subtrees are left with frequencies 6 and 5. These combine to obtain the Huffman tree with a root that has a frequency of 11. Thus, the tree determined with the static algorithm would be identical to the dynamically generated tree of *Figure 20*(b). Generally, the trees determined by the static Huffman algorithm and the FGK algorithm will not be identical, Vitter (1987) discusses this further. Differences in the shape of the tree can stem from choices in building the tree when two nodes (leaf or internal) have the same weight and from the rebalancing procedure..
Figure 21 more clearly illustrates the sibling property. Gallager (1978) defines a binary tree as having the sibling property “if each node (except the root) has a sibling and if the nodes can be listed in order of non-increasing probability with each node being adjacent in the list to its sibling.”

Figure 21 lists each node in the tree. The list of nodes is in order of decreasing probability. The table illustrates that each node has a sibling (except for the root node). Further, each node is adjacent to its sibling in the list. More formally, if the tree holds \( K \) symbols, then Knuth shows that the tree has \( 2K-1 \) nodes. For each \( k \), where \( 0 < k < K - 2 \), the \( 2^k \)th and the \( (2^k-1) \)th element must be siblings (Knuth, 1987).

![Tree diagram with probabilities](image)

**Figure 21. Sibling property illustration.**

Algorithm \( \Lambda \), also known as Vitter algorithm (Vitter, 1987), improves upon the FGK algorithm in several ways. Vitter proves that both the lower bound, and the upper bound, on the number of transmitted bits is up to two times better with algorithm \( \Lambda \). Vitter achieves this efficiency by improving his algorithm so the tree is in better balance.
than with the FGK algorithm. A second improvement is that when a node moves, the number of interchanges is limited to one. Vitter’s algorithm is known to create a more balanced tree than Knuth’s algorithm. If an input file can be compressed using the static Huffman algorithm down to $S$ bits and it consists of $n$ symbols, then the FGK algorithm can compress with a maximum of to $2S + n$ bits. Vitter significantly improves this. With algorithm $\Lambda$, less than $S + n$ bits will be transmitted (Vitter, 1987).

For this research the resulting algorithm can be applied equally well to both algorithms $\Lambda$ and FGK. The technique of using a frequent item identification algorithm (Metwally, Agrawal, & El Abbadi, 2005) to moderate the size of the data structures and the algorithm and its improvements can be applied to either.

The Huffman tree only approaches the true minimum bound for the entropy in the message. The true minimum bound for the entropy in each message is

$$H = - \sum_{i=1}^{n} P_i \log_2 P_i$$

where $n$ is the number of bits to be encoded and $P_i$ is the probability of the symbol in the message of length $n$. This was one of the main contributions of Shannon (1948). The number returned by this equation is a lower bound on the entropy in a given message and in general is NOT an integer. The Huffman tree is a minimum encoding, as represented by an integer number of bits. This is because each compressed token is transmitted independent of the other tokens. It is possible to encode with non-integer numbers of compressed bits using the arithmetic compression algorithm (Witten, Neal & Cleary, 1987).

In both Vitters algorithm and the FGK algorithm, a special node, the NYT node (for ‘not yet transmitted’), is part of the tree with a frequency of zero. When a new
symbol is processed in the data stream, and the symbol is not already in the tree, the NYT symbol, and then the uncompressed token immediately following, is transmitted to the receiver. Both sender and receiver then incrementally update their Huffman tree by adding the new symbol with a frequency of one. On the other hand, if the symbol is already in the tree, then the Huffman code corresponding to the position in the tree is transmitted. In this case both sender and receiver need to increment the frequency of the item in the tree.

The tree is then updated to maintain the sibling property using the FGK algorithm or algorithm Λ, since nodes were either added to the tree or item frequencies updated (Knuth, 1985) (Vitter, 1987).

In algorithm Λ each node is numbered as with the FGK algorithm. Knuth (1985) shows that a Huffman tree with \( n \) leaf nodes has \( n-1 \) internal nodes and \( 2n-1 \) total nodes. This applies to the tree as built by algorithm Λ as well. The Λ empty tree starts with the single NYT node. It has a frequency of 0 and a node number of \( 2^{n-1} \). When an existing symbol is encountered in the tree, its node frequency is incremented and the tree is checked for the sibling property. If the tree needs to be updated, then algorithm Λ is called. If the symbol is a new symbol not yet in the tree then the NYT node frequency is set to 1, the symbol is transmitted and the NYT node is set to the new symbol. A new NYT node is spawned.

As with the FGK algorithm, the nodes are numbered from left to right, and from bottom to top. Algorithm Λ uses an *implicit numbering*. With Vitter all leaves of the same weight \( w \) precede all internal nodes of the weight \( w \). The FGK algorithm did not
enforce this constraint. This constraint keeps the tree in balance better than the FGK algorithm.

The dynamic tree starts with the NYT node, and spawn from it. The root node will always have the node number \(2n-1\). Next, Vitter defines a block to be all nodes with the same frequency and uses the implicit numbering scheme.

The algorithm \(\Lambda\) (Vitter, 1987) update procedure is as follows. Its purpose is to maintain the sibling property and implicit numbering.

If the symbol received has never been seen before then the NYT node spawns a new NYT node and a leaf node as its two children. The old NYT node (which is now the parent) and the new leaf node’s frequency are both incremented. The leaf node’s identity is set to the received symbol. If the received symbol is already in the tree that node is inspected to see if it’s the highest numbered leaf node in the block. If it is not, it is shifted into the spot where it is the spot ahead of all the internal nodes in its block. Then the weight of the node is incremented. The word ‘shifted’ is important because all the internal nodes ahead of it must be shifted into the position just opened. If the node is an internal node to be incremented, then there is a different sequence. Internal nodes must be shifted into the place above all leaf nodes with a weight that is 1 higher than the current weight. All the leaf nodes then get shift to down one into the spot just open. The internal node then gets incremented. Thus, the internal node maintains a spot with all other internal nodes of the same weight.

This completes the exchange of nodes at that level in the Huffman tree. If this was the root node, then the algorithm is finished. If it is not, then the current node is set
to the parent node of the current node and this algorithm is repeated one level up the Huffman tree. The pseudocode is presented in Figure 22.

**Pseudocode Λ**

Input: Huffman Tree, pointer N to leaf to increment)

Output: Updated Huffman Tree

1. **Repeat**
2. **If** (N is a leaf node)
3. Slide N into the last (rightmost) node in its block ahead of all internal nodes with the same weight.
4. Slide all nodes into the spot open by N.
5. Increment the frequency of node N.
6. **Else** {N is an internal node}
7. Slide N into the spot to the right of all leaf nodes with a weight of 1 higher.
8. Slide all nodes into the spot open by N
9. Increment the frequency of node N
10. N ← parent of N
11. **Until** N = root
12. Increment frequency of the root

*Figure 22. Core pseudocode for Vitters algorithm Λ. Adapted from “Algorithm 673: Dynamic Huffman Coding” by J. S. Vitter, 1987, *ACM Transactions on Mathematical Software, 15*(2), 158-167.*

A simpler but less efficient method pre-calculates a set of Huffman variable size codes based on preset probabilities (Salomon, 2004). This set of Huffman codes are randomly assigned to items in the input stream. As the input stream progresses the frequency of each item is updated. The list of items is then sorted by frequency. The most frequent items are then at the top, which has the shorter preset Huffman code. This method is simple and seems straightforward to adapt.
Dynamically Decreasing a Node Weight

Knuth (1985) provides a procedure for the FGK algorithm to decrease a node's weighting and rebalancing the Huffman tree dynamically. Knuth did not provide any insight as to why a decrement of the node weights might be necessary. In the context of a data compression scheme for a database stream, perhaps the weight of nodes would be decreased because of a time sensitivity of tokens in the stream.

In any case, an algorithm that can decrease a node weighting in a dynamic Huffman tree may be important to reducing the dynamic trees size to fit into a memory-limited machine as the algorithm removes old symbols. Knuth’s algorithm to decrease a node's weight by unity proceeds similar to his FGK algorithm for increasing the weight, but in reverse. First, the node to be decreased is identified. This leaf node is exchanged with the node that is the lowest numbered node in its block. Recall that the block is all nodes that have the same weight, consisting of both leaf and internal nodes. After the exchange, the node weight is decreased by one. The algorithm then moves up one level in the Huffman tree to its parent. That node is then exchanged with the lowest numbered node in its block and then the weight is decreased by one. The algorithm will sequentially process nodes up the tree until it gets to the root.

As an example, consider the tree previously given as an example and depicted in Figure 20(b). This tree will have its weighting of the ‘e’ node decreased by one, three times. Since it has a weight of three, it is expected that the NYT node will absorb the ‘e’ node after the operation. Figure 23 depicts the tree after the ‘e’ weight is decreased by one and the tree rebalanced. Figure 24 and Figure 25 depict the tree after the ‘e’ weight is decreased by one, two more times, and the NYT node absorbs the ‘e’ node (Figure 26).
Figure 23. Decreasing the symbol "e" weight by one, to 2.

Figure 24. Decreasing a node ‘e’ weight by one, to 1.
Two frequent item counting algorithms investigated to moderate the size of the Huffman tree by identifying the frequent items are the Frequent-k and SpaceSaving algorithms. The Frequent-k algorithm (Karp, Shenker & Papadimitriou, 2003) keeps an
item list of length \( k \) where \( k \) is chosen to be less than the number of unique items in the streams alphabet. It counts frequent items in the data stream. The problem is defined as follows. Assume a data stream \( S \) contains items \( x_1, \ldots, x_N \). The number of items in the stream is \( N \). These items are drawn from a set \( I \). The frequent items are those items in \( S \) that occur more than \( \phi N \) times. \( \phi \) is the support of an item in the stream. An item must occur more than \( \phi N \) times in the stream for it to be a frequent item. An exact solution to this problem will require \( O(\min\{N, |I|\}) \) space. The Frequent-k and SpaceSaving algorithms focus on an inexact solution where the memory required is less than \( O(\min\{N, |I|\}) \) (Teubner, Muller, & Alonso, 2011).

The Frequent-k algorithm maintains a list of the \( k \) items and a counter for each item, \( t_i \), in the stream, and where \( k \) is picked to be less than \( n \). The algorithm inserts new items into the list if they are not there already, and it initializes the count to one. It does not allow the list to grow larger than \( k \). A proof exists to show that it maintains the list of the \( k \) most frequent items and their relative frequency.

Frequent-k is an “\( \epsilon \) approximate” algorithm. Cormode and Hadjieleftheriou (2008) note that according to Bose (2003) “executing this algorithm with \( k = 1/\epsilon \) ensures that the count associated with each item on termination is at most \( \epsilon n \) below the true value.” Cormode and Hadjieleftheriou provide a definition, “Given a stream \( S \) of \( n \) items, the \( \epsilon \)-approximate frequent items problem is to return a set of items \( F \) so that for all items \( i \in F, f_i \geq (\phi - \epsilon) n \), and there is no \( i \notin F \) such that \( f_i > \phi n \); \( \phi \) is the support of the item in the stream. Then \( \phi n \) is the ideal frequency of the item in the stream for it to be considered frequent. \( f_i \) is the frequency of each of the items \( i \) in the \( \epsilon \)-approximate set \( F \). \( \phi - \epsilon \), therefore, is an approximation to \( \phi \). If \( \epsilon \neq 0 \), then \( f_i \) approximates \( \phi n \), and it must
not be less than $f_i$. In other words, the $\epsilon$ approximate set should not overestimate the count.

**Frequent-k Algorithm**

1. A list, which is initialized to null, is maintained of item/count pairs.

2. The list is updated as new items arrive in the stream. There are three possibilities when an item arrives:
   a. If the item is in the list, then its count is simply incremented.
   b. If the item is not in the list, and the list length is less than the maximum list length $k$, then the new item is added to the list and its count is initialized to 1.
   c. The final possibility is that the item is not in the list and the list is full (number of items in list > $k$). In this case:
      i. all items counts in the list are decremented by one.
      ii. Items whose count reaches 0 are removed from the list.
      iii. The new item is not added to the list.


The time costs of Frequent-k are dominated by $O(1)$ dictionary operations every update, and the $O(k)$ cost of decrement all the counts in the list (Cormode & Hadjieleftheriou, 2008).

The SpaceSaving algorithm (Metwally, Agrawal, & El Abbadi, 2005) pseudocode is listed in *Figure 28*. The time costs of SpaceSaving are dominated by $O(1)$ dictionary operations every update and finding the item with minimum count $O(\log k)$ (Cormode & Hadjieleftheriou, 2008).
SpaceSaving Algorithm

1. A list, which is initialized to null, is maintained of item/count pairs.

2. When a new item arrives, there are three possibilities:
   a. If it is in the list, then its count is incremented.
   b. If the new item is not in the list, and the list is not full, then the item is added to the list and its count is initialized to 1 (as with Frequent-k).
   c. If the item is not in the list, and the list is full, then Space-Saving operates differently than Frequent-k. In this case:
      i. Space-Saving finds the item with the smallest count.
      ii. It replaces the item with the smallest count with the new item.
      iii. It increments the count by 1.


When the SpaceSaving algorithm finds a new item in the input stream, and the list is full, it does not start the new item out at a count of 1 as in the Frequent-k algorithm. Rather, it assumes that the new item might have occurred in the stream before and it just lost count of it because another item had replaced it. Thus, SpaceSaving algorithm never underestimates the count of an item. SpaceSaving has the property of maintaining an accurate count for items that appear early in the stream (Cormode & Hadjieleftheriou, 2008).

Figure 29 presents an example of the Frequent-k algorithm. Figure 30 presents an example of the SpaceSaving algorithm. In part (a) an initial string of
‘aacccbbbbddeeeeee’ is input on the stream. This fills all the available slots in the list for each algorithm. In part (b) the subsequent string ‘ffbbgg’ is input on the stream.

The first two ‘f’ character input to the Frequent-k algorithm result in count of all symbols in the list being decrement by 2. In addition, the ‘a’ and the ‘d’ symbol counts reach zero, so they are removed from the list. Next, two ‘b’ characters are input to the algorithm. This symbol is in the list, its count is incremented twice. Its count is restored a value of 4. Finally, two ‘g’ characters are input to the algorithm. ‘g’ is not in the list, but there are empty slots in the list. The ‘g’ symbol is put into the first available slot, the slot previously occupied by ‘a’.

Figure 29. Frequent-k algorithm example, $k = 5$.

The SpaceSaving algorithm proceeds differently from Frequent-k as illustrated in (b) of Figure 30. The first ‘f’ character is not in the list and the list is full. The algorithm finds the first symbol in the list with the lowest count. This is the ‘a’ character. The ‘a’ character is then replaced with the ‘f’ character and its count is incremented. The second ‘a’ character results in the count being increment one more time. Next, the two ‘b’ characters are input to the algorithm. ‘b’ is in the list, its count is incremented twice. Finally, two ‘g’ characters are input to SpaceSaving. ‘g’ is not in the list and the list is
full. SpaceSaving identifies the first item in the list with the lowest count. This time that is the ‘d’ character. The ‘d’ character is replaced with the ‘g’ character and the count is incremented twice. Once the list is full, it will always remain full with SpaceSaving. SpaceSaving simply replaces the symbol with the lowest count with the new symbol when the list is full.

![Figure 30. SpaceSaving algorithm example, k = 5.](attachment:image.png)

Another algorithm to find frequent item in a stream is Lossy Counting (Manku & Motwani, 2002). This algorithm is further optimized around the time complexity of the frequent item task; it is like Frequent-k in that it keeps a count of recent items. The SpaceSaving algorithm has the additional benefit of accurately counting items that occur early in the stream (Cormode & Hadjieleftheriou, 2008), rather than just providing identification of frequent items.
Lossy Compression

In a lossy compression scheme, the recovered data stream would not be identical to the original stream. Lossy compression is typically applied to data such as voice and video where some loss of the original fidelity can be tolerated. Several researchers have explored lossy compression applied to a database stream.

As an example of lossy compression applied to a database stream, Muthukrishnan’s (2011) recommends that a sensor database stream may be compressed using a lossy algorithm. In this research, the author looks at several data stream sources. A compressed sensing system would compress the data at the generating data source. A system that employs a lossy compression system, if the goal were to minimize compute resources rather than communications bandwidth, could employ the lossy compression hardware anywhere in the transmission channel (for instance at the receiver rather than the source). He suggests “Compressed Functional Sensing” and he writes “We need to extend compressed sensing to functional sensing, where we sense only what is appropriate to compute different functions and SQL queries (rather than simply reconstructing the signal) and furthermore, extend the theory to massively distributed and continual framework to be truly useful for new massive data applications above.” In effect, he may be suggesting that the SQL query be moved to the source to achieve compression of the data stream. Another possibility he may be suggesting, for a lossy compression of the data stream, would be to move up the concept hierarchy.

Lossy compression of XML data is proposed by M. Cannataro (Cannataro, Carelli, Pugliese, & Sacca, 2001). This may have application to a lossy compression streaming algorithm. In this application, the author envisions a sales application where
daily sales are sent to a manager for approval. If the manager were sitting at their desk with a large monitor, there may be no problem in displaying or accessing the information. On the other hand, if the manager were using their cell phone then perhaps only the daily sales total amount is presented. However, perhaps that is too little information. The phones display and network may have more capacity to present additional information. In the scenario, they envision the phone negotiating with the source an available bandwidth.

The solution the authors (Cannataro, Carelli, Pugliese, & Sacca, 2001) proposes is for the source to first negotiate a transmission and lossy compression rate. The document is then delivered at the negotiated rate. The source then identifies several dimensions of the original datacube such as item type and customer city. It then processes the hierarchy over those dimensions and some aggregate measures, such as the item quantity. It processes the datacube over those aggregate functions, over the dimensions and delivers to the destination a ‘synthetic’ datacube. The author claims that this is a lossy synthetic version of the original datacube.

Cannataro (2001) points out the various categories of data compression. For instance, lossless vs. lossy compression. These terms reference the reversibility of the compression. If the restored data is identical to the original data, the compression is lossless. Another category is on which features the data is compressed. Cannataro makes a distinction between source coding and entropy coding. Source coding refers to compression made on the semantics of the data, whereas entropy coding is made on the redundancy in the data.
One of the future directions proposed by Cannataro is to focus on the analysis of the error in this lossy scheme. Many lossy compressors have measurable errors and suitable metrics could be developed. This is an important metric that could be delivered with the compressed data. While the author explores the possibility of lossy compression on a database stream, it seems to be most applicable to data that can tolerate an inexactness in the reproduced, uncompressed, stream such as audio, video or other sensor data such as temperature.

A typical data stream processing system (Rajaraman & Ullman, 2012) may have several input streams which are asynchronous, or even have non-periodic time schedules. There may or may not be an archival storage system in any data stream processing system. In this data stream processing, although it may be archiving parts or the whole stream, it is generally not practical to answer queries over the database using the database archive. Secondly, as the author points out, there is a limited working store that may hold summaries, or parts, of the data stream. This is central support to the ‘memory limited’ premise of this research.

Rajaraman and Ullman (2012) point out typical sources of the streaming data. They point out the data may be sensor data, image data, or internet and web traffic. Sensor data might come from a temperature sensor that is coupled with a GPS unit that can read altitude or height. If the oceans were covered in sensors, one every 150 sq. mi., and each sensor generated a data point at a 10Hz rate, then 3.5 terabytes of information would be generated every day (Rajaraman & Ullman, 2012).

Web traffic is another source of Streaming data. Sites such as Google and Yahoo! generate billions of clicks per day. “A sudden increase in the click rate for a link could
indicate some news connected to that page, or it could mean the link is broken and needs to be repaired” (Rajaraman & Ullman, 2012).

The limited memory of a data stream processing system is reiterated by Marascu and Masseglia “Mining Sequential Patterns from Temporal Streaming Data” (2005). Here they note the attributes that set data stream processing apart from other database processing. For instance, new elements are generated continuously and they must be considered as fast as possible. Blocking of the data or operations is not allowed and the data may be considered only once (single-pass). Most importantly they note that memory size is “restricted.”

**Frequent Item-Set Stream Mining**

Frequent item-set stream mining is closely related to frequent item stream mining. Jin and Agrawal (2005) propose a method based on the Frequent-k algorithm (Karp, Shenker & Papadimitriou, 2003). In their Item-set mining algorithm, a Lattice of all item-sets up to some $L_k$ is maintained, where $k$ is the maximal frequent item-set. Item-sets for $k < 2$ are maintained similar to the Frequent-k algorithm. All two-item combinations of items in the stream enumerated and are added to $L_2$ using Frequent-k. The researchers invoke a routine “ReducFreq” when the array for Lattice $L_2$ is filled. ReducFreq decrements the count of all items in $L_2$. It also triggers a second stage. The second stage deals with item-sets for $k > 2$. It progresses one level at a time. For $L_3$ it enumerates all three item-set combinations in the transaction in the input stream. However, if the 2 item subsets are not contained in $L_2$, then the 3 itemset is not added to $L_3$. In this way, the *Apriori* property is exploited. This second phase continues until all
lattices are updated, up to the maximum item-set. As item-sets are added to each lattice, a count is updated, or ReducFreq is called again if the Lattice is filled.

Item-set compression poses memory challenges as well. *Item-set compression* finds *frequent combinations* of items that occur in each of the stream’s transactions. The combinatorial memory requirement growth of an item-set compression algorithm from direct application of a bottom-up item-set identification algorithm will require (Jin & Agrawal, 2005):

\[ \Omega\left(\frac{1}{\theta} \times \binom{i}{C} \right) \]

space for the lattice, where \( l \) is the length of each transaction, \( i \) is the number of potential frequent item-sets, \( \theta \) is the support threshold, \( \Omega \) is a constant, and \( C \) is the combinatorial operator. As the equation indicates, this approach is prohibitively expensive when \( l \) and \( i \) are large.

The amount of compression offered by an item-set identification and compression algorithm varies by the cardinality of the item-set and the frequency of the item-sets. Smaller item-sets that occur more often could possibly provide a higher overall compression ratio than larger items that occur less often.

The definition of the compression ratio commonly used is

\[ \frac{c}{u} \]

where \( c \) is the length of the compressed data stream and \( u \) is the length of the uncompressed data stream. An estimate on the compression ratio for a frequent item-set compression algorithm can be developed with a few assumptions about the data stream. The first is the number of item IDs are “much much” greater than the number of transaction IDs. That is, the item IDs dominate the data stream. The second is that a
frequent item-set compresses to a single ‘compression ID’ that is the same size as each of
the item IDs. Finally, it is assumed that synchronization data, such as the NYT token to
be discussed later, are a negligible part of the stream. If an item-set, \( x \), has a support of
\( supp(x) \), and the length of the item-set is \(|x|\), then the contribution of any single itemset to
the compression ratio estimate is:

\[
\frac{c}{u} = u - supp(x) \cdot u \cdot \frac{|x| - 1}{u} = 1 - supp(x) \cdot (|x| - 1)
\]

Thus, the contribution that an itemset \( x \) makes to the overall compression is proportional to
\((|x| - 1) \cdot supp(x)\)

Finding the frequent item-sets that minimize the compression based on
identification of frequent item-sets may be an area for future research.

Several algorithms for frequent item-set identification on a static database exist
(Agrawal & Srikant, 1994; Savasere, Omiecinski, & Navathe, 1995). A common
requirement for these algorithms is that they require the database to reside in memory, or,
the database to stream into memory once, or several times.

Several algorithms for frequent item-set identification provide for some sort of
compression of the in-memory database structure (Bodon & Rónyai, 2003; Han, Pei &
Yin, 2000; Shenoy, Haritsa & Sudarshan, 2000; Zaki & Gouda, 2003; Zaki,
Parthasarathy, Oghara & Li, 1997). These are examined next.

**Transaction Database Compression**

Several of the popular frequent item set mining algorithms identify compression
as a technique to help advance the frequent item set mining process.
Han, Pei, and Yin (2000) proposed a form of compression. Their FP-growth algorithm builds a tree in memory from the transaction database. The tree represents the database in compressed format. Branches of the tree point from item-sets with common prefixes to their super-sets. FP-growth is not bottom up as with Apriori. FP-growth trees can grow to large sizes. Although FP-growth can be very fast, it is considered impractical for very large databases (Zhang, Zhang, Jin, & Bakos, 2013).

Eclat (Zaki, Parthasarathy, Ogihara & Li, 1997) provides a depth first traversal for frequent item sets. In this algorithm intersections of known frequent item-sets identify new frequent item sets. This algorithm uses the vertical representation of the transaction database as depicted in Table 7 (c). The vertical format provides a type of compression in the Eclat algorithm. Assume that the vertical transaction database is stored in main memory. The items that do not meet the minimum support count do not need to be in memory. “The main benefit of vertical tid is that it allows intersect, thereby enabling us to ignore all infrequent items/itemsets” (Ashrafi, Taniar, & Smith, 2007). These authors note that for large databases Eclat can run out of a limited memory.
Table 7
*Vertical Versus Horizontal Formats*

(a) Horizontal format and (b) horizontal bitmap format

<table>
<thead>
<tr>
<th>Transaction ID</th>
<th>Item</th>
<th>Transaction ID</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>A, B</td>
<td>100</td>
<td>A</td>
</tr>
<tr>
<td>200</td>
<td>D, E</td>
<td>200</td>
<td>B</td>
</tr>
<tr>
<td>300</td>
<td>A, C</td>
<td>300</td>
<td>C</td>
</tr>
<tr>
<td>400</td>
<td>A, C</td>
<td>400</td>
<td>D</td>
</tr>
<tr>
<td>500</td>
<td>C</td>
<td>500</td>
<td>E</td>
</tr>
<tr>
<td>600</td>
<td>D, E</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Horizontal bitmap format

<table>
<thead>
<tr>
<th>Transaction ID</th>
<th>ID</th>
<th>Item</th>
<th>Item</th>
<th>Item</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>300</td>
<td>1</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>1</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>1</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>600</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(c) Vertical tidset format and (d) Vertical bitmap format

<table>
<thead>
<tr>
<th>Item</th>
<th>Transaction ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
</tr>
<tr>
<td>B</td>
<td>200</td>
</tr>
<tr>
<td>C</td>
<td>300</td>
</tr>
<tr>
<td>D</td>
<td>400</td>
</tr>
<tr>
<td>E</td>
<td>500</td>
</tr>
<tr>
<td>A</td>
<td>600</td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Item</th>
<th>Transaction ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100 200 300 400 500 600</td>
</tr>
<tr>
<td>B</td>
<td>100 200 300 400 500 600</td>
</tr>
<tr>
<td>C</td>
<td>100 200 300 400 500 600</td>
</tr>
<tr>
<td>D</td>
<td>100 200 300 400 500 600</td>
</tr>
<tr>
<td>E</td>
<td>100 200 300 400 500 600</td>
</tr>
</tbody>
</table>

Eclat using diffsets was proposed (Zaki & Gouda, 2003). Diffsets was demonstrated to drastically cut down the amount of memory necessary to hold intermediate results and speed up processing. Zaki and Gouda (2003) present in their paper a comparison of the performance and compression offered by Apriori, Viper, FP-growth, Eclat and Eclat using diffsets. The Eclat using diffsets algorithm identifies memory as a limited resource. Diffsets does not compress the transaction database, they compress candidate item-sets.

Bodon and Rónyai (2003) compare a Trei data structure to hash trees to store the candidate item-sets. Their compression method provides significant savings of memory and processing speed over hash trees. They use Apriori, bottom up processing to build the max itemsets. The Treis data structure does not compress the transaction database.
VIPER (Shenoy, Haritsa & Sudarshan, 2000) is a vertical mining algorithm that stores data in compressed bit vectors called ‘snakes’. This algorithm uses the Vertical bit vector format of the transaction database. In the compression scheme runs of 0 bits and runs of 1 bits are encoded using an encoding based on the Golomb encoding scheme. The compression process to create the snake they called ‘skinning.’ Runs of 1 and 0 bits are divided into groups of size $W_1$ and $W_0$ respectively. Each group is represented by a bit vector with single weight bit set to 1. The authors claim that this approach is superior than a simple RLE encoding because in a transaction database there will be runs of many 0 bits and only a few 1 bits.

In their scheme, they believe they have identified additional redundancies in the vertical bit vector format that they can remove and achieve a higher compression ratio. For a full description of the Viper skinning compression scheme to create snakes, please see Shenoy, Haritsa, and Sudarshan. Compressed Bitmaps are proposed by Garcia-Molina, Ullman and Windom (2008) for the bitmap database formats as depicted in Table 2 (b) and Table 2 (d) although Garcia and Molina are not specifically addressing transaction databases in their proposal. In this book the authors note that if the file has $n$ records, and each record consists of a field of items with $m$ items, then the file consists of a bit vector of $mn$ bits. The book notes that if $m$ is large, and the number of items in each transaction is small, then the probability of a 1 bit is $\approx 1/m$. The research goes on to describe a run length encoding (RLE) scheme.

An important note made by the authors is that the RLE scheme they propose is not an optimal encoding scheme for long runs of 0, although they characterize it as a simple encoding scheme. They note that other encoding schemes can improve the
compression ratio by a factor of 2. An optimal encoding scheme presented in this research proposal is the Golomb encoding scheme.

Similar to Golomb encoding, Compressed Bitmaps breaks the encoded run into two parts. The first part is a unary prefix code. The second part is the binary coded number of 0 bits in the run. This second field is a variable length field. A unary code is required for the first part because unary is a prefix code. The unary code is chosen to represent the number of bits in the second binary part. As an example, suppose the record 000001101 is to be encoded. The first task is to encode a string of five 0’s followed by a 1. In binary this is coded as 101. Three bits are required to specify the number of 0 bits. Thus, the unary part will be 110. This is a 3 in the unary system. The coding will start out as 110101. Next, a string of no zero bits will be encoded. The binary part is encoded as 0. One bit is required so the unary part is encoded as 0. The encoded string now becomes 11010100. Finally, a string of 1 zero bits will be encoded. The binary part is encoded as 1. One bit is required for this encoding; thus the unary part is encoded as 0. The final encoded string becomes 1101010001.

It is important to note that in this example the string to encode ended in a 1. Typically, in a vertical bitmap encoded transaction database with a low support count, the record will end in a string of 0’s since the probability of a 1 bit is low. Similar to the Golomb encoding scheme prototyping in the initial investigation, the authors point out that the trailing string of 0’s need not be encoded. “Since we presumably know the number of records in the file, the additional 0’s can be added.”

Golomb compression gains its edge in compression because it ‘tunes’ the length of the binary part of the code, and the prefix code, to the length of the median number of
0 bits. Their research shows how to perform bitwise AND and OR operations on the RLE encoded vectors, and how to manage the indexes of the bitmaps. They demonstrate that decompression of the compressed bit vector stored database is not necessary for many common database operations. Ashrafi, Taniar and Smith (2007) propose an interesting compression technique they call “diff-bits” for the vertical tidset format depicted in Table 2 (b). They note their format offers good compression when the support is below 3.33%. As an example, consider the database in Table 8.

Table 8
*Sample Database for Diff-Bits Algorithm*

<table>
<thead>
<tr>
<th>Item</th>
<th>Transaction IDs (TIDs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bread</td>
<td>1, 2, 10, 20</td>
</tr>
<tr>
<td>Butter</td>
<td>1,3,10,40</td>
</tr>
<tr>
<td>Beer</td>
<td>1,10</td>
</tr>
</tbody>
</table>

The first step in compressing this database is to compute the difference in the TIDs. The authors offer that the reason for using the difference in the TIDs, rather than the TIDs themselves, is that there may be, on average, several transactions per item. The size of the required memory will divide down by this average size. For example, assume that standard 32-bit TIDs are used. This is a typical word size in a computer. Assume that the average number of items per transaction is 20. If instead the difference in the tids is stored rather than the tids themselves, then each tid difference only requires a 20-bit word on average. On average the tid values would be spread across all 32-bit values. Tid differences for the example are show in Table 9.

Table 9
*Calculation of Transactional ID (TID) Differences*
The next step in compressing the vertical transaction database is to calculate the binary compression code. The binary compression code is in two parts. The first part is a prefix code and the second part is the binary representation of the tid differences.

For the prefix code, the authors suggest a 5-bit fixed width integer. They suggest 5-bits because a 5-bit code can encode a number from 0 to 31, which is the maximum number of bits in a tid and the standard word size for many computers. In the example for Bread, the third tid difference is an 8. This can be encoded as 1000 in binary. Because the proposed compression scheme uses a variable number of bits to encode the tid difference, the prefix needs to indicate the number of bit used. Encoding 1000 requires 4-bits. Four bits are indicated as a 5-bit prefix with the binary number 00100.

Next the authors point out that the tid difference will always start in a 1. This is because the leading 0 padding bits are not required. Therefore, the starting 1 need not be encoded in the tid difference. A tid difference of 8 is encoded as 00100|000. The vertical bar indicates the partition between the prefix and the encoded tid difference. Table 10 summarizes the calculated bit vectors for the diff-bits compressions scheme.

<table>
<thead>
<tr>
<th>Item</th>
<th>TID differences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bread</td>
<td>1, 1, 8, 10</td>
</tr>
<tr>
<td>Butter</td>
<td>1, 2, 7, 30</td>
</tr>
<tr>
<td>Beer</td>
<td>1, 9</td>
</tr>
</tbody>
</table>

Table 10

*Calculation of Diff-Bits in Bit Vector*

<table>
<thead>
<tr>
<th>Item</th>
<th>Diff-bits compressed bit vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bread</td>
<td>00001 00001 00100</td>
</tr>
<tr>
<td>Butter</td>
<td>00001 00010</td>
</tr>
<tr>
<td>Beer</td>
<td>00001 00100</td>
</tr>
</tbody>
</table>
Other Compression Algorithms

Modern compression software for the PC is based on dynamic dictionary compression algorithms (Nelson & Gailly, 1996). Dictionary based techniques differ from the Huffman/RLE based techniques in that these do not require a statistical model of the data. Dictionary based techniques identify and compress strings of data. The simplest type of dictionary compression may be that which uses a static dictionary. This approach creates a fixed dictionary prior to encoding. The encoding process can only compress strings in that are contained in the dictionary. The receiver, or decoder, must also have the fixed dictionary. An algorithm might proceed by looking up items to compress in the dictionary and returning their index in the table. The index would be shorter than the string and would be the transmitted or stored value. If for instance the table were to hold 2048 entries, then long strings might be compressed to 11-bit values. In the case of a transaction database the item ID might be compressed using this method. This table-based substitution is the normal consequence of the decomposition of a database schema.

Static dictionary techniques based on Diagram Coding (Gage, 1994) have been proposed as a compression technique. In diagram encoding the source alphabet is encoded using the ‘standard’ encoding symbols, and frequently used pairs of the source alphabet are also encoded using any remaining symbols. For instance, assume that the standard symbol size is 10 bits, thus 1024 symbols are available. An encoding scheme might use the first 600 symbols to encode item keys in a small grocery store. The remaining 424 symbols would be used for frequent item symbol pairs.
In 1977 and 1978 two papers were written (LZ77 and LZ78) by Jacob Ziv and Abraham Lempel on which modern dynamic dictionary based compression programs are built. “Dictionary-based compression techniques are presently the most popular forms of compression in the lossless arena. Almost without exception, these techniques can trace their origins back to the original work published by Ziv and Lempel in 1977 and 1978” (Nelson & Gailly, 1996).

The Lempel/Ziv 1977 algorithm achieves compression by using a sliding window on the data stream to compress. There are two parts to the sliding window. The first part is a buffer of recent data. The second part is a short look-ahead buffer. New data is shifted into the look-ahead buffer and then shifts into the recent data buffer. The algorithm looks for matches of data from the look-ahead buffer to that in the recent data buffer. If it finds a match, a token is emitted that contains the offset in the recent data buffer, the length of the match, and the next character in the look-ahead buffer that does not match. If there is not a match, the token emitted is two zeros followed by the non-matching character. The algorithm can then transmit the message one token at a time if there are no matching characters. This algorithm is suited to compressing data that contains strings of similar data. For instance, LZ77 will compress a long string of ASCII space characters as a single token. The algorithm can compress strings that exceed the length of the look-ahead buffer. For instance, suppose there is a long string of space characters. A space character in the look ahead buffer will match on a space character in the data buffer. As that character is shifted into the data buffer it will now match with the next space to be shifted into the look-ahead buffer. In this manner long strings can be compressed as long as the repeating character part of the string is shorter than the data
buffer. The main challenge in this algorithm is to program a string-matching algorithm that looks for variable length strings starting at each index in the recent data buffer (Ziv & Lempel, 1977). LZ77 is a greedy algorithm. Once it finds a match, it does not look any farther in the buffer for a better match. Because LZ77 only compresses on strings contained in its buffer, it compresses better on data that favors repetitiveness in its recent data. So, for instance, a dictionary or address book might compress well. LZ77 might do well on the last name in the address book, whereas the street names or first name might not compress as well.

Another algorithm, LZSS, based on LZ77, improves compression (Storer & Szymanski, 1982). It keeps another data structure, a binary tree, to maintain the list of recent matches that shift out of the data buffer. LZSS also improves on the emitted tokens compression when there is not a match. LZSS uses a single bit to indicate whether the data byte is a token or a character. Thus, when a match does not occur, LZSS can more efficiently encode that data.

Both LZ77 and LZSS cannot match recent data on data that shifts out of the recent data buffer (although LZSS keeps a list of previous matches). LZ78 solves this issue (Ziv & Lempel, 1978) by removing the sliding window. Instead, it keeps a dictionary of matching strings. The dictionary structure is a multi-way tree. Although traversing such a tree is easy to find matching leaves, each node can have up to 256 children (if an 8-bit alphabet is used.) Thus, to minimize memory requirement in the data structure, each node must keep a list of pointers to its children rather than a static array structure. LZ78 starts the tree off with a single pointer with the NULL character. When the first character is read in, it matches its previous character, which is NULL, on the
single NULL node in the tree. The algorithm then outputs its first token. The token contains the index of the matched node (in this case 0), and the matched character. It also adds this matched string to the tree. Suppose now the second character is the same as the first. The algorithm would now match with node 1 in the tree. The algorithm would read one more character from the input. This character would not match node 1. The output would then be a token, which contained the node number, 1, and the third character. A new node would be created in the tree, which would contain the three characters in the input stream recently read. LZ78 continues in this way.

LZ78 (Ziv & Lempel, 1978), like LZ77 (Ziv & Lempel, 1977) and LZSS (Storer & Szymanski, 1982), compresses continuous strings of input characters. LZ78 differs from LZ77 in that it will match on strings farther back than LZ77 can with its limited data buffer. This may or may not be an advantage depending on the nature of the data to compress. Two issues with the LZ78 algorithm are that the decoder needs to maintain the same tree structure as the encoder, and the tree can grow to fill up available memory quickly. Nelson & Gailly (1996) discuss these issues in their book.

Initial Investigation (Prior Research Work)

Overview

Prior research provided encouraging results for the compression of a static database. In the research, a database compression harness was written to compress and compare several benchmark databases for machine learning. The code was written in C# in the Visual Studio IDE. It was based on the original algorithm by Huffman (1952) and Golomb (1966). The three algorithms compared were:

• A static (two pass) Huffman compression scheme.
• An RLE compression based on ideal Golomb prefix codes.

• An RLE compression based on ideal Golomb prefix codes, with items sorted by frequency.

The asymptotic time to compress a database is determined for each compression type. The research develops an algorithm for RLE encoding, based on Golomb prefix codes, to exploit the horizontal bit vector, transaction database, structure. Note that it would be straightforward to apply results of the RLE compression algorithms to streaming databases. As noted in the literature review the RLE compression using Golomb prefix codes is a two-pass algorithm, but a good approximation can be made of the \( m \) value and compression achieved in a single pass.

*Compression Algorithms Used to Achieve Results in the Initial Study*

Figure 31 lists the pseudocode for the Huffman Compression Algorithm used in the compression harness. The compression harness also implements two other compression schemes based on a Golomb RLE compression described later in this paper. The Huffman compression algorithm reads the complete database twice. The first pass tabulates the frequencies of each database item. In the second pass each item is re-encoded with its new optimal minimum entropy ID. A dictionary data structure performs a fast lookup of items codes and their calculated Huffman code and code length.

The first step in calculating the Huffman codes is to build the Huffman tree. The tree is a set of nodes whose leaf nodes are each item in the original database. The software builds a dictionary by searching the tree for each item, then traversing the tree back to the root. The path back to the root is the Huffman code and Huffman code length. Left branches are arbitrarily set to be a ‘1’ bit, and right branches a ‘0’.
The list, $T$, is initialized to a list of nodes. Initially there are $r$ nodes, one for each symbol. Each node is a 5-tuple. The 5-tuple is a structure containing a symbol (if the node is a leaf node), a count, two links to its children, and a link back up to its parent. If the node is an internal node, the count is the sum of the count of its two children. If the node is a leaf node, then the count is the number of times the symbol occurs in the file. Lines 1 through 12 of the pseudocode calculate the item frequencies and set up the initial list of nodes. It is asymptotic to $O(n)$ time, where $n$ is the total number of items sold in the database.
Figure 31. Pseudocode for static Huffman compression.
In Lines 13 through 22 the nodes are built into a binary tree using Huffman’s approach. Line 15, assuming a binary sort, is $O(\log_2 r)$ time, where $r$ is the total number of different items in the database. The item IDs are represented in a binary format. The list of nodes needs to be sorted $r$ times. Thus, building the Huffman tree occurs in $O(r \times \log_2 r)$ asymptotic time. To write the compression codes, lines 37 to 44, to the output requires $O(n)$ time, for all $n$ items. Line 14 creates a new node and sets it to null. This will be the parent of the two nodes, $a$ and $b$, with the smallest count in $T$.

Lines 24 through 36 create a dictionary of the uncompressed symbol, and compressed symbol codes. Finally, in lines 38 through 45, the input file is read a second time to compress the ID’s.

The canonical form of the Huffman code is not determined in this software harness. Use of the canonical form would not affect the compression ratios. The canonical Huffman codes will have the same length as the codes calculated and provide the same compression ratios. Calculation of the canonical codes would occur in $O(r)$ time. A canonical Huffman code will be relevant to the hardware item decoding in transaction support count circuitry implemented in reconfigurable hardware.

Figure 32 presents the pseudocode to RLE encode a transaction database. The resulting output file will be a bit map RLE encoded file using Golomb ideal prefix codes. Calculation of the prefix codes is straightforward using algorithms in Salomon (2007). Although the pseudocode shows a single pass over the database, an extra pass over the file before processing was necessary in the compression harness. This extra pass served three purposes. It was used to collect statistics and compute the optimal “$M$” value. It was also necessary to sort each line of the transaction database. Several of the databases
had lines not in lexicographic order. Finally, and most important, it was necessary to
renumber the items to remove non-existent item ID numbers. Non-existent item ID
numbers would lower the overall Golomb compression ratio score by adding unnecessary
0’s to the strings.

Figure 32. Pseudocode to write Golomb RLE database.

This form of RLE compression is very good at compressing long runs of 0 bits.
This corresponds to only a few of the available items occurring in each transaction. If
long runs of 0 bits can be followed by long runs of 1 bits then other compression schemes
should be considered.

An important optimization occurs in this pseudocode that gives an edge to the
RLE compression scheme. The trailing string of 0’s was not encoded. This is because a
carriage return (or other special character) delimits each transaction in the file. When the
compressed database is read later to create the specialized hardware, the synthesizer need
not create registers for the last run of 0’s. A single special bit can encode the transaction
delimiter outside of the registers used to hold the compressed IDs. A similar approach is
used by Compressed Bitmaps (Garcia-Molina, Ullman and Windom, 2008). Here, the
authors note that each record in the horizontal bitmap format has a fixed length, thus the
length of the last run of 0’s can be inferred.

Compression using a transaction delimiter was included in the harness and used to
compare compression schemes. The transaction delimiter was implemented as a ‘special’
item with the largest item ID. Each transaction ended in this special ID. These results
are not presented in this research. The compression ratios obtained were like those
obtained here, but were a few percent less in each case.

A second optimization to the RLE compression was coded in the harness. This
optimization is presented as a separate result herein. This optimization provides a few
percent gain to the ‘standard’ Golomb RLE compression as evidenced in Table 11. Since
the last run of 0’s need not be coded and registers created, it is of benefit to make this last
run of 0’s as long as possible. This can be achieved by making a “1” in the last run of 0’s
less probable. The software harness renumbers the items such that low items numbers are
frequent items in the database, and the higher numbered items are the less frequent items.
This requires an initial pass through the database similar to the Huffman compress to
determine the item frequencies and sort the items by frequency.
Table 11
Comparison of Compression Ratio (c/u) Results from prior research

<table>
<thead>
<tr>
<th>Database</th>
<th>Static Huffman compression ratio (%)</th>
<th>Golomb compression ratio (%)</th>
<th>Optimized Golomb compression ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accidents</td>
<td>72.4</td>
<td>54</td>
<td>44</td>
</tr>
<tr>
<td>BMS1</td>
<td>72.5</td>
<td>93</td>
<td>90</td>
</tr>
<tr>
<td>Kosarak</td>
<td>61.0</td>
<td>77</td>
<td>76</td>
</tr>
<tr>
<td>Retail</td>
<td>72.0</td>
<td>77</td>
<td>77</td>
</tr>
<tr>
<td>T10I4D100K</td>
<td>88.0</td>
<td>78</td>
<td>74</td>
</tr>
<tr>
<td>T40I10D100K</td>
<td>92.0</td>
<td>60</td>
<td>58</td>
</tr>
<tr>
<td>BMS-POS</td>
<td>64.1</td>
<td>75</td>
<td>74</td>
</tr>
<tr>
<td>BMS-WebView2</td>
<td>78.0</td>
<td>85</td>
<td>83</td>
</tr>
</tbody>
</table>

The asymptotic time to RLE compress the transaction database using ideal Golomb prefix codes is $O(n)$ time, where $n$ is the length of the binary string to compress. Calculation of the prefix codes is a straightforward calculation from each of the input items in each transaction. This assumes an approximate value of $m$ is sufficient, or an exact $m$ value is available before encoding, where $m$ is the parameter as required by the Golomb compression algorithm to determine the mean run length. An exact value of $m$ can be calculated in $O(n)$ time. This is because a second, initial, pass over the data will be required to calculate the probabilities.

Optimization of the Golomb compression using this scheme requires an initial pass through the database, similar to Huffman, to determine item frequencies. This extra pass occurs in $O(n)$ time. Sorting of the list of items occurs in $O(\log_2 r)$ time. The $O(n)$ term will dominate since the items are being streamed from secondary storage and $n$ will always be larger than $r$. The asymptotic time comparisons are listed in Table 12.
Table 12
Comparison of Asymptotic Encoding Time for Compression Schemes

<table>
<thead>
<tr>
<th></th>
<th>Static Huffman compression</th>
<th>Golomb RLE compression</th>
<th>Golomb RLE compression with optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to prepare symbols</td>
<td>(O(n)) to read database</td>
<td>(O(1))</td>
<td>(O(n)) to read database</td>
</tr>
<tr>
<td></td>
<td>(O(r^*\log_2 r)) to build tree</td>
<td></td>
<td>(O(\log_2 r)) to sort item list</td>
</tr>
<tr>
<td></td>
<td>(O(r^*\log_2 r)) to traverse tree</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time to encode database</td>
<td>(O(n)) to encode</td>
<td>(O(n)) to encode</td>
<td>(O(n)) to encode</td>
</tr>
</tbody>
</table>

Note. \(n\) is the total number of items in the database. \(r\) is the number of different items in the database.

Conclusion From the prior research

Huffman Compression (Huffman, 1952) provides a maximal compression when there is a large variation in the frequency of items. Huffman will compress the worst when all items are of similar probability, that is, the symbols are all of minimum entropy because they are all random. In these cases, RLE compression (Golomb, 1966) will be a better choice. A Huffman algorithm also will not compress a two-symbol alphabet. RLE compression may be able to be used for these cases. Golomb codes are a minimum entropy prefix code (Golomb, 1966) for RLE compression. The Golomb compression ratio is related to the “\(m\)” value (Golomb, 1966). Databases that will compress well with this RLE compression are those where each transaction includes, on average, a few of many items. This corresponds to a large \(m\) value and long runs of 0 bits. Golomb compression, using the algorithm in this paper, will compress well when low numbered items are more frequent in the database. This is because the algorithm in this paper does not encode the last run of 0’s and uses a transaction terminator instead. The RLE compression with optimization (the second form of Golomb prefix RLE compression
implemented in the compression harness) will favor those databases where a few items occur more frequently, not necessarily low numbered items.

The exact compression ratio obtained using each compression scheme is dependent on the real-world probabilities of items in the database. Synthetic data did not compress well using the Huffman Compression.
Chapter 3

Methodology

Approach

The first part of this research is investigatory. In the first part, algorithms will be prototyped and verified to be correct. Steven M. Ross and Gary R. Morrison outline the essence of the experimental research method (1996). Here they identify the “true experiment” as maximizing the validity of the experiment. The scientific method is a logical method of posing a question and finding an answer. An abbreviated version of this method (Dodig-Crnkovic, G., 2002) identifies several steps to answering the question:

1. Posing the question
2. Formulate a hypothesis and a possible answer.
3. Make predictions about the outcome
4. Test the hypothesis. If the results do not match the predicted outcome, then repeat tests 2, 3 and 4 until agreement occurs.
5. Once there is agreement between the hypothesis and the results then the hypothesis becomes a theory. The theory provides a set of rules that define a new class of phenomena or a new concept.

Discussion of the Proposed Memory Limited Dynamic Huffman Algorithm

In the fourth step of this series, a modified FGK (Knuth, 1985) algorithm will be developed to work in a memory limited machine using a frequent item identification algorithm (Metwally. Agrawal, & El Abbadi, 2005). The modification should be straightforward. The FGK algorithm, modified to work on memory limited machines,
will need to keep track of the number of nodes in the dynamic Huffman tree. When the number of nodes exceeds the preselected value, $k$, a new node will not be added to the tree. In this case, the algorithm will search the tree for the node with the lowest frequency. When that node is identified, then its symbol will be replaced with the new symbol found in the input stream. The weight of this symbol is also incremented. At this point in the algorithm, the code for the Not Yet Transmitted (NYT) symbol is emitted in the output stream. The path from the leaf node to the root of the tree determines the compressed code. Following the NYT symbol, the uncompressed symbol is emitted in the output stream. Next, the tree will need to be rebalanced. Anytime the frequency of a node is changed, the tree needs rebalancing. The node rebalancing subroutine will process nodes up the Huffman tree and exchange the leaf node and its subtree, or its ancestor’s node and its subtree, with the node that has the highest count in its block. When a node is exchanged with the highest numbered node in its block, this is the time the nodes frequency is incremented. The node frequency is always incremented, whether the symbol was found in the tree, it was a new symbol, or it is replacing an existing symbol because the tree is full.

There is one algorithm detail that is required to keep the decoder in synchronism with the encoder. When a new symbol is input to the compressor, and the compressor tree is full, the new symbol is required to be transmitted to the decoder so both sides (compressor and decoder) can keep identical trees. In this case, the frequent item identification algorithm by Metwally, Agrawal, and El Abbadi (2005) dictates that the new symbol replaces the symbol in the tree with the lowest frequency. However, the NYT symbol will need to be transmitted to the receiver along with the uncompressed
symbol on the input, not the replaced symbols code. Additionally, in this case, the NYT node will not be split and a new node will not be created.

*Figure 33* presents the pseudocode for the Memory Limited Dynamic Huffman algorithm. Inputs to the pseudocode are the Huffman tree and the new symbol, s, from the input stream. Outputs are the compressed output stream, and the updated Huffman tree.

When a symbol arrives in the input stream, the algorithm needs to locate it in the Huffman tree. There are three possibilities. The first possibility is the symbol already exists in the Huffman tree. In this case the compressed code is determined from the path in the Huffman tree from the symbols node to the root node. The weight of this symbol is then incremented.

The next possibility is that symbol does not exist in the Huffman tree, but the number of nodes is less than the maximum allowed number of nodes. It can be proven that if the max allowable number of symbols or leaf nodes in the tree is N, then the maximum nodes (including internal nodes) in the binary tree will be N*2^1. If the Huffman tree is not full, then the algorithm proceeds to grow the tree. First the NYT node is split. A parent is created with two new children, the NYT node and a new leaf node that contains the input symbol, s. The output stream contains the compressed code for the NYT node (determined from the path of the NYT node to the root), and the uncompressed symbol, s. The weight of the new leaf node is set to 0. This will be incremented to 1 later in the call to the tree rebalancing routine.

The third and last possibility is that the symbol, s, does not exist in the Huffman tree, and the number of nodes in the Huffman tree is larger than the maximum allowed
In this case the Huffman tree, $H$, is searched for the leaf node with the smallest weight. The symbol in this leaf node is replaced with the symbol, $s$. Later in the call to the tree rebalancing, its weight will be incremented and the tree rebalanced. Finally, the compressed code for the NYT symbol, and the uncompressed symbol, $s$, is output.

---

**Memory limited dynamic Huffman algorithm**

Input: Huffman Tree $H$, Stream to process $S$.
Output: Updated Huffman Tree $H$, compression stream

1. **For each** $s$ **in** $S$
2. **If** $s$ **є** $H$
3. Current Node $\leftarrow$ Find node with $s$ in $H$
4. **Output** Compression Code  //Path from Current Node to Root
5. **Else**
6. **If** $|H| >$ MaxNodes
7. Current Node $\leftarrow$ Find node min weight symbol in $H$
8. Current_Node Symbol $\leftarrow$ s.
9. **Else**
10. Split NYT node into two nodes with new parent.
11. Assign one of the new nodes symbol $s$ with weight 0
12. Current Node Symbol $\leftarrow$ NYT node
13. **Output** uncompressed symbol, $s$
14. **Output** Compression Code  //Path from Current Node to Root
15. **Call** Rebalance Tree ($H$, Current_Node)  // Pseudocode from Figure 6
16. **Return** $H$

---

*Figure 33.* Memory limited dynamic Huffman algorithm pseudocode.

The memory limited dynamic Huffman algorithm is diagrammatically illustrated in *Figure 34* to *Figure 36*. Note that *Figure 34* mirrors the “backward adaptation model” (as previously illustrated in *Figure 7*).
Figure 34. Overview of the memory limited dynamic Huffman algorithm.

Figure 35 illustrates the three possibilities when the memory limited dynamic Huffman tree is updated with a symbol for encoding. The three possibilities are (a) the symbol is already in the tree, (b) the symbol is not in the tree and needs to be added to the tree, or (c) the symbol is not in the tree, the tree is full and an old symbol must be swapped with the new symbol. Figure 36 illustrates rebalancing the tree. In this example the leaf node with symbol ‘n’ is swapped with the highest numbered node in its group, the ‘e’. After the swap the ‘n’ node’s weight is incremented and processing continues with its parent.
**Update Tree with new Symbol – 3 Choices**

(a) Input symbol already in tree:
Increment weight, then Rebalance tree

(b) Input symbol not in tree, and tree not full:
Split NYT node and add new node.

(c) Input symbol not in tree, and tree full:
Find symbol node with minimum weight and replace symbol, then increment weight and rebalance tree.

*Figure 35.* Memory limited dynamic Huffman algorithm cases.
The pseudocode in Figure 37 builds upon the pseudocode provided by Knuth (1985) to implement the memory limited dynamic Huffman compression. Knuth’s pseudocode was presented in Figure 10 through Figure 15. The pseudocode of Figure 37 replaces that in Figure 15.

The pseudocode by Knuth implements a simple hash lookup to find the node in the tree that corresponds a symbol. The “A” array provide the hash function. Its lookup is based on the ASCII character code of the symbol being processed. A more robust hash function is assumed by the insert, delete, length and try/search methods in lines 3, 7, 12, 15 and 16. Typically, the hash table will provide constant time, $O(1)$ searching of the table (Cormen, Leiserson, Rivest, & Stein, 2009). It is necessary to use a more robust hash table function such as this, rather than the statically allocated hash table as proposed by Knuth. A statically allocated hash table that would hold all the possible input symbols would defeat the purpose of the memory limited function. A complete description of the new encode procedure follows. The new Hashtable is named HASHTABLE_A. It is assumed that there are four methods on this object. The TrySearch method looks up a
key in the dictionary. If the key is not contained in the dictionary, the `exists` variable is set to false. If the key does exist in the dictionary, the `exists` variable is set to true and the value is returned. The Delete method deletes a key/value pair from the dictionary, the Insert method inserts a key/value pair, and the Length method returns the number of items in the dictionary.

```plaintext
1. procedure encodeMemLimited (integer key);
2. begin integer i, j, q, t; boolean exists;
3. exists = HASHTABLE_A.TrySearch(key, value);
4. i ← 0;
5. if exists then q ← A[value]
6. else comment encode zero weight;
7. if HASHTABLE_A.Length < MAXNODES then
8.     q ← A[PSEUDO_SYM]; PSEUDO_SYM = PSEUDO_SYM + 1;
9.     if q < 2 × R then t ← E + 1 else q ← q - R; t ← E fi;
10.    for j ← 1 to t do i ← i + 1; S[i] ← key mod 2; key ← key div 2 od;
11.    q ← A[0] comment point to new node;
12.    HASHTABLE_A.Insert(key, q)
13. else
14.     q ← B[H]; comment point to the node with least weight;
15.    HASHTABLE_A.Delete(A[q]) comment delete this key in hash table;
16.    HASHTABLE_A.Insert(key, q) comment create new key, value fi fi;
17. while q < Z do
18.     i ← i + 1; S[i] ← (q + 1) mod 2;
19.     q ← P[(q + 1) div 2] od;
20. while i > 0 do transmit (S[i]); i ← i - 1 od;
21. end;
```

Figure 37. Original algorithm FGK modified to be memory limited. Adapted from “Dynamic Huffman Coding” by D. E. Knuth, 1985, Journal of Algorithms, 6(2), 163-180.

First the encode procedure looks to see if the symbol exists in the hash table. If it does, then the node that corresponds to that symbol is identified and the encode procedure proceeds as before. If the symbol does not exist in the hash table, then there are either one of two possibilities from here. The first is the number of symbols in the
table exceeds a constant. That constant is MAXNODES, the maximum number of symbols allowed in the hash table and tree. If the number of nodes in the hash table is less than the maximum number allowed, then the symbol is added to the hash table and a new node is created in the tree for it. The new symbol is a ‘pseudo’ symbol. It is created with the NEXT_SYMBOL global variable. The new hash table then simply maps symbols on the input to the new ‘pseudo’ symbols created. On the other hand, if the maximum number of nodes allowed has been reached, then the node with the least weight is identified in line 13. Its symbol and ‘pseudo symbol’ pair is deleted from the hash table, and a new key, value pair is created in the hash table with the new symbol.

As an example of the memory limited dynamic Huffman algorithm follows. Assume that in the example given in Figure 16 the maximum number of nodes is set to 9. The algorithm will proceed as before up to Figure 19. Figure 19 is repeated here as Figure 38. At this point the next character that appears in the input stream is the ‘r’ symbol. This is a new symbol and is not contained in the Huffman tree. The Huffman tree contains 9 nodes, the maximum number of nodes for this example. The algorithm then searches for the leaf node with the smallest weight. In this case that will be either the node with the ‘i’ or the ‘g’ symbol. Both have a weight of 1. If the algorithm searches the tree down left branches first the ‘i’ node will be selected. The tree will be rebalanced as shown in Figure 39.

Cumulative input string: engineer

Cumulative output stream: e 0n 00g 100i 11 10 10 1000r
Figure 38. Dynamic Huffman tree for string "enginee."

Figure 39. Huffman tree for string "engineer."
Up to the point as illustrated in *Figure 38*, the dynamic Huffman compression algorithm and the new memory limited dynamic Huffman compression algorithm generate identical trees.

The next symbol to appear in the input stream is the symbol ‘i’. But this symbol is no longer in the tree, it was replaced by the ‘r’ symbol. Again, the leaf node with the lowest weight is selected and replaced with the input symbol. This time it’s the ‘g’ symbol. The updated Huffman tree is presented in Figure 40. The output stream now looks as follows:

Cumulative input string: engineeri

Cumulative output stream:  e 0n 00g 100i 11 10 10 1000r 0011i

Finally, the last ‘n’ and ‘g’ are added as illustrated in *Figure 41* and *Figure 42*. When the ‘g’ is input to the algorithm, it does not exist in the Huffman tree anymore (it was replaced by the ‘i’ symbol.) This time the symbol with the lowest weight is the ‘i’ symbol. ‘g’ replaces ‘i’, then “g’s” weight is incrementated and the tree rebalanced.

| Final, cumulative input string: engineering |
| Final, cumulative output stream:  e 0n 00g 100i 11 10 10 1000r 0011i 01 000g |

The compression ratio would be \((8*7+25)/(8*11) = 92\%\). Compare this with the compression ratio of the previous example, where the Huffman tree is allowed to grow without a memory constraint; 74\%. These hand calculations were verified with computer simulations.
Figure 40. Huffman tree for string "engineeri," memory constrained.

Figure 41. Huffman tree for string "engineerin," memory constrained.
Space and Time Asymptotic Complexity

The space complexity for the dynamic Huffman compression is $O(n)$, where $n$ is the number of nodes in the Huffman tree. When the Huffman compression is applied to a transaction database, and the symbols to compress are the transaction item ID’s, then $n$ is also the number of items in the database. The memory limited dynamic Huffman compression has a space complexity of $O(\min(n,k))$, where $n$ is defined as before and $k$ is the maximum number of nodes in the tree. If the tree is stored in a binary tree data structure represented as a set of arrays as suggested by Knuth (1985), then the total memory requirements are about $12n \lg n + 2n \lg w$ bits, where $w$ is a bound on the number of bits required to hold the weight of the of all $n$ symbols (Knuth, 1985). For the memory limited version of the dynamic Huffman table, substitute $\min(n, k)$ into this
equation for \( n \). Thus, the memory limited version of the binary tree will never grow to be larger than \( 12r \lg r + 2r \lg w \) bits.

Creating and maintaining the memory limited dynamic Huffman tree requires the following four operations (and only these four operations):

- Identify and split the NYT node (researched by Knuth and others)
- Find a symbols node in the tree (researched by Knuth and others)
- Find the node with the minimum weight in the tree (new)
- Increment a node's weight and rebalance the tree (researched by Knuth and others)

Beyond these four operations, no other operations on the Huffman tree are required to maintain the tree and support the memory limited dynamic Huffman algorithm.

Furthermore, three of the operations are discussed by Knuth (1985). The algorithm, coding, time requirements are all well researched. Only one new operation is required to support the proposed memory limited dynamic Huffman algorithm (above Knuth’s original algorithm). The new operation is finding the node with minimum weight in the tree.

Knuth (1985) presents a time complexity for rebalancing the dynamic Huffman tree. Worst case the time complexity is \( O(b) \), where \( b \) is the level of the tree. The procedure for rebalancing the Huffman tree includes switching the node being incremented with the highest numbered node in its block. If a binary tree is a Huffman tree, then a block will never span more than two levels. Once the interchange is complete, then the algorithm moves to the parent node, where that node is then
interchanged with the highest numbered node in its block. The worst-case number of interchanges thus will be \( b \), where \( b \) is the maximum depth of the tree. Pigeon (2003) credits Knuth as further ‘fine tuning’ the time complexity as \( O(-\lg P(X = a_i)) \) for a symbol \( a_i \).

Understanding that the tree rebalancing has time complexity of \( O(b) \), worst case, requires understanding the rebalancing process. First, Knuth (1985) defines a ‘group’ as all nodes with the same weight. Knuth notes that a group will span a maximum of two levels (the simple explanation for this phenomenon is that if you select any node, the parent node exists on the level above it and it must have a weight higher than the selected node). The first step is to identify the leaf node whose weight is to be incremented. This node will be swapped with the highest numbered node in its group. When the node is swapped, its sub-tree (if any) is swapped with it. The node’s weight is then incremented. The algorithm then recursively proceeds to the parent of the node just incremented until the root is reached. Worst case, there will be \( b \) levels to iterate and swap subtrees. The swap itself is as simple as setting the 6 sets of pointers. The setting of pointers will be accomplished in constant asymptotic time.

Searching for a symbol in the Huffman tree can be done in \( O(1) \) time (constant time). An efficient data structure, such as a hash table, could be used. Knuth uses a hash table.

Identification of the NYT node and finding the node with the minimum weight are related. Finding the NYT node and splitting it can be done in \( O(1) \) asymptotic time. Typically, a pointer will be maintained to this important node so it can be readily found. Searching for an item with the lowest weight will not be necessary. The node with the
lowest weight will always be the highest numbered leaf node, not including the NYT node, in the table. Finding the node with the minimum weight can be accomplished in O(1) time. This node will always be on the same subtree as the NYT node. This subtree can only look like either of the subtrees in Figure 43.

![Figure 43. NYT node configuration.](image)

As suggested by Knuth, a pointer structure can be included as part of the nodes data structure to quickly traverse nodes. In the memory-limited version of the algorithm, a larger symbol word size would need to be accommodated and therefore the dictionary is required to be dynamically maintained. An O(1) asymptotic search time can be achieved in the memory limited algorithm as well.

**Expansion of the Compressed File**

Related to the asymptotic time and memory complexity of the algorithm, is the maximum number of ‘swaps’ that will result from various values of $k$, and how this is related to other parameters of the database. When the Huffman tree is full, and a new item is input to be compressed, then the algorithm must find the item in the tree with the lowest frequency and swap this item with the new item. This is a swap. The swap is important because it is the behavior that is different between the memory limited dynamic Huffman, and the ‘standard’ dynamic Huffman compression. If the number of swaps is
finite, then there is an assurance that the expansion of the compressed file is also finite.

When a swap occurs, the algorithm also outputs the NYT code. This is the longest code in the dynamic Huffman tree. The algorithm also outputs the uncompressed symbol. For each memory swap, the algorithm will output this pair because it constantly ‘forgets’ infrequent, but previously seen symbols. To establish that the expansion of the compressed file is limited, three limitations must first be established. The NYT code length is limited, the uncompressed symbol is limited in length, and the number of swaps is limited.

The uncompressed symbol length is a function of the overall system. This will be limited by the user to typically 8 bits, 32 bits, or a number of bits optimized for the size of the alphabet.

An upper bound on the length of the NYT code is possible using research provided by Abu-Mostafa and McEliece (2000). Here, the researchers calculate the maximum length of a Huffman code word. They find that if the probability, $p$, of a symbol is in the range $0 < p < 1/2$, and if $r$ is an index such that

$$\frac{1}{Fr + 3} < p < \frac{1}{Fr + 2}$$

where $F_r$ is the $r^{th}$ Fibonacci number, then the longest code word for that symbol is at most of length $r$ bits. The Fibonacci sequence is recursively defined as:

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}, \text{ for } n \geq 2$$

This research is significant because when a swap occurs, the output stream will receive the NYT symbol and the uncompressed symbol. The length of the NYT symbol at this point will be the same as the length of the least probable symbol up to that point in the stream. Note that this establishes an upper bound on the length of the symbol.
Swap Maximum Bound Analysis

An analysis of the upper bound on the maximum number of swaps follows. As an example, suppose during compression the Huffman tree is full and a new symbol appears in the stream. Assume that new symbol only occurs once in the stream (its frequency is 1). It is the least probable symbol. At most two swaps will occur. The first swap occurs when the new symbol is swapped with the least frequent symbol in the Huffman tree. The second swap occurs when the previously swapped out symbol is swapped in again because it is a more probable symbol.

For example, if the total number of symbols in the alphabet is 5, the limit is set to 4 symbols, and the least probable of the 5 symbols occurs once, then the most swaps that can occur is 2.

Next, assume that the least probable symbol occurs more than once. The most number of swaps that can occur will be two times \( p \), the probability of the symbol in the stream, times \( m \), the number of items in the stream.

This result can be generalized.

Formally, given:

\[ I = \{ i_1, \ldots, i_n \}; I \] is the set of symbols and \( n \) is the number of different symbols in the input stream \( S \).

\[ I_x = i_{x1}, \ldots, i_{xn}; I_x \] is the list of symbols ordered by probability where,

\[ P(i_{x1}) < P(i_{x2}) \cdots < P(i_{x(n-k)}) \cdots < P(i_{xn}); k \] is a user chosen constant and \( k < n \). \( k \) will be the maximum number of symbols allowed in the Huffman tree and \( P(x) \) is the probability of an item \( x \) in the stream.
Note that the case, \( P(i_{x1}) = P(i_{x2}) \cdots = P(i_{x(n-k)}) \cdots = P(i_{xn}) \) is not a realistic case. Huffman compression is possible only when there is a non-uniformity in the symbol probabilities. Secondly, for this discussion, note that the frequency of an item in the stream is equal to the total number of items in the stream times the items probability in the stream.

When the Huffman tree is full, and a new symbol is processed that is not already in the Huffman tree, a “swap” occurs in the Huffman tree. The new symbol is swapped with the symbol with the lowest probability in the tree.

Define the uncompressed data stream, \( S \), as:

\[
S = s_1 \cdots s_m ; \text{ where } m \text{ is the number of symbols in the uncompressed stream and each item in } S \in I.
\]

An upper bound on the number of swaps that can occur will be equal to:

\[
Max \ Swaps = 2m \left( P(i_{x1}) + P(i_{x2}) \cdots + P(i_{x(n-k)}) \right)
\]

\[
Max \ Swaps = 2m \sum_{j=1}^{n-k} P(i_{xj})
\]

Where \( m \) is the total number of items in the stream, \( k \) is the maximum number of items allowed in the Huffman tree. This is Equation 1. If \( I_x = i_{1} \ldots i_{x} \) is the list of items ordered by probability, then \( P(i_{x}) \) is the probability of an item, \( i_{x} \), in the stream, then \( n \) is the number of distinct items in the stream. The maximum swap’s that can occur is two times the sum of the frequency of the least probable \( n-k \) symbols.

Equation 1 can be rewritten to use the frequency of the items, rather than their probabilities.
\[ Max \ Swaps = 2 \sum_{j=1}^{n-k} F(i_j) \]

Where \( k \) is the maximum number of items allowed in the Huffman tree, \( F(i_k) \) is the frequency an item, \( i_k \) (the number of times it occurs in the stream at some point in time), and \( n \) is the number of distinct items in the stream at some point in time. A comparison of the memory and time complexity of the proposed memory limited algorithm to the dynamic Huffman algorithm as proposed by Knuth (1985) is summarized in Table 13.

Table 13
*Comparison of Asymptotic Time/Memory Complexity*

<table>
<thead>
<tr>
<th></th>
<th>Dynamic Huffman</th>
<th>Memory limited dynamic Huffman</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asymptotic time complexity to determine node with smallest frequency</td>
<td>(not required)</td>
<td>( O(1) )</td>
<td>Node with smallest frequency is last node in node list</td>
</tr>
<tr>
<td>Asymptotic memory complexity</td>
<td>( O(n) )</td>
<td>( O(\min(n,r)) )</td>
<td>where ( n ) is number of unique symbols seen in stream, and ( r ) is a chosen constant</td>
</tr>
<tr>
<td>Asymptotic time complexity to search tree for symbol</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>Constant time using a hash table and appropriate data structure for Huffman tree</td>
</tr>
<tr>
<td>Asymptotic time to identify and split the NYT node</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>Constant time</td>
</tr>
<tr>
<td>Asymptotic time to find minimum weight node</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>Constant time</td>
</tr>
<tr>
<td>Asymptotic max time complexity to re-build tree</td>
<td>( O(b) )</td>
<td>( O(b) )</td>
<td>where ( b ) is the max depth of the Huffman tree</td>
</tr>
<tr>
<td></td>
<td>( O(-\lg(P(a_i))) )</td>
<td>( O(-\lg(P(a_i))) )</td>
<td>For symbol ( a_i ), with probability ( P(a_i) ) (Pigeon, 2003)</td>
</tr>
</tbody>
</table>
"Tail" Items

In equation 1, the quantity \( n-k \) refers to the last \( n-k \) items in the list of items sorted by probability. These can be called the "tail items" in the list. The tail items are significant in this research because if there are many of them, then by equation 1, there may be many swaps leading to a greater expansion of the compressed file when the memory is limited \((k < n)\). In the example shown in Figure 44, the user chosen constant \( k \) is 305. The value \( n \) is the number of different items in the database. In this example, it is 470. The number of tail items is 165. Each one of these items has a frequency of less than 50 in this database. The shape of this item frequency histogram is typical for many real databases.

Figure 44. Histogram of item distribution in a database depicting tail items.
**Relationship of distribution and compression ratio**

The probability distribution of the database will affect the compression ratio in several ways. As demonstrated in the literature review section, a source stream where the input symbols have a uniform probability distribution will not compress well, or at all, with Huffman compression techniques. As a further example, the two benchmark synthetic databases exhibit a more uniform distribution profile. The synthetic databases did not compress as well as the other benchmark databases using the Huffman algorithm in the prior research.

Additionally, the synthetic databases have many tail items because the distribution profile is ‘flatter’. It is expected that these databases will not compress well with the memory limited dynamic Huffman compression algorithm as proposed herein because there are more items in the tail. Many tail items result in more swaps as indicated by equation 1. Many swaps will lead to a rapid decrease in the compression ratio because there will be many NYT and uncompressed symbols in the compressed output stream.

**Swap Minimum Bound**

The equation for the minimum bound on the number of swaps is:

\[
\text{Min Swaps} = n - k
\]

This is Equation 2. This minimum bound is found by noting that the minimum number of swaps occur when all the tail items appear at the end of the uncompressed stream. Then, one swap must occur of each item in the list of tail items.

**Proposed Work**

First, the FGK algorithm (Knuth, 1985) will be prototyped and verified to be correct. To verify correctness of the algorithm several texts will be compressed with the
algorithm. The text will then be decompressed to restore the file. The original and restored texts will be compared using commercial software that can compare files. In his original paper, Knuth (1985) gives some results of compressing files. The original files as used by Knuth will be compressed and the results compared to the results as documented by Knuth.

Second, the algorithm will be modified to limit the size of the tree using the Frequent item identification method proposed by Metwally, Agrawal, and El Abbadi (2005). This modification will be verified to be correct. Two variation of the memory limited dynamic Huffman compression will be prototyped. The first variation is the standard algorithm as outlined in Figure 28. In the second variation of the memory limited dynamic Huffman compression algorithm, when a replacement of a symbol occurs because the table is full, the weight of the added symbol will not be incremented. This will be discussed in more detail later in this research. See Figure 50 and Figure 51 for a comparison of the variation. The results of the memory limited dynamic Huffman algorithm will be verified to be correct before proceeding to the second step of the investigation. In the second step this research will benchmark the compression on several transaction databases used by researchers to benchmark datamining algorithms. These databases are listed in Table 1 and repeated in Table 14.

The format of the raw data from these benchmark databases varies. A synopsis of the database original formats is listed in Table 14. Some formats require pre-processing to bring them into the horizontal transaction format. During the initial investigation, it was found there were other anomalies in the databases. The databases will need to be
‘scrubbed’ to remove the anomalies and put the databases into a constituent format. The scrubbing process will focus on the following items:

- Items are in lexicographic order (for consistency with prior research).
- Gaps in Item ID numbering removed.
- Transaction ID added if it is missing.
- ‘Other’ anomalies, stray character, missing newline.
- Item ID’s separated by a single comma character.
- The file is in an ASCII coded binary format (for readability).
- It is important to add a transaction delimiter to end of line since each transaction is not a fixed length.
- Most importantly, the bit width of the item ID is assumed to be ideal for the database being considered. The bit width is calculated as
  \[ b = \text{ceiling}(\log_2 n) \], where \( n \) is the number of different items. This is important when the compression ratio is calculated. If for instance all item IDs in all databases were assumed to have a fixed 32-bit width, an inflated compression ratio would result.
- Verify prior research (conducted over two years ago) was similarly scrubbed.

Table 14
Structure of Benchmark Databases
A compression of these databases using the algorithms developed in phase 1 will be performed and the results of the compression tabulated. Since this is a single pass (adaptive) compression a graph of the effective compression ratio over time will be an important metric. This can be compared to the static compression ratios as tabulated in Table 11.

The compression ratio is a key metric used to compare the performance of the dynamic Huffman compression and the memory limited Huffman compression. The native format of the databases listed in Table 14 is an ASCII character format. The resulting compressed data stream will be in a binary format with a variable word length. For comparison, the ASCII input format will be changed to a binary format. In addition, this research assumes the binary input format is a fixed width word whose word length is adjusted to fit the maximum number of items in the item list and no more. More formally, if $b$ is the word length of the input file item IDs, and $n$ is the number of different items in the database:

$$b = \text{ceiling}(\log_2 n)$$
The compression ratio will be calculated as

\[
\frac{c}{u}
\]

where \( c \) is the size of the uncompressed string in bits, and \( u \) is the length of the uncompressed string in bits.

Other researchers, Abu-Mostafa and McEliece (2000), assume a fixed 32-bit TID code word size for all their research into transaction database compression. Their argument can be simplified to noting that 32 bits is a typical word size for computers. This would provide better overall compression results than using a word size that was adjusted for the maximum number of transaction IDs, as in this research. Having high compression numbers is not relevant in the long run. This research is concerned with the relative compression result of the non-memory limited Dynamic Huffman compression to the memory limited results obtained with the algorithm proposed here.

Each of the eight databases will be compressed with a Dynamic Huffman compression that is not memory limited. Several metrics are important to collect. These are:

- Bit size of the input file and the output file
- Compression ratio
- Number of different items. The number of different items is important to determine the input item bit size. The benchmark database files are all in ASCII. Further, each symbol is not of the optimum bit width. Preprocessing of each file is performed to optimize the input file size to be minimal by adjusting the input symbol bit size to be minimal.
• Algorithm run times (for comparison to the database size, and size of memory).

• Actual number of “swaps” that occurred

• Frequency of all symbols for calculation of “maximum swaps” (eq. 1).

• A calculation of the theoretical maximum number of swaps.

An important parameter is the number of symbols, $k$, to maintain in the frequent item identification List. As the number of symbols is increased it is expected that the compression ratio would approach that as obtained by the static two pass compression. The experiments will be performed for several values of $k$. The most effective value of $k$ will be selected. Finally, the most effective combination of algorithms will be run so a plot of $k$ versus the compression ratio can be developed to determine how the size of memory affects the compression ratio for real world applications.

Other plots to be generated are the compression ratio vs $k/n$. The quantity $k/n$ is a dimensionless number. It can be expressed as a percentage of the maximum number of items in the Huffman tree to the total number of different items in the database. It provides a look at the relationship between the compression ratio and $k$, that is independent of the value of $n$ for a database. In the initial study, it was determined that the database needed to be compressed with 20 different values of $k$, to get enough data points to plot well and to reasonably determine if the plot was a smooth function.

Figure 45 is a plot of the expected compression ratio that a dynamic compression algorithm will achieve without any memory limitation (as this research proposes). The lower and upper bounds on the compression are calculated by Vitter (1978). In this plot, the solid line indicates the compression ratio achieved by the static Huffman compression
algorithm on the file BMS1. The static compression results in a constant compression ratio (straight line) for the file because the algorithm does not need to learn the tokens and their frequencies to compress. The 12.5% compression ratio is the experimental value given in Table 11.

The short-dotted line in the graph is the lower bound of the expected dynamic compression ratio. It starts off at 100% because when the algorithm first starts processing the file, the Huffman tree is empty. For each new symbol that is encountered in the input stream (a stream is a file that is processed in a single pass) it must transmit the NYT symbol and the uncompressed symbol on the input file. Thus, initially, there may be more bits transmitted than received.

![Figure 45. Expected dynamic versus static compression ratio.](image)

As the algorithm processes and learns the items in the database, the compression is expected to asymptotically reach a limit defined by the compression ratio achieved with the static algorithm. In fact, Vitter (1978) proves that bounds on the maximum
compression achieved by the FGK algorithm will be less than $2S + t - 4n + 2$, and greater than $S - n + 1$; where $S$ is the number of transmitted bits of the static Huffman for a message with $n$ distinct symbols of length $t$. It is important to note that in the upper bound, the FGK algorithm will asymptotically reach the performance of the static Huffman compression. The number of bits transmitted by the FGK algorithm will never be more than twice the static Huffman algorithm. The dynamic compression results of the memory limited algorithm will be less than that of the FGK algorithm. The divergence between the solid line and the dotted lines will be determined by $k$, the maximum number of symbols to be held in memory.

Knuth (1981) provides detailed pseudocode for the implementation of the FGK algorithm. In addition, there are many implementations available in the public domain. As a base, public domain code will be chosen and verified to be correct. Knuth provides results of the algorithm on an available dataset, the first ten Fairy Tales, by Grimm. In the verification phase of this research the performance on this dataset with the implementation of FGK will be verified against Knuth’s results.

Final tasks for the proposed work is to verify the eq. 1, the upper bound on the number of swaps. This will be verified on the eight benchmark databases. Each complete database file will be read in and the number of occurrences of each symbol will be tabulated and sorted. The predicted upper bound on the number of swaps will be calculated from eq. 1.

Next, the database will be compressed for various values of $k$. The maximum number of swaps will be recorded and compared to the predicted upper bound.
Resources

This research will require a computer with a C# compiler to prototype and evaluate the compression algorithms. The C# compiler will be the Microsoft product Visual Studio 2013. This is free from the Nova Southeastern DreamSpark Academic Alliance. The benchmark databases will be required to evaluate the algorithm performance. These are available freely online from the UCI Machine Learning Repository (http://archive.ics.uci.edu/ml/datasets.html) and other sources (see benchmark list Table 1). Nova Library resources will be required for the literature review portion of this research. The Terasic DE0-Nano Cyclone IV development and evaluation board will be required for the FPGA processing study portion of this research.
Chapter 4

Results

Verification of Algorithm Coding

A prototype of the memory limited dynamic Huffman algorithm is coded and verified. Base code of a dynamic Huffman algorithm in C# was downloaded from http://dynamichuffman.codeplex.com (Bassman, 2014). This code is modified with the frequent item identification algorithm as proposed by Metwally, Agrawal, & El Abbadi (2005). It also includes modifications that instrument the code to gather measurements. As a verification, the algorithm was tested with the textual data of the first ten Grimm’s Fairy Tales. Although the original textual file used by Knuth (1985) was obtained, there were a few minor differences due to formatting characters in the file (the file was obtained in HTML format). Donald Knuth confirms (personal communication, September 17, 2016) the original file was for a ‘SAIL’ computer with a slightly non-standard character set. This may account for the minor differences.

First, the memory limited dynamic Huffman algorithm was used to compress the complete 1.41Mb text. In Chapter 4 of this research, results will be presented on compression of the actual benchmark datasets. The number of nodes in the Huffman tree were not limited. The compressed text was then decompressed. The recovered text was compared to the original text and no differences were found. Next, the same exercise was repeated with memory limited. Memory was limited by repeating the experiment with the number of nodes limited to 10, 20 and 30 nodes. The file was compress and decompressed. No differences were found between the original file and the restored file.
Next, the results obtained by Knuth (1985) were attempted to be duplicated. In this research, he tabulates two metrics, $\Sigma b$ and $\Sigma b_{opt}$ for 1000, 10000 and 100000 characters processed. Here, Knuth is comparing the performance of the Dynamic Huffman Algorithm with the Static Huffman Algorithm. The $\Sigma b$ quantity is the number of bits written into the output stream as produced by the dynamic Huffman algorithm. The $\Sigma b_{opt}$ quantity is the sum of the weighted path length of all symbols in the Huffman tree. This weighted path length will be the same as the number of bits transmitted by a static Huffman algorithm. Of course, the static Huffman algorithm must initially communicate or transmit the shape of the Huffman tree as well (or the canonical Huffman form of compressed codes must be used.) The $\Sigma b_{opt}$ quantity does not include this overhead. But the $\Sigma b$ quantity does include the bits required to transmit the NYT code and the original symbol to the decoder. As Knuth points out, “As the file gets larger, the overhead ratio grows to the point where a two-pass scheme would transmit fewer bits, yet the on-line method is not far from optimum.”

Table 15 presents a comparison of Knuth’s FGK algorithm results to the results of the memory limited dynamic Huffman algorithm. The small difference between the performance of the two algorithms may be attributed to minor difference in the symbols in the original file and the file used for comparison.
Table 15
*Algorithm Verification to Knuth’s Original Grimm Fairy Tale Results*

<table>
<thead>
<tr>
<th>Symbols processed</th>
<th>Dynamic vs. static</th>
<th>Memory limited dynamic Huffman algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Knuth’s original FGK results(^a)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(\Sigma b)</td>
<td>(\Sigma b_{opt})</td>
</tr>
<tr>
<td>1000</td>
<td>4558</td>
<td>4297</td>
</tr>
<tr>
<td>10000</td>
<td>44733</td>
<td>44272</td>
</tr>
<tr>
<td>100000</td>
<td>440164</td>
<td>439613</td>
</tr>
</tbody>
</table>


**Performance**

Next, the compression performance of the memory limited dynamic Huffman algorithm was put to the test. The compression performance was compared to the Grimms Fairy Tales file with ASCII characters taken “1 at a time”, and ASCII characters “taken 2 at a time.” The “1 at a time” experiment is identical to Knuth’s 7-bit character experiment, and the “2 at a time” experiment is equivalent to his 14-bit character experiment. In the “2 at a time” experiment, each symbol to compress is created from the next 2 ASCII characters in the input stream, whereas the 1 at a time algorithm creates a node in the Huffman tree for every new character in the input stream. The algorithm did not try to do any ‘preprocessing’ of the symbol pairs. It simply took the next two characters in the input stream and used them as the input symbol.

In the series “1 at a time” in *Figure 46*, the line flattens out to a compression ratio of about 53% after a limit of 60 nodes. Each character is 1 symbol. 60 characters are the maximum number symbols in the file when the characters are taken 1 at a time. This corresponds to about 120 nodes in the Huffman tree. The second series, “2 at a time”,
flattens out after 1900 nodes are set as the limit. This is because when the characters are taken 2 at a time 1900 nodes are produced. After 500 nodes are reached, the table indicates that “2 character at a time” processing compression ratio is better than the “1 character at a time” processing. Compression ratio is defined as compressed size/uncompressed size.

In this “two at a time” compression, many more node are produced. This is because the original file contained 60 different symbols. When taken in combination, up to 3600 combinations may be produced. The actual number of combinations of characters taken “2 at a time” the file produced was approximately 1900 different combinations.

![Graph](image)

Figure 46. One- versus 2-at-a-time compression ratio comparison.

Since the compression using the Memory limited dynamic Huffman algorithm was improved by considering character pairs as a single symbol, an experiment was
performed that considered three characters as a single symbol. The “3 at a time”
experimental compression results are compared to the character pair results in *Figure 47*. 
There was no preprocessing when creating the “3 at a time” symbols. The next three 7-
bit ASCII characters in the input stream were assembled into the 21-bit symbol. The 
memory limited compression ratio for the “2 at a time” compression is 48%, the ratio for
the “3 at a time” is 44%. An exact calculation of the improvement is 9.1% when
compression is done on 2 vs. 3 at a time. The number of symbols generated when the
text is considered “3 at a time” is 6,121. It takes at least 5,000 symbols when the
symbols are considered “3 at a time” to equal the compression ratio when the symbols are
considered “2 at a time”. In the case of “2 at a time” a maximum of 1920 symbols were
produced. “3 at a time” produced 6121 symbols. The cumulative number of bits
produced for the “3 at a time” dynamic algorithm vs the optimal number of bits a static
compression algorithm would produce are tabulated in Table 16.
Table 16

<table>
<thead>
<tr>
<th>Symbols processed</th>
<th>$\Sigma b$</th>
<th>$\Sigma b_{opt}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>22,025</td>
<td>8,848</td>
</tr>
<tr>
<td>10000</td>
<td>146,148</td>
<td>98,908</td>
</tr>
<tr>
<td>100000</td>
<td>1,114,803</td>
<td>1,014,560</td>
</tr>
</tbody>
</table>

In Figure 41 are the results of a compression experiment was run with the symbol chosen “4 at a time.” In this experiment four 7-bit ASCII characters are considered a single 28-bit symbol. This experiment produced up to 41,347 symbols. The best compression ratio was about 43.6%. It took up to 40,000 nodes to beat the compression obtained by 12,241 nodes where the symbols were created “3 at a time.” It’s difficult to
tell from the graph, but “4 at a time” beat “3 at a time” by a tiny 0.0021% (at a cost of about 29,000 nodes).

The cumulative number of bits produced for the “4 at a time” dynamic algorithm vs the optimal number of bits a static compression algorithm would produce are tabulated in Table 17.

![Figure 48. Three- versus 4-at-a-time compression ratio comparison.](image)

Table 17
*Bits Produced “4 at a Time”*

<table>
<thead>
<tr>
<th>Symbols processed</th>
<th>$\sum b$</th>
<th>$\sum b_{opt}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>32,539</td>
<td>9,361</td>
</tr>
<tr>
<td>10000</td>
<td>240,128</td>
<td>112,428</td>
</tr>
<tr>
<td>100000</td>
<td>1,609,116</td>
<td>1,199,647</td>
</tr>
</tbody>
</table>

Finally, an experiment was devised where the compression symbol is whole words. Here, a ‘word’ is defined as any sequence of character separated by a space character. This includes the special characters and punctuation characters. Multiple space characters are collapsed to a single character. The resulting single space character was not compressed into the output stream since it could be simply added during decoding as a separator between symbols. The results for “word at a time” are shown in *Figure 49.*

It is interesting that when the algorithm considers each symbol to be a word, the compression is more than the fixed number of character compression algorithms, at about 40.5%. On the other hand, a significant number of symbols are produced, 60,033 word symbols vs, 41,347 symbols chosen “4 at a time.”

Knuth (1985) concluded that the extra memory requirements for the character double experiment did not justify the incrementally better compression ratio.
Table 18

Bits Produced “Word at a Time”

<table>
<thead>
<tr>
<th>Symbols processed</th>
<th>Σb</th>
<th>Σb_{opt}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>31,256</td>
<td>8,043</td>
</tr>
<tr>
<td>10000</td>
<td>252,515</td>
<td>93,062</td>
</tr>
<tr>
<td>100000</td>
<td>2,005,366</td>
<td>1,001,072</td>
</tr>
</tbody>
</table>

Optimization of Algorithm

The frequent item identification algorithm (Metwally, Agrawal & El Abbadi, 2005) instructs that when the table (or in this case the Huffman tree) is full, a search is done for the symbol with the lowest weight or frequency. This symbol is replaced with
the new symbol. Its weight is then incremented. During verification of the algorithm it was noticed that perhaps the algorithm is over estimating the frequency of new symbols in the stream. The pseudocode for the frequent item identification algorithm is presented in *Figure 50.*

**Space-Saving Pseudocode**

1. \( T \leftarrow \emptyset \)
2. For each \( i \)
3. If \( i \in T \)
4. Then \( c_i \leftarrow c_i + 1 \)
5. Else if \(|T| < k\)
6. Then \( T \leftarrow T \cup \{i\} \)
7. \( c_i \leftarrow 1 \)
8. Else \( j \leftarrow \arg \min_{(i \in T)} c_j \)
9. \( c_j \leftarrow c_j + 1 \)
10. \( T \leftarrow T \cup \{i\} \setminus \{j\} \)

*Figure 50.* Frequent item identification pseudocode.

As a test the algorithm was modified to not increment a new symbol but only when replacing an old symbol. When a new symbol replaces an old symbol, the algorithm *does not* increment the symbol after it is replaced. When this change is applied to the Memory Limited Dynamic Huffman Algorithm it is called “Option B.” The pseudocode is presented in *Figure 51.* A comparison of the performance of the algorithm with and without “Option B” is presented *Figure 52.*
Space-Saving Pseudocode with “Option B”

1. \( T \leftarrow \emptyset \)
2. For each \( i \)
3. If \( i \in T \)
   
   Then \( c_i \leftarrow c_i + 1 \)
   
   Else if \(|T| < k\)
   
   Then \( T \leftarrow T \cup \{i\} \)
   
   \( c_i \leftarrow 1 \)

4. Else \( j \leftarrow \text{arg min}_{j \in T} c_j \)

5. \( T \leftarrow T \cup \{i\}\) \( \setminus \{j\} \)

Figure 51. Frequent item identification pseudocode with “Option B.”

The pseudocode of Figure 50 and Figure 51 are identical except for line #9. A description of the pseudocode follows. The array \( T \) is a set of tuples. It is initialized in line 1. The maximum size that \( T \) will be allowed to grow is \( k \), where \( k \) is a user defined constant. Each tuple in the set consists of an item ID and count. The variable \( i \) is the next item in the input stream to process. Line 2 iterates through all items in the stream. If the item is already in the set \( T \), then the associated count is incremented in line 4. If the item is not in the set, then there are two possibilities. The set is less than the maximum size, \( k \), or it has reached its maximum size. If there is room in the set then lines 6 and 7 add the item to the set, and the items count is initialized to 1. If the set is full (it has reached its maximum size), then line 8 sets \( j \) to the item in the set \( T \) whose count in minimum. Line 10 then removes the item with the minimum count from the set and adds the new item. The item is removed, but not its count. This is important. In line 9 the count, which remained from the removed item, is incremented. The “Option B” algorithm attempts to create a more accurate view of the item count by pessimistically not incrementing the removed item count. This option requires more research since it is
different from the frequent item identification algorithm as originally proposed by Metwally, Agrawal, and El Abbadi (2005).

To test the memory limited Dynamic Huffman algorithms compression ratio an experiment was performed where the input to each algorithm (with and without “Option B”) was the text of the file “Grimm’s Fairy Tales.”

The performance of the memory limited dynamic Huffman algorithm with the Option B outperforms the memory limited dynamic Huffman algorithm without the option by about 9% when the number of allowed nodes is low for this file. This is shown in Figure 52. Note that the two algorithms converge at about 60 nodes. This is about

\[ \text{Figure 52. “Option B” performance.} \]
half of the total number of nodes in the file if the Huffman tree could grow to accommodate all symbols. The actual measured values are shown in Table 19.

Table 19
*Measured Data for Modified Frequent Item Identification Algorithm*

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Without “Option B”</th>
<th>With “Option B”</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1.230698785</td>
<td>1.074234339</td>
</tr>
<tr>
<td>30</td>
<td>0.795063870</td>
<td>0.673381659</td>
</tr>
<tr>
<td>61</td>
<td>0.559337138</td>
<td>0.560810829</td>
</tr>
<tr>
<td>101</td>
<td>0.549010153</td>
<td>0.549058881</td>
</tr>
<tr>
<td>129</td>
<td>0.548742861</td>
<td>0.548742946</td>
</tr>
<tr>
<td>135</td>
<td>0.548742692</td>
<td>0.548742692</td>
</tr>
</tbody>
</table>

**Characteristics of Benchmark Transaction Data**

The benchmark transaction databases were collected from several sources including the KDD 2000 Cup website and the original datasets as provided by Agrawal, Imielinski and Swami (1993). See Table 1 for the citation to the source of each transaction database. Table 20 lists a brief description of the source and nature of the data.

Huffman compression is a statistical compression technique, the probability distribution of the item frequencies therefore is important to the achieved compression ratios. The probability distribution of each of the benchmark database examples are presented in *Figure 53 to Figure 60*. 
Table 20
*Description of Benchmark Transaction Database source data*

<table>
<thead>
<tr>
<th>Database</th>
<th>Database source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accidents</td>
<td>Traffic accident data</td>
</tr>
<tr>
<td>BMS1</td>
<td>KDD CUP 2000: click-stream data from a webstore named Gazelle</td>
</tr>
<tr>
<td>Kosarak</td>
<td>Click-stream data of a Hungarian on-line news portal</td>
</tr>
<tr>
<td>Retail</td>
<td>Retail market basket data from an anonymous Belgian retail store</td>
</tr>
<tr>
<td>BMS-POS</td>
<td>KDD CUP 2000: click-stream data from a webstore named Gazelle</td>
</tr>
<tr>
<td>BMS-WebView2</td>
<td>KDD CUP 2000: click-stream data from a webstore named Gazelle</td>
</tr>
<tr>
<td>T10I4D100K</td>
<td>Synthetic data from the IBM Almaden Quest research group</td>
</tr>
<tr>
<td>T40I10D100K</td>
<td>Synthetic data from the IBM Almaden Quest research group</td>
</tr>
</tbody>
</table>

*Figure 53.* Frequency of items in the accidents database.
**Figure 54.** Frequency of items in the BMS1 database.

**Figure 55.** Frequency of items in the BMS-POS database.
Figure 56. Frequency of items in the BMS-Webview2 database.

Figure 57. Frequency of items in the Kosarak database.
Figure 58. Frequency of items in the Retail database.

Figure 59. Frequency of items in the T40I10D100K database.
Each of the benchmark databases exhibit an approximate Zipf distribution (Powers, 1998). The Zipf distribution is known to have an approximate distribution that follows a distribution inversely proportional to its rank, where the rank in this case is the item number (Powers, 1998). Probability distributions that are more uniformly distributed, such as with the synthetic benchmark databases, did not compress well in the prior research. These databases did not compress well in this research as well as is indicated in the data that follow. Additionally, the synthetic databases had many items in their ‘tail’. This resulted in a rapid deterioration in the compression ratio as memory was limited. In fact, the achieved compression ratio (documented in the section to follow) was over 1.0, this indicates an expansion of the data rather than a compression.
Database Compression Results

The coding of the memory limited dynamic Huffman algorithm was verified to be correct and the benchmark databases were compressed using the algorithm. Summaries of several metrics and measurements follow for each benchmark database. Appendix A presents the raw data. As noted in Figure 59 and Figure 60, the synthetic databases T40I10D100K and T1014D100K show a flatter distribution curve. This indicates the synthetic data set has less entropy than the ‘real’ data sets (the data is more random). In the previous research work the synthetic data did not compress well with the Huffman compression. The compression ratios achieved were markedly less than the real data sets. In the results that follow, these synthetic data sets continued to not compress as well as the real data sets.

Accidents Benchmark Transaction Database Summary

The results of compressing the “Accidents” transaction database using the memory limited dynamic Huffman compression algorithm are presented as follows.

Note. Ideal item ID size = 9 bits. summarizes the produced bits and the minimum weighted path length for the Accidents benchmark database. The minimum weighted path length is the sum of the weighted path from the root to the leaves in the dynamic Huffman tree using the frequency of each leaf node as its weight multiplied by the number of links to the root.
Table 21
Accidents Produced Bits and Minimum Weighted Path Length

<table>
<thead>
<tr>
<th>Symbols processed</th>
<th>$\sum b$</th>
<th>$\sum b_{opt}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>7590</td>
<td>6311</td>
</tr>
<tr>
<td>10000</td>
<td>66044</td>
<td>64203</td>
</tr>
<tr>
<td>100000</td>
<td>640211</td>
<td>637892</td>
</tr>
</tbody>
</table>

*Note.* Ideal item ID size = 9 bits.

*Figure 61* through *Figure 64* show the results of applying the memory limited dynamic Huffman algorithm to the benchmark “Accidents” database. When plotted on a semi-log graph the histogram of the frequency of items appears to closely follow a straight line. This approximates a “Zipf” distribution. The paper “Applications and Explanations of Zipf’s Law” provides a good start to understanding the origin of this empirical law (Powers, 1998). Similar to the item frequency histogram, the compression ratio vs the number of nodes in the tree appears as a straight line. It’s important to note that the actual number of swaps is always less than the maximum number of swaps as predicted by eq. 1. This also appears almost as a straight line when plotted on a semi-log scale.

Note that compression ratios greater than 1.0 are an expansion of the source data rather than a compression.
Figure 61. Accidents static versus memory limited dynamic compression ratio.

Figure 62. Accidents actual versus calculated max swaps.
Figure 63. Accidents actual versus calculated max swaps (semi-log).

Figure 64. Accidents actual swaps versus the compression ratio (semi-log).
**BMS1 Benchmark Transaction Database Summary**

The results of compressing the “BMS1” transaction database using the memory limited dynamic Huffman compression algorithm follow. Table 22 summarizes the produced bits and the minimum weighted path length for the BMS1 benchmark database. The minimum weighted path length is the sum of the weighted path from the root to the leaves in the dynamic Huffman tree using the frequency of each leaf node as its weight multiplied by the number of links to the root.

Table 22

**BMS1 Produced Bits and Minimum Weighted Path Length**

<table>
<thead>
<tr>
<th>Symbols processed</th>
<th>$\sum b$</th>
<th>$\sum b_{opt}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>8996</td>
<td>6913</td>
</tr>
<tr>
<td>10000</td>
<td>64663</td>
<td>62087</td>
</tr>
<tr>
<td>100000</td>
<td>619374</td>
<td>615942</td>
</tr>
</tbody>
</table>

*Note.* Ideal item ID size = 9 bits.

*Figure 65 through Figure 68* show the results of applying the memory limited dynamic Huffman algorithm to the benchmark “BMS1” database. When plotted on a semi-log graph the histogram of the frequency of items appears to closely follow a straight line. Similarly, the compression ratio vs the number of nodes in the tree appears as a straight line. The actual number of swaps is always less than the maximum number of swaps as predicted by eq. 1. This also appears almost as a straight line when plotted on a semi-log scale.
**Figure 65.** BMS1 static versus memory limited dynamic compression ratio.

**Figure 66.** BMS1 actual versus calculated max swaps.
Figure 67. BMS1 actual versus calculated max swaps (semi-log).

Figure 68. BMS1 actual swaps versus the compression ratio (semi-log).
**BMS-POS Benchmark Transaction Database Summary**

The results of compressing the “BMS-POS” transaction database using the memory limited dynamic Huffman compression algorithm follow.

Table 23 summarizes the produced bits and the minimum weighted path length for the BMS-POS benchmark database. The minimum weighted path length is the sum of the weighted path from the root to the leaves in the dynamic Huffman tree using the frequency of each leaf node as its weight multiplied by the number of links to the root.

Note that compression ratios greater than 1.0 are an expansion of the source data rather than a compression.

<table>
<thead>
<tr>
<th>Symbols processed</th>
<th>$\Sigma b$</th>
<th>$\Sigma b_{opt}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>9196</td>
<td>6586</td>
</tr>
<tr>
<td>10000</td>
<td>68101</td>
<td>63725</td>
</tr>
<tr>
<td>100000</td>
<td>730370</td>
<td>723163</td>
</tr>
</tbody>
</table>

*Note.* Ideal item ID size = 11 bits.

*Figure 69* through *Figure 72* show the results of applying the memory limited dynamic Huffman algorithm to the benchmark “BMS-POS” database. When plotted on a semi-log graph the histogram of the frequency of items appears to closely follow a straight line. Similarly, the compression ratio vs the number of nodes in the tree appears as a straight line. Again, for this benchmark database, the actual number of swaps is always less than the maximum number of swaps as predicted by eq. 1. This also appears almost as a straight line when plotted on a semi-log scale.
**Figure 69.** BMS-POS static versus memory limited dynamic compression ratio.

**Figure 70.** BMS-POS actual versus calculated max swaps.
Figure 71. BMS-POS actual versus calculated max swaps (semi-log).

Figure 72. BMS-POS actual swaps versus the compression ratio (semi-log).

**BMS-Webview2 Benchmark Transaction Database Summary**

The results of compressing the “BMS-Webview2” transaction database using the memory limited dynamic Huffman compression algorithm follow. Table 24 summarizes
the produced bits and the minimum weighted path length for the BMS-Webview2 benchmark database. The minimum weighted path length is the sum of the weighted path from the root to the leaves in the dynamic Huffman tree using the frequency of each leaf node as its weight multiplied by the number of links to the root.

Table 24

<table>
<thead>
<tr>
<th>Symbols processed</th>
<th>$\sum b$</th>
<th>$\sum b_{opt}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>12737</td>
<td>7670</td>
</tr>
<tr>
<td>10000</td>
<td>103325</td>
<td>85794</td>
</tr>
<tr>
<td>100000</td>
<td>932816</td>
<td>902000</td>
</tr>
</tbody>
</table>

*Note.* Ideal item ID size = 11 bits.

*Figure 73* through *Figure 76* show the results of applying the memory limited dynamic Huffman algorithm to the benchmark “BMS-Webview2” database. When plotted on a semi-log graph the histogram of the frequency of items appears to closely follow a straight line. Again, for this benchmark database, the compression ratio vs the number of nodes in the tree appears as a straight line. It’s important to note that the actual number of swaps is always less than the maximum number of swaps as predicted by eq. 1. This also appears almost as a straight line when plotted on a semi-log scale.

Note that compression ratios greater than 1.0 are an expansion of the source data rather than a compression.
Figure 73. BMS-Webview2 static versus memory limited compression ratio.

Figure 74. BMS-Webview2 actual versus calculated max swaps.
Figure 75. BMS-Webview2 actual versus calculated max swaps (semi-log).

Figure 76. BMS-Webview2 actual swaps versus compression ratio (semi-log).
Kosarak Benchmark Transaction Database Summary

The results of compressing the “Kosarak” transaction database using the memory limited dynamic Huffman compression algorithm follow. Table 25 summarizes the produced bits and the minimum weighted path length for the Kosarak benchmark database. The minimum weighted path length is the sum of the weighted path from the root to the leaves in the dynamic Huffman tree using the frequency of each leaf node as its weight multiplied by the number of links to the root.

Table 25
Kosarak Produced Bits and Minimum Weighted Path Length

<table>
<thead>
<tr>
<th>Symbols processed</th>
<th>∑b</th>
<th>∑bopt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>15705</td>
<td>7617</td>
</tr>
<tr>
<td>10000</td>
<td>138970</td>
<td>92114</td>
</tr>
<tr>
<td>100000</td>
<td>1105771</td>
<td>958237</td>
</tr>
</tbody>
</table>

Note. Ideal item ID size = 16 bits.

Figure 77 through Figure 80 show the results of applying the memory limited dynamic Huffman algorithm to the benchmark “Kosarak” database. When plotted on a semi-log graph the histogram of the frequency of items appears to closely follow a straight line. Again, for this benchmark database, the compression ratio vs the number of nodes in the tree appears as a straight line. It’s important to note that the actual number of swaps is always less than the maximum number of swaps as predicted by eq. 1. This also appears almost as a straight line when plotted on a semi-log scale.

Note that compression ratios greater than 1.0 are an expansion of the source data rather than a compression.
Figure 77. Kosarak static vs. memory limited dynamic compression ratio

Figure 78. Kosarak actual versus calculated max swaps
Retail Benchmark Transaction Database Summary

The results of compressing the “Retail” transaction database using the memory limited dynamic Huffman compression algorithm follow.
Table 26 summarizes the produced bits and the minimum weighted path length for the Retail benchmark database. The minimum weighted path length is the sum of the weighted path from the root to the leaves in the dynamic Huffman tree using the frequency of each leaf node as its weight multiplied by the number of links to the root.

Table 26
Retail Produced Bits and Minimum Weighted Path Length

<table>
<thead>
<tr>
<th>Symbols processed</th>
<th>$\sum b$</th>
<th>$\sum b_{opt}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>16729</td>
<td>7953</td>
</tr>
<tr>
<td>10000</td>
<td>140590</td>
<td>95212</td>
</tr>
<tr>
<td>100000</td>
<td>1137663</td>
<td>1039451</td>
</tr>
</tbody>
</table>

Note. Ideal item ID size = 15 bits.

Figure 81 through Figure 84 show the results of applying the memory limited dynamic Huffman algorithm to the benchmark “Retail” database. When plotted on a semi-log graph the histogram of the frequency of items appears to closely follow a straight line. Again, for this benchmark database, the compression ratio vs the number of nodes in the tree appears as a straight line. It’s important to note that the actual number of swaps is always less than the maximum number of swaps as predicted by eq. 1. This also appears almost as a straight line when plotted on a semi-log scale.

Note that compression ratios greater than 1.0 are an expansion of the source data rather than a compression.
Figure 81. Retail static versus memory limited dynamic compression ratio.

Figure 82. Retail actual versus calculated max swaps.
Figure 83. Retail actual versus calculated max swaps (semi-log).

Figure 84. Retail actual swaps versus the compression ratio (semi-log).
**T40I10D100K Benchmark Transaction Database Summary**

The results of compressing the “T40I10D100K” transaction database using the memory limited dynamic Huffman compression algorithm follow. Table 27 summarizes the produced bits and the minimum weighted path length for the T40I10D100K benchmark database. The minimum weighted path length is the sum of the weighted path from the root to the leaves in the dynamic Huffman tree using the frequency of each leaf node as its weight multiplied by the number of links to the root.

Table 27

<table>
<thead>
<tr>
<th>Symbols processed</th>
<th>( \sum b )</th>
<th>( \sum b_{opt} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>14084</td>
<td>8696</td>
</tr>
<tr>
<td>10000</td>
<td>102203</td>
<td>92469</td>
</tr>
<tr>
<td>100000</td>
<td>939618</td>
<td>928024</td>
</tr>
</tbody>
</table>

*Note.* Ideal item ID size = 10 bits.

*Figure* 85 through *Figure* 88 show the results of applying the memory limited dynamic Huffman algorithm to the benchmark “T40I10D100K” database. When plotted on a semi-log graph the histogram of the frequency of items appears to closely follow a straight line. Again, for this benchmark database, the compression ratio vs the number of nodes in the tree appears as a straight line. It’s important to note that the actual number of swaps is always less than the maximum number of swaps as predicted by eq. 1. This also appears almost as a straight line when plotted on a semi-log scale.
Figure 85. T40I10D100K static versus memory limited dynamic compression ratio.

Figure 86. T40I10D100K actual versus calculated max swaps.
Figure 87. T40I10D100K actual versus calculated max swaps (semi-log).

Figure 88. T40I10D100K actual swaps versus the compression ratio (semi-log).
**T1014D100K Benchmark Transaction Database Summary**

The results of compressing the “T1014D100K” transaction database using the memory limited dynamic Huffman compression algorithm follow. Table 28 summarizes the produced bits and the minimum weighted path length for the T1014D100K benchmark database. The minimum weighted path length is the sum of the weighted path from the root to the leaves in the dynamic Huffman tree using the frequency of each leaf node as its weight multiplied by the number of links to the root.

Table 28

<table>
<thead>
<tr>
<th>Symbols processed</th>
<th>$\sum b$</th>
<th>$\sum b_{opt}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>12870</td>
<td>8227</td>
</tr>
<tr>
<td>10000</td>
<td>95467</td>
<td>87494</td>
</tr>
<tr>
<td>100000</td>
<td>891435</td>
<td>881649</td>
</tr>
</tbody>
</table>

*Note.* Ideal item ID size = 10 bits.

*Figure 89* through *Figure 92* show the results of applying the memory limited dynamic Huffman algorithm to the benchmark “T1014D100K” database. When plotted on a semi-log graph the histogram of the frequency of items appears to closely follow a straight line. Again, for this benchmark database, the compression ratio vs the number of nodes in the tree appears as a straight line. It’s important to note that the actual number of swaps is always less than the maximum number of swaps as predicted by eq. 1. This also appears almost as a straight line when plotted on a semi-log scale.

Note that compression ratios greater than 1.0 are an expansion of the source data rather than a compression.
Figure 89. T1014D100K static versus memory limited dynamic compression ratio.

Figure 90. T1014D100K actual versus calculated max swaps.
Figure 91. T1014D100K actual versus calculated max swaps (semi-log).

Figure 92. T1014D100K actual swaps versus the compression ratio (semi-log).
Discussion of Benchmark Compression Results

The first table presented above for each of the benchmark databases collected data summaries is the produced bits and the minimum weighted path length for 1000 symbols processed, 10,000 symbols processed and 100,000 symbols processed. The raw data collected is presented in Appendix A.

Note that for each of the databases the produced bits are greater than the minimum weighted path length, as would be expected. Recall the minimum weighted path length is the ideal number of bits in a compressed file that does not include the overhead of the dynamic compression or the overhead in transmitting the shape of the Huffman tree in a static compression scheme.

As noted above, most of the item distributions appear to follow the zipf (Powers, 1998) distribution. The distribution of items is a compilation, a list, of the number of times each item ID appears in the database. The list of the items is then sorted by frequency and displayed as a histogram. When plotted on a semi-log graph, the list appears almost as a straight line, following a zipf distribution. An exception may be the synthetic databases. The item distributions in these cases also appear to be straight line when plotted on a semi-log axis, but the distribution appears to be more uniform and flatter.

The next figure is that of the dynamic compression ratio vs the number of nodes. Also plotted on this figure is the static compression ratio as determined in prior research. There are a couple of important data points here. Firstly, in all cases the dynamic compression ratio approaches the static compression ratio when the number of nodes is large. When the number of nodes is large, the memory limited dynamic Huffman
compression algorithm performs identical to the dynamic Huffman compression algorithm as proposed by Knuth (1985). This is because the number of nodes allowed in the tree, $k$, will equal the number of different items in the database, $n$. Next, as identified by Knuth (1985) and Vitter (1989), it is noted that for a large number of compressed items, the static Huffman compression will approach that of the dynamic Huffman compression. This is because, as noted by Vitter and Knuth, the ‘overhead’ is a fixed amount.

As the number of allowed nodes in the Huffman tree, $k$, is reduced, it is noted that the compression ratio decreases. This is a result of eq. 1. Eq. 1 predicts that the number of swaps will increase as the $k$ is reduced (and the number of tail items, $n-k$, increases). The reduced compression is a result of more NYT symbols, and a corresponding uncompressed input symbol, being introduced into the compressed output stream, for each swap.

It is interesting to compare these results to the Pareto 80/20 principal (Reh, 2005). Pareto, based on the power law as proposed by Zipf, stated that 80% of the results come from the 20% vital few. It’s a general heuristic rule. If the Pareto rule holds for the compression of the benchmark databases using the memory limited dynamic Huffman compression algorithm, it would be expected that the achieved data compression would be 80% of the static data compression, if memory were limited to only 20% of the memory needed for the static compression. Based on this principal the following tabulates the results of the compression ratio vs $k$ with the actual 20% results, and what the Pareto 80/20 rule predicts in Table 29. The Pareto 80/20 rule is a rough ‘rule of
The rule seems to underestimate the achieved compression for all but three of the benchmark databases.

The figures present the relationships of the actual vs theoretical number of max swaps as predicted by eq.1. In every case the actual number of swaps is less than the theoretical number of maximum swaps. This lends evidence to the correctness of the equation.

Table 29
*Comparison of Actual 20% Compression Results to Pareto*

<table>
<thead>
<tr>
<th>Benchmark database</th>
<th>Actual 20% of final compression</th>
<th>Pareto 20%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accidents</td>
<td>79.86%</td>
<td>87%</td>
</tr>
<tr>
<td>BMS1</td>
<td>86.8%</td>
<td>86.6%</td>
</tr>
<tr>
<td>BMS-POS</td>
<td>72.3%</td>
<td>77.47%</td>
</tr>
<tr>
<td>BMS-Webview2</td>
<td>97.34%</td>
<td>93.9%</td>
</tr>
<tr>
<td>Kosarak</td>
<td>66.5%</td>
<td>73.28%</td>
</tr>
<tr>
<td>Retail</td>
<td>94.25%</td>
<td>86.9%</td>
</tr>
<tr>
<td>T40I10D100K</td>
<td>107%</td>
<td>112%</td>
</tr>
<tr>
<td>T1014D100K</td>
<td>106%</td>
<td>106%</td>
</tr>
</tbody>
</table>

Note that compression ratios greater than 100% are an expansion of the source data rather than a compression. For a discussion of the synthetic database results see the section on Characteristics of the Benchmark Transaction Data.

The final figure presents the compression ratio vs. the actual number of swaps. The plot is a monotonically decreasing line or curve, indicating that the compression ratio is some monotonically decreasing function of the number of swaps, and thus $k$. 
Chapter 5
Conclusions, Implications, Recommendations

Conclusions

The streaming transaction database is important to many applications. These include retail and online sales, stock market transactions, security tracking of layer 3 and layer 2 switches, fitness trackers, connected cars and a host of other applications where the data can be fit as a transaction ID key and a list of item IDs. Compression of the transaction data stream must be accomplished as compression of a horizontally formatted transaction database. A streaming database may span periods of weeks or years. During this period, the item IDs may change; new ones may appear and old ones disappear. The frequency of the item ID’s may have a temporal component.

In this research, Huffman compression is proposed as a solution to this compression. This research presents a modification to dynamic Huffman compression (Knuth, 1985) that can scale to the limited hardware resources of high speed, dedicated computing hardware (i.e., the FPGA). It allows memory constrained hardware to perform Huffman compression using an alphabet, or symbol list, that would otherwise overflow the limited memory. It adapts to the temporal changes to item frequencies.

This research developed a new algorithm, the memory limited dynamic Huffman algorithm, that was demonstrated to compress a data stream using less memory than the dynamic Huffman algorithm proposed by Knuth (1985), also known as algorithm FGK. The amount of memory the algorithm uses is user defined. The amount of memory consumed is chosen by a constant \( k \) that is defined as the number of nodes in the Huffman tree. In the original FGK algorithm, the number of nodes allowed in the tree
was defined as \( n \), and had to be greater than or equal to the total number of different items, or symbols, in the database schema.

When the constant \( k \) is chosen to equal \( n \), the memory limited dynamic Huffman algorithm operates identical to the original FGK algorithm. It provides similar compression ratios, and asymptotic time and memory requirements. On the other end, when \( k \to 1 \), the resulting compression ratio will be worse than 1.0. It will be an expansion. In fact, at this extreme, the resulting compression can be calculated exactly. Note that when \( k = 1 \), only one symbol will be allowed in the Huffman tree. This symbol will be the NYT symbol. There will not be any room in the tree for new symbols so the algorithm will constantly emit the NYT symbol in the ‘compressed’ output stream followed by the uncompressed input symbol. Further, the NYT symbol will have a length of 1 bit since it is the only item in the tree. Assume the input symbols have a bit length of \( b \). Assume compression ratio is defined as \( c/u \), where \( c \) is the compressed files length in bits, and \( u \) is the original file length in bits. Then, the resulting compression ratio will be \((b + 1)/b\), or

\[
\text{Compression Ratio}_{|k=1} = 1 + \frac{1}{b}
\]

This is Equation 3. Equation 3 will be greater than 1.0, thus it will be an expansion of the input file. The designer can expect the resulting compression ratio when \( k \) is chosen between \( 1 < k < n \), between these two extremes. The exact relationship between the compression ratio and \( k \) may require additional research, but the empirical evidence suggests that mathematically, it is a concave up, decreasing, curve between these two points. A concave down function \( f(x) \), is defined as one where all tangents to
the curve are below the function \( f(x) \). The exception to this shape seem to be the synthetic databases. The shape seems to ‘top’ out at low values of \( k \). This may be due to an item distribution that does not follow the zipf (Powers, 1998) distribution.

Note that it was shown in the Discussion of Benchmark Compression Results section, that for these benchmark databases Pareto’s 80/20 heuristic of the vital few (Reh, 2005) seems to roughly hold (see Table 29). These results will guide a designer in choosing the constant \( k \).

Finally, eq. 3 will only be an upper bound for the transaction database compression result obtained herein. The actual compression ratio will be less for two reasons. Both reasons are due to the compression model chosen for the transaction database. The compressed output stream includes an uncompressed transaction ID and a compressed transaction delimiter for each transaction. The transaction delimiter will also be encoded as a 1-bit prefix in the Huffman tree. A transaction delimiter occurs at the end of every transaction. It is used because each transaction is variable length. These differences will cause the worst-case compression ratio to be a bit better than the upper limit of \( 1 + 1/b \).

If a software engineer were to decide to use the memory limited dynamic Huffman algorithm, that person might have a couple questions:

Question? Can the memory limited dynamic Huffman algorithm compress and decompress a data file? Has it been verified against the Knuth (1985) FGK algorithm?
Answer: The algorithm has been verified to compress decompress data files. It has been verified to obtain results identical to algorithm FGK (Knuth, 1985).

Question? How much memory and processor time is required by the memory limited dynamic Huffman algorithm?

Answer: From a theoretical standpoint, the algorithm has identical memory and time requirements to algorithm FGK (Knuth, 1985) when \( k = n \). When \( k < n \), the memory requirements are less than algorithm FGK. The required size of the Huffman tree will be proportional to \( k \). It will be ‘controlled’ by the selection of the value \( k \).

Question? What is the main advantage of the memory limited dynamic Huffman compression algorithm?

Answer: The algorithm provides a ‘dial’ that controls the amount of memory consumed. This is particularly important for memory limited compute machines, or applications where the symbol list may be large. The dial is the constant \( k \). On one end the dial allows the algorithm to perform no worse than the original algorithm FGK, but also has the same memory requirement. On the other end, the dial will limit memory, but will also limit the resulting compression obtained.

Question? How can the resulting compression be determined for a given value of \( k \) at design time?

Answer: An equation to determine the compression ratio obtained would necessarily depend on the distribution of items and the distribution item
IDs in the transactions. Some rough guides can be provided. A static compression of the data, or a sampling technique could determine the best-case compression that can be obtained, i.e. $k = n$. At the other end of the spectrum, when $k = 1$, Eq. 3 can be used. Finally, Pareto’s 80/20 heuristic can be considered.

Thus, this research shows that the memory limited dynamic Huffman algorithm does not consume more memory or time than the FGK algorithm. It provides identical compress results to the FGK algorithm when it is uses the same amount of memory ($k = n$). The amount of memory the algorithm consumes can be ‘dialed down’, when memory is reduced then the resulting compression will also be reduced. When $k = 1$, the expected compression can be calculated by eq. 3.

**Implications**

Data mining can be a data intensive operation. Database operations such as the join and aggregation require access to large blocks of the database for their computation. In fact, as has been shown in this research, many algorithms exist for database mining applications that propose compression of the data as part of their algorithm. Generally, the proposal in these algorithms is for the data to be compressed so larger sections can reside in main memory and reduce secondary memory I/O operations. The memory limited dynamic Huffman algorithm provides a method to compress a transaction database so larger sections can fit into the primary memory. The FPGA space study characterized how much of the database can fit into the on-chip memory for each of the benchmark databases.
Finally, the memory limited dynamic Huffman algorithm may find use as a general compression algorithm in an application where a large list of symbols will result compared to the available main memory. The application would necessarily have a non-uniform item frequency distribution. An example of a non-linear distribution is the zipf distribution (Powers, 1998).

**Recommendations**

Vilfredo Paredo’s 80/20 heuristic (Rey, 2005) sparks an interesting possibility for further research. The memory limited dynamic Huffman algorithm is based on algorithm FGK. Perhaps a different approach, based on Paredo’s observation would yield a different compression algorithm. A second recommendation for further research would be to provide a more exact theoretical and analytical basis for the shape of the compression ratio vs. $k$ curve. Finally, additional research could focus on alternate methods of updating the frequency of the NYT code. The probability of the NYT code seems to differ between the memory limited and non-limited dynamic Huffman algorithms and may provide a possibility for algorithm optimization.

Recommendations for the software engineer interested in implementing the memory limited dynamic Huffman algorithm include understanding the nature of the frequency distribution of items in the database. Firstly, a non-uniform distribution of the item IDs should exist to take advantage of Huffman type compressions. Other compression techniques, such as RLE compression, do not rely on a non-uniform distribution. Second, Huffman compression will not compress a binary alphabet. For this application RLE compression may be a better choice. Finally, the item distribution in the target application should have many, infrequent, tail items. When the shape of the
histogram has many infrequent tail items then less swaps will occur. Less swaps will lead to a better overall compression ratio approaching that of the static Huffman compression algorithm and the opportunity to save memory as compared to algorithm FGK (Knuth, 1985).
Appendix A

Raw Data

Table A1

*Accidents Database Raw Data: Ideal Item ID Bit Size = 9, Uncompressed File Size = 106569477*

<table>
<thead>
<tr>
<th>k</th>
<th>swaps actual</th>
<th>max swaps</th>
<th>dynamic compression</th>
<th>Compressed size</th>
<th>run time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1.087</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>3323604</td>
<td>5001968</td>
<td>0.903</td>
<td>96232238</td>
<td>0:11:06.328</td>
</tr>
<tr>
<td>200</td>
<td>844037</td>
<td>1336496</td>
<td>0.783</td>
<td>83443900</td>
<td>0:26:24.503</td>
</tr>
<tr>
<td>300</td>
<td>170568</td>
<td>338352</td>
<td>0.739</td>
<td>78754844</td>
<td>0:38:58.548</td>
</tr>
<tr>
<td>500</td>
<td>4682</td>
<td>9158</td>
<td>0.738</td>
<td>78648274</td>
<td>1:01:33.063</td>
</tr>
<tr>
<td>600</td>
<td>680</td>
<td>1176</td>
<td>0.731</td>
<td>77902288</td>
<td>1:09:05.415</td>
</tr>
<tr>
<td>700</td>
<td>214</td>
<td>304</td>
<td>0.731</td>
<td>77902288</td>
<td>1:14:20.623</td>
</tr>
<tr>
<td>800</td>
<td>87</td>
<td>134</td>
<td>0.726</td>
<td>77369440</td>
<td>1:19:23.709</td>
</tr>
<tr>
<td>900</td>
<td>19</td>
<td>34</td>
<td>0.725</td>
<td>77262871</td>
<td>1:21:58.996</td>
</tr>
<tr>
<td>939</td>
<td>0</td>
<td>0</td>
<td>0.725</td>
<td>77262871</td>
<td>1:13:42.846</td>
</tr>
</tbody>
</table>

- = data point not recorded
Table A2

*BMS1 Database Raw Data: Ideal Item ID Bit Size = 9, Uncompressed File Size = 1881243*

<table>
<thead>
<tr>
<th>k</th>
<th>swaps actual</th>
<th>max swaps</th>
<th>dynamic compression</th>
<th>Compressed size</th>
<th>run time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1.064862434</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>1.049138787</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>82255</td>
<td>158212</td>
<td>0.898146598</td>
<td>1689632</td>
<td>00:00:05.168</td>
</tr>
<tr>
<td>200</td>
<td>53726</td>
<td>102794</td>
<td>0.868146598</td>
<td>1867453</td>
<td>00:00:11.140</td>
</tr>
<tr>
<td>300</td>
<td>33796</td>
<td>65376</td>
<td>0.802481126</td>
<td>1509662</td>
<td>00:00:17.091</td>
</tr>
<tr>
<td>400</td>
<td>20321</td>
<td>40340</td>
<td>0.795587667</td>
<td>1529454</td>
<td>00:00:23.512</td>
</tr>
<tr>
<td>500</td>
<td>11870</td>
<td>22642</td>
<td>0.792587667</td>
<td>1491050</td>
<td>00:00:31.530</td>
</tr>
<tr>
<td>600</td>
<td>5573</td>
<td>11424</td>
<td>0.738729659</td>
<td>1389730</td>
<td>00:00:37.083</td>
</tr>
<tr>
<td>700</td>
<td>2583</td>
<td>4230</td>
<td>0.731160192</td>
<td>1375490</td>
<td>00:00:43.251</td>
</tr>
<tr>
<td>800</td>
<td>800</td>
<td>1262</td>
<td>0.724355652</td>
<td>1362689</td>
<td>00:00:47.000</td>
</tr>
<tr>
<td>900</td>
<td>184</td>
<td>234</td>
<td>0.722091192</td>
<td>1358429</td>
<td>00:00:47.941</td>
</tr>
<tr>
<td>996</td>
<td>1</td>
<td>1</td>
<td>0.721839762</td>
<td>1357956</td>
<td>00:00:47.108</td>
</tr>
</tbody>
</table>
Table A3

**BMS-POS Database Raw Data: Ideal Item ID Bit Size = 11, Uncompressed File Size = 42705542**

<table>
<thead>
<tr>
<th>k</th>
<th>swaps actual</th>
<th>max swaps</th>
<th>dynamic compression</th>
<th>Compressed size</th>
<th>run time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1.07</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>500</td>
<td>327902</td>
<td>601654</td>
<td>0.706750449</td>
<td>30182161</td>
<td>00:37:38.000</td>
</tr>
<tr>
<td>800</td>
<td>138686</td>
<td>258540</td>
<td>0.673883029</td>
<td>28778540</td>
<td>01:05:51.086</td>
</tr>
<tr>
<td>1000</td>
<td>86444</td>
<td>159386</td>
<td>0.662901831</td>
<td>28309582</td>
<td>01:22:32.290</td>
</tr>
<tr>
<td>1250</td>
<td>48611</td>
<td>88084</td>
<td>0.656886312</td>
<td>28052686</td>
<td>01:21:12.228</td>
</tr>
<tr>
<td>1500</td>
<td>26466</td>
<td>46664</td>
<td>0.652291499</td>
<td>27856462</td>
<td>01:39:25.122</td>
</tr>
<tr>
<td>1750</td>
<td>13342</td>
<td>23064</td>
<td>0.64874786</td>
<td>27705129</td>
<td>01:53:57.752</td>
</tr>
<tr>
<td>2000</td>
<td>6329</td>
<td>10440</td>
<td>0.647521743</td>
<td>27652767</td>
<td>02:16:07.656</td>
</tr>
<tr>
<td>2250</td>
<td>2873</td>
<td>4228</td>
<td>0.646985092</td>
<td>27629849</td>
<td>02:26:37.191</td>
</tr>
<tr>
<td>2500</td>
<td>1113</td>
<td>1476</td>
<td>0.646025989</td>
<td>27588890</td>
<td>02:34:37.465</td>
</tr>
<tr>
<td>2750</td>
<td>453</td>
<td>590</td>
<td>0.64583496</td>
<td>27580732</td>
<td>02:30:55.774</td>
</tr>
<tr>
<td>3000</td>
<td>191</td>
<td>312</td>
<td>0.645639973</td>
<td>27572405</td>
<td>02:34:23.000</td>
</tr>
<tr>
<td>3250</td>
<td>35</td>
<td>62</td>
<td>0.645636836</td>
<td>27572271</td>
<td>02:44:31.288</td>
</tr>
<tr>
<td>3500</td>
<td>1</td>
<td>1</td>
<td>0.645636789</td>
<td>27572269</td>
<td>02:44:25.756</td>
</tr>
</tbody>
</table>
Table A4

BMS-Webview2 Database Raw Data: Ideal Item ID Bit Size = 12, Uncompressed File Size = 5227212

<table>
<thead>
<tr>
<th>k</th>
<th>swaps actual</th>
<th>max swaps</th>
<th>dynamic compression</th>
<th>Compressed size</th>
<th>run time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1.07</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>500</td>
<td>222053</td>
<td>389856</td>
<td>0.995397279</td>
<td>6248595</td>
<td>02:12.134</td>
</tr>
<tr>
<td>1000</td>
<td>169790</td>
<td>278912</td>
<td>0.981231111</td>
<td>5547280</td>
<td>08:17.011</td>
</tr>
<tr>
<td>2000</td>
<td>110609</td>
<td>152996</td>
<td>0.957945031</td>
<td>5111926</td>
<td>10:02.757</td>
</tr>
<tr>
<td>3000</td>
<td>74292</td>
<td>83110</td>
<td>0.906933088</td>
<td>4949820</td>
<td>29:22.104</td>
</tr>
<tr>
<td>4000</td>
<td>37528</td>
<td>40058</td>
<td>0.88524571</td>
<td>4627367</td>
<td>22:01.680</td>
</tr>
<tr>
<td>5000</td>
<td>7391</td>
<td>14832</td>
<td>0.838683795</td>
<td>4383978</td>
<td>34:04.437</td>
</tr>
<tr>
<td>6000</td>
<td>292</td>
<td>2552</td>
<td>0.797821286</td>
<td>4170381</td>
<td>53:39.925</td>
</tr>
<tr>
<td>6684</td>
<td>1</td>
<td>1</td>
<td>0.782809077</td>
<td>4091909</td>
<td>26:41.860</td>
</tr>
</tbody>
</table>
Table A5

*Kosarak Database Raw Data: Ideal Item ID Bit Size = 16, Uncompressed File Size = 144144272*

<table>
<thead>
<tr>
<th>k</th>
<th>swaps actual</th>
<th>max swaps</th>
<th>dynamic compression</th>
<th>Compressed size</th>
<th>run time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1.040408994</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>5355473</td>
<td>10229312</td>
<td>1.002609982</td>
<td>144520486</td>
<td>00:03:31.840</td>
</tr>
<tr>
<td>500</td>
<td>4508579</td>
<td>8019836</td>
<td>0.968413188</td>
<td>139591214</td>
<td>00:43:14.285</td>
</tr>
<tr>
<td>1000</td>
<td>3810859</td>
<td>6680176</td>
<td>0.958983268</td>
<td>138231945</td>
<td>01:34:13.578</td>
</tr>
<tr>
<td>2000</td>
<td>3010300</td>
<td>4987808</td>
<td>0.876791913</td>
<td>126384532</td>
<td>03:17:35.897</td>
</tr>
<tr>
<td>5000</td>
<td>1760752</td>
<td>2775958</td>
<td>0.78738929</td>
<td>113497656</td>
<td>08:15:39.560</td>
</tr>
<tr>
<td>8000</td>
<td>1212674</td>
<td>1794606</td>
<td>0.738336047</td>
<td>106426912</td>
<td>11:37:35.803</td>
</tr>
<tr>
<td>10000</td>
<td>921185</td>
<td>1378346</td>
<td>0.704294847</td>
<td>101520068</td>
<td>13:57:21.962</td>
</tr>
<tr>
<td>20000</td>
<td>362249</td>
<td>539258</td>
<td>0.643793151</td>
<td>92799095</td>
<td>1.21:44:24.255</td>
</tr>
<tr>
<td>40000</td>
<td>93930</td>
<td>116156</td>
<td>0.618570032</td>
<td>89163327</td>
<td>5.05:12:14.685</td>
</tr>
<tr>
<td>45000</td>
<td>66846</td>
<td>80812</td>
<td>0.616307147</td>
<td>88837145</td>
<td>7.08:42:45.846</td>
</tr>
<tr>
<td>55000</td>
<td>31335</td>
<td>38614</td>
<td>0.612512463</td>
<td>88290163</td>
<td>10.14:16:53.829</td>
</tr>
<tr>
<td>65000</td>
<td>14311</td>
<td>17538</td>
<td>0.611349156</td>
<td>88122479</td>
<td>12.21:51:18.261</td>
</tr>
<tr>
<td>75000</td>
<td>4450</td>
<td>7538</td>
<td>0.610768751</td>
<td>88038817</td>
<td>15.04:35:25.072</td>
</tr>
<tr>
<td>82542</td>
<td>1</td>
<td>1</td>
<td>0.610709602</td>
<td>88030291</td>
<td>16.06:55:02.405</td>
</tr>
</tbody>
</table>
Table A6

*Retail Database Raw Data: Ideal Item ID Bit Size = 15, Uncompressed File Size = 14951070*

<table>
<thead>
<tr>
<th>k</th>
<th>swaps actual</th>
<th>max swaps</th>
<th>dynamic compression</th>
<th>Compressed size</th>
<th>run time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1.05683</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1000</td>
<td>568599</td>
<td>972768</td>
<td>0.98428168</td>
<td>14716064.3</td>
<td>00:10:45.711</td>
</tr>
<tr>
<td>5000</td>
<td>310246</td>
<td>427200</td>
<td>0.974856582</td>
<td>14575149</td>
<td>00:44:02.832</td>
</tr>
<tr>
<td>10000</td>
<td>160639</td>
<td>182530</td>
<td>0.872991431</td>
<td>13052156</td>
<td>02:46:16.608</td>
</tr>
<tr>
<td>20000</td>
<td>32017</td>
<td>33470</td>
<td>0.775495199</td>
<td>11594483</td>
<td>05:45:00.891</td>
</tr>
<tr>
<td>25000</td>
<td>11701</td>
<td>12204</td>
<td>0.745606769</td>
<td>11147619</td>
<td>07:59:02.450</td>
</tr>
<tr>
<td>30000</td>
<td>2885</td>
<td>2938</td>
<td>0.729850573</td>
<td>10912047</td>
<td>10:51:57.407</td>
</tr>
<tr>
<td>32943</td>
<td>1</td>
<td>1</td>
<td>0.724280737</td>
<td>10828772</td>
<td>18:38:34.070</td>
</tr>
</tbody>
</table>

Table A7

*T40I10D100K Database Raw Data: Ideal Item ID Bit Size = 10, Uncompressed File Size = 40605070*

<table>
<thead>
<tr>
<th>k</th>
<th>swaps actual</th>
<th>max swaps</th>
<th>dynamic compression</th>
<th>Compressed size</th>
<th>run time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1.08</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1000</td>
<td>1072226</td>
<td>1128498</td>
<td>1.061436294</td>
<td>47160202</td>
<td>00:00:49:22.493</td>
</tr>
<tr>
<td>1200</td>
<td>583436</td>
<td>644068</td>
<td>1.055367815</td>
<td>42853284</td>
<td>00:00:51:21.000</td>
</tr>
<tr>
<td>1400</td>
<td>283515</td>
<td>304320</td>
<td>0.995682066</td>
<td>40429740</td>
<td>00:01:03:01.910</td>
</tr>
<tr>
<td>1600</td>
<td>98857</td>
<td>100410</td>
<td>0.948954502</td>
<td>38532364</td>
<td>00:01:17:50.235</td>
</tr>
<tr>
<td>1800</td>
<td>9031</td>
<td>9306</td>
<td>0.930760026</td>
<td>37793576</td>
<td>00:00:57:44.700</td>
</tr>
<tr>
<td>1886</td>
<td>1</td>
<td>1</td>
<td>0.929365471</td>
<td>37736950</td>
<td>00:01:00:57.321</td>
</tr>
</tbody>
</table>
Table A8

*T1014D100K Database Raw Data: Ideal Item ID Bit Size = 10, Uncompressed File Size = 11102280*

<table>
<thead>
<tr>
<th>k</th>
<th>swaps actual</th>
<th>max swaps</th>
<th>dynamic compression</th>
<th>Compressed size</th>
<th>run time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1.098598</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>500</td>
<td>-</td>
<td>-</td>
<td>1.0538722</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>900</td>
<td>283328</td>
<td>294560</td>
<td>0.995832478</td>
<td>11056011</td>
<td>12:37.4</td>
</tr>
<tr>
<td>1100</td>
<td>152396</td>
<td>165744</td>
<td>0.989900183</td>
<td>10990149</td>
<td>19:22.6</td>
</tr>
<tr>
<td>1300</td>
<td>46965</td>
<td>76610</td>
<td>0.91402847</td>
<td>10147800</td>
<td>18:18.1</td>
</tr>
<tr>
<td>1500</td>
<td>15523</td>
<td>20550</td>
<td>0.895552535</td>
<td>9942675</td>
<td>22:03.7</td>
</tr>
<tr>
<td>1742</td>
<td>1</td>
<td>1</td>
<td>0.883236867</td>
<td>9805943</td>
<td>27:15.9</td>
</tr>
</tbody>
</table>
References


Teubner, J., Muller, R., & Alonso, G. (2011b). Frequent item computation on a chip. *IEEE Transactions on Knowledge and Data Engineering*, 23(8), 1169-1181. doi: 10.1109/TKDE.2010.216


Certification of Authorship

Submitted to (Advisor’s Name): Dr. Junping Sun

Student’s Name: Damon Bruccoleri

Date of Submission: 2018

Purpose and Title of Submission: Database Streaming Compression on Memory-Limited Machines

Certification of Authorship: I hereby certify that I am the author of this document and that any assistance I received in its preparation is fully acknowledged and disclosed in the document. I have also cited all sources from which I obtained data, ideas, or words that are copied directly or paraphrased in the document. Sources are properly credited according to accepted standards for professional publications. I also certify that this paper was prepared by me for this purpose.

Student's Signature: __________________________________________________