United States Patent	12395185
Kind Code	B2
Date of Patent	August 19, 2025
Inventor(s)	Cooper; Joshua et al.

Adaptive data processing system with dynamic technique selection and feedback- driven optimization

Abstract

A system and method for adaptive data processing combining compression and encryption. The system analyzes input data characteristics, compares probability distributions, and creates a transformation matrix to convert data into a dyadic distribution. It generates a main data stream of transformed data and a secondary stream of transformation information. The system dynamically selects and applies processing techniques, including transformation, encoding, compression, and encryption algorithms, based on analyzed characteristics and real-time performance metrics. It compresses the main data stream using Huffman coding and implements security measures to protect the output. A feedback loop monitors technique effectiveness, updates a knowledge base, and influences future selections. The system can operate in lossless, lossy, or modified lossless modes, adapting to different application requirements. This approach offers an efficient solution for scenarios where both data reduction and security are critical concerns.

Inventors: **Cooper; Joshua (Columbia, SC), Fickes; Grant (Columbia, SC), Yeomans; Charles (Orinda, CA)**

Applicant: **AtomBeam Technologies Inc.** (Moraga, CA)

Family ID: **1000008762861**

Assignee: **ATOMBEAM TECHNOLOGIES INC (Moraga, CA)**

Appl. No.: **18/981612**

Filed: **December 15, 2024**

Prior Publication Data

Document Identifier	Publication Date
US 20250119158 A1	Apr. 10, 2025

Related U.S. Application Data

continuation parent-doc US 18305305 20230421 US 11811428 20231107 child-doc US 18503135
continuation parent-doc US 17514913 20211029 US 11424760 20220823 child-doc US 17875201
continuation parent-doc US 17458747 20210827 US 11422978 20220823 child-doc US 17875201 20220727
continuation parent-doc US 16455655 20190627 US 10509771 20191217 child-doc US 16716098
continuation parent-doc US 17404699 20210817 US 11385794 20220712 child-doc US 17727913
continuation parent-doc US 17875201 20220727 US 11700013 20230711 child-doc US 18161080
continuation-in-part parent-doc US 18770652 20240712 PENDING child-doc US 18981612
continuation-in-part parent-doc US 18503135 20231106 US 12237848 child-doc US 18770652
continuation-in-part parent-doc US 18190044 20230324 US 11831343 20231128 child-doc US 18305305
continuation-in-part parent-doc US 17875201 20220727 US 11700013 20230711 child-doc US 18190044
continuation-in-part parent-doc US 17404699 20210817 US 11385794 20220712 child-doc US 17514913
continuation-in-part parent-doc US 16455655 20190627 US 10509771 20191217 child-doc US 17404699
continuation-in-part parent-doc US 16200466 20181126 US 10476519 20191112 child-doc US 16455655
continuation-in-part parent-doc US 15975741 20180509 US 10303391 20190528 child-doc US 16200466
continuation-in-part parent-doc US 16923039 20200707 US 11232076 20220125 child-doc US 17458747
continuation-in-part parent-doc US 16716098 20191216 US 10706018 20200707 child-doc US 16923039
continuation-in-part parent-doc US 17727913 20220425 US 11620051 20230404 child-doc US 18190044 20230324
continuation-in-part parent-doc US 17234007 20210419 US 11782879 20231010 child-doc US 18305305 20230421
continuation-in-part parent-doc US 17180439 20210219 US 11366790 20220621 child-doc US 17234007
continuation-in-part parent-doc US 16923039 20200707 US 11232076 20220125 child-doc US 17180439
continuation-in-part parent-doc US 18462309 20230906 US 12224776 child-doc US 18981612 20241215
continuation-in-part parent-doc US 18161080 20230129 US 12218695 child-doc US 18462309
us-provisional-application US 62578824 20171030
us-provisional-application US 63027166 20200519
us-provisional-application US 62926723 20191028
us-provisional-application US 63388411 20220712
us-provisional-application US 63485518 20230216
us-provisional-application US 63232041 20210811
us-provisional-application US 63140111 20210121
us-provisional-application US 63486170 20230221

Publication Classification

Int. Cl.: G06F7/00 (20060101); **G06N20/00** (20190101); **H03M7/30** (20060101)

U.S. Cl.:

CPC **H03M7/3059** (20130101); **G06N20/00** (20190101); **H03M7/6005** (20130101);

Field of Classification Search

CPC: H03M (7/3059); H03M (7/6005); G06N (20/00)

USPC: 700/29; 700/43; 700/117; 700/118; 700/119; 700/248; 700/286; 700/291; 700/295; 705/7.11; 705/7.13; 705/7.25; 705/7.31; 705/7.35; 705/28; 705/37; 705/317; 707/693; 707/688; 707/737; 707/769; 707/770; 707/802

References Cited

U.S. PATENT DOCUMENTS

Patent No.	Issued Date	Patentee Name	U.S. Cl.	CPC
9524392	12/2015	Naehrig et al.	N/A	N/A
10346043	12/2018	Golden et al.	N/A	N/A
11424760	12/2021	Cooper et al.	N/A	N/A
2009/0055139	12/2008	Agarwal	707/999.005	G06Q 30/0254
2012/0254333	12/2011	Chandramouli	709/206	G06F 40/10
2012/0331025	12/2011	Gemulla	708/200	G06F 7/588
2014/0351229	12/2013	Gupta	N/A	N/A
2015/0066834	12/2014	Jeffries	707/752	H03M 7/70
2015/0254566	12/2014	Chandramouli et al.	N/A	N/A
2018/0196609	12/2017	Niesen	N/A	N/A
2019/0140658	12/2018	Cooper et al.	N/A	N/A
2020/0128307	12/2019	Li	N/A	N/A
2020/0264778	12/2019	Yang et al.	N/A	N/A
2020/0295779	12/2019	Ki et al.	N/A	N/A
2020/0395955	12/2019	Choi et al.	N/A	N/A
2021/0056079	12/2020	Cooper et al.	N/A	N/A
2021/0373776	12/2020	Cooper et al.	N/A	N/A

Primary Examiner: Uddin; Md I

Attorney, Agent or Firm: Galvin Patent Law LLC

Background/Summary

CROSS-REFERENCE TO RELATED APPLICATIONS

(1) Priority is claimed in the application data sheet to the following patents or patent applications, each of which is expressly incorporated herein by reference in its entirety: Ser. No. 18/770,652 Ser. No. 18/503,135 Ser. No. 18/305,305 Ser. No. 18/190,044 Ser. No. 17/875,201 Ser. No. 17/514,913 Ser. No. 17/404,699 Ser. No. 16/455,655 Ser. No. 16/200,466 Ser. No. 15/975,741 Ser. No. 62/578,824 Ser. No. 17/458,747 Ser. No. 16/923,039 Ser. No. 63/027,166 Ser. No. 16/716,098 Ser. No. 62/926,723 Ser. No. 63/388,411 Ser. No. 17/727,913 Ser. No. 63/485,518 Ser. No. 63/232,041 Ser. No. 17/234,007 Ser. No. 17/180,439 Ser. No. 63/140,111 Ser. No. 18/462,309 Ser. No.

63/486,170 Ser. No. 18/161,080

BACKGROUND OF THE INVENTION

Field of the Invention

(2) The present invention is in the field of computer data storage and transmission, and in particular to statistical analysis of datasets for automated algorithm training.

Discussion of the State of the Art

(3) As computers become an ever-greater part of our lives, and especially in the past few years, data storage has become a limiting factor worldwide. Prior to about 2010, the growth of data storage far exceeded the growth in storage demand. In fact, it was commonly considered at that time that storage was not an issue, and perhaps never would be, again. In 2010, however, with the growth of social media, cloud data centers, high tech and biotech industries, global digital data storage accelerated exponentially, and demand hit the zettabyte (1 trillion gigabytes) level. Current estimates are that data storage demand will reach 175 zettabytes by 2025. By contrast, digital storage device manufacturers produced roughly 1 zettabyte of physical storage capacity globally in 2016. We are producing data at a much faster rate than we are producing the capacity to store it. In short, we are running out of room to store data, and need a breakthrough in data storage technology to keep up with demand.

(4) The primary solutions available at the moment are the addition of additional physical storage capacity and data compression. As noted above, the addition of physical storage will not solve the problem, as storage demand has already outstripped global manufacturing capacity. Data compression is also not a solution. A rough average compression ratio for mixed data types is 2:1, representing a doubling of storage capacity. However, as the mix of global data storage trends toward multi-media data (audio, video, and images), the space savings yielded by compression either decreases substantially, as is the case with lossless compression which allows for retention of all original data in the set, or results in degradation of data, as is the case with lossy compression which selectively discards data in order to increase compression. Even assuming a doubling of storage capacity, data compression cannot solve the global data storage problem. The method disclosed herein, on the other hand, works the same way with any type of data.

(5) Transmission bandwidth is also increasingly becoming a bottleneck. Large data sets require tremendous bandwidth, and we are transmitting more and more data every year between large data centers. On the small end of the scale, we are adding billions of low bandwidth devices to the global network, and data transmission limitations impose constraints on the development of networked computing applications, such as the “Internet of Things”.

(6) Furthermore, as quantum computing becomes more and more imminent, the security of data, both stored data and data streaming from one point to another via networks, becomes a critical concern as existing encryption technologies are placed at risk.

(7) What is needed is a fundamentally new approach to data storage and transmission, that allows for dramatically more storage versus existing methods on the same physical storage device, and that supports automated system efficacy monitoring and model training.

SUMMARY OF THE INVENTION

(8) The inventor has developed a system and method for adaptive data processing that combines compression and encryption techniques. This system utilizes statistical analyses of datasets to compare probability distributions, creates transformation matrices, and transforms input data into dyadic distributions. The system dynamically selects and applies processing techniques based on analyzed characteristics and real-time performance metrics, generating a main data stream of transformed data and a secondary stream of transformation information. It implements security measures to protect the output and operates in various modes to accommodate different application requirements.

(9) According to a first preferred embodiment, a system for adaptive data processing is disclosed, comprising: a computing device comprising a processor and memory; a dynamic processing

subsystem stored in the memory and operable on the processor, wherein the dynamic processing subsystem, when operating on the processor, causes the processor to: receive input data; analyze characteristics of the input data; retrieve a first estimated probability distribution associated with a previous training dataset; estimate a second probability distribution of the input data; compare the first and second probability distributions to determine a difference value; create a transformation matrix based on the properties of the input data; transform the input data into a dyadic distribution using the transformation matrix; generate a main data stream of transformed data and a secondary data stream of transformation information; dynamically select and apply a combination of processing techniques; compress the main data stream using Huffman coding; and adaptively adjust the selection and application of processing techniques based on real-time performance metrics. The system further comprises a feedback loop mechanism and an output module for creating new codewords, combining data streams, packaging processed data, implementing security measures, and transmitting the packaged data.

(10) According to another preferred embodiment, a method for adaptive data processing is disclosed, comprising the steps of: receiving input data; analyzing characteristics of the input data; retrieving a first estimated probability distribution; estimating a second probability distribution; comparing the probability distributions; creating a transformation matrix; transforming the input data into a dyadic distribution; generating main and secondary data streams; dynamically selecting and applying processing techniques; compressing the main data stream; adaptively adjusting technique selection; monitoring effectiveness; updating a knowledge base; influencing future selections; creating new codewords; combining data streams; packaging processed data; implementing security measures; and transmitting the packaged data and metadata to a recipient system.

Description

BRIEF DESCRIPTION OF THE DRAWING FIGURES

(1) The accompanying drawings illustrate several aspects and, together with the description, serve to explain the principles of the invention according to the aspects. It will be appreciated by one skilled in the art that the particular arrangements illustrated in the drawings are merely exemplary, and are not to be considered as limiting of the scope of the invention or the claims herein in any way.

(2) FIG. **1** is a diagram showing an embodiment of the system in which all components of the system are operated locally.

(3) FIG. **2** is a diagram showing an embodiment of one aspect of the system, the data deconstruction engine.

(4) FIG. **3** is a diagram showing an embodiment of one aspect of the system, the data reconstruction engine.

(5) FIG. **4** is a diagram showing an embodiment of one aspect of the system, the library management module.

(6) FIG. **5** is a diagram showing another embodiment of the system in which data is transferred between remote locations.

(7) FIG. **6** is a diagram showing an embodiment in which a standardized version of the sourceblock library and associated algorithms would be encoded as firmware on a dedicated processing chip included as part of the hardware of a plurality of devices.

(8) FIG. **7** is a diagram showing an example of how data might be converted into reference codes using an aspect of an embodiment.

(9) FIG. **8** is a method diagram showing the steps involved in using an embodiment to store data.

(10) FIG. **9** is a method diagram showing the steps involved in using an embodiment to retrieve

data.
(11) FIG. **10** is a method diagram showing the steps involved in using an embodiment to encode data.
(12) FIG. **11** is a method diagram showing the steps involved in using an embodiment to decode data.
(13) FIG. **12** is a diagram showing an exemplary system architecture, according to a preferred embodiment of the invention.
(14) FIG. **13** is a diagram showing a more detailed architecture for a customized library generator.
(15) FIG. **14** is a diagram showing a more detailed architecture for a library optimizer.
(16) FIG. **15** is a diagram showing a more detailed architecture for a transmission and storage engine.
(17) FIG. **16** is a method diagram illustrating key system functionality utilizing an encoder and decoder pair.
(18) FIG. **17** is a method diagram illustrating possible use of a hybrid encoder/decoder to improve the compression ratio.
(19) FIG. **18** is a flow diagram illustrating the use of a data encoding system used to recursively encode data to further reduce data size.
(20) FIG. **19** is an exemplary system architecture of a data encoding system used for cyber security purposes.
(21) FIG. **20** is a flow diagram of an exemplary method used to detect anomalies in received encoded data and producing a warning.
(22) FIG. **21** is a flow diagram of a data encoding system used for Distributed Denial of Service (DDOS) attack denial.
(23) FIG. **22** is an exemplary system architecture of a data encoding system used for data mining and analysis purposes.
(24) FIG. **23** is a flow diagram of an exemplary method used to enable high-speed data mining of repetitive data.
(25) FIG. **24** is an exemplary system architecture of a data encoding system used for remote software and firmware updates.
(26) FIG. **25** is a flow diagram of an exemplary method used to encode and transfer software and firmware updates to a device for installation, for the purposes of reduced bandwidth consumption.
(27) FIG. **26** is an exemplary system architecture of a data encoding system used for large-scale software installation such as operating systems.
(28) FIG. **27** is a flow diagram of an exemplary method used to encode new software and operating system installations for reduced bandwidth required for transference.
(29) FIG. **28** is a block diagram of an exemplary system architecture of a codebook training system for a data encoding system, according to an embodiment.
(30) FIG. **29** is a block diagram of an exemplary architecture for a codebook training module, according to an embodiment.
(31) FIG. **30** is a block diagram of another embodiment of the codebook training system using a distributed architecture and a modified training module.
(32) FIG. **31** is a method diagram illustrating the steps involved in using an embodiment of the codebook training system to update a codebook.
(33) FIG. **32** is a block diagram illustrating an exemplary hardware architecture of a computing device.
(34) FIG. **33** is a block diagram illustrating an exemplary logical architecture for a client device.
(35) FIG. **34** is a block diagram showing an exemplary architectural arrangement of clients, servers, and external services.
(36) FIG. **35** is another block diagram illustrating an exemplary hardware architecture of a computing device.

(37) FIG. **36** is a system diagram illustrating a data deconstruction engine containing a compression engine that may be used for pre-compression of high-entropy data streams before being compacted using codebook techniques, according to an embodiment.
(38) FIG. **37** is a system diagram illustrating a data reconstruction engine containing a decompression engine that may be used for decompressing high-entropy data streams after they have been de-compacted from their codebook-translated format, according to an embodiment.
(39) FIG. **38** is a method diagram illustrating the operation of a compression engine within a data deconstruction engine, according to an embodiment.
(40) FIG. **39** is a method diagram illustrating the operation of a decompression engine within a data reconstruction engine, according to an embodiment.
(41) FIG. **40** is a message flow diagram illustrating the use of pre-compression of high-entropy data streams before being compacted using codebook techniques, transmitted to a data reconstruction engine, expanded from their codebook format, and decompressed, before being transmitted to a receiver or end-user, according to an embodiment.
(42) FIG. **41** is a block diagram illustrating exemplary architecture of adaptive data processing system.
(43) FIG. **42** is a method diagram illustrating the use of adaptive data processing system.
(44) FIG. **43** is a method diagram illustrating the use of dynamic processing subsystem.
(45) FIG. **44** is a method diagram illustrating the use of feedback loop mechanism.
(46) FIG. **45** is a method diagram illustrating the use of output subsystem.
(47) FIG. **46** is a method diagram illustrating the use of system controller.

DETAILED DESCRIPTION OF THE INVENTION

(48) The inventor has conceived and reduced to practice a system and method for adaptive data processing using automated system efficacy monitoring and model training, wherein statistical analyses of test datasets are used to determine if the probability distribution of two datasets are within a pre-determined range, and responsive to that determination new encoding and decoding algorithms may be retrained in order to produce new data sourceblocks. The new data sourceblocks may then be processed and assigned new codewords which are compiled into an updated codebook which may be distributed back to encoding and decoding systems and devices.
(49) An important factor in machine learned algorithm and model degradation over time is related to data drift. Data drift is a change in the distribution of data such as a change between real-time production data and a baseline (training) dataset. Indeed, most real-world datasets suffer from this problem and can cause models and their underlying algorithms to produce sub-optimal outputs the longer they are in use. To make the systems robust against data drift and other model behavioral changes, an adaptive data processing system is disclosed which facilitates periodic sampling of incoming, real-world data, which may be gathered and analyzed to determine if data drift has occurred. Furthermore, if data drift is discovered, then the system may automatically retrain existing algorithms in order to account for the changes in the incoming data.
(50) The adaptive data processing system operates on the principle of dynamically selecting and applying a combination of processing techniques based on analyzed characteristics of input data and the difference between current and historical probability distributions. These processing techniques may include transformation algorithms, encoding algorithms, compression algorithms, and encryption algorithms. The system leverages concepts from information theory, cryptography, and data compression to achieve efficient and secure data processing.
(51) At the core of the system is a dynamic processing subsystem that analyzes input data characteristics and compares probability distributions. The system retrieves a first estimated probability distribution associated with a previous training dataset from a monitoring database. It then estimates a second probability distribution of the input data. By comparing these distributions, the system can determine a difference value, which is crucial for detecting data drift and adapting processing techniques accordingly.

(52) The dynamic processing subsystem selects and applies processing techniques based on the analyzed characteristics and the calculated difference value. For instance, when dealing with image data, the system may apply a mathematical transform followed by an entropy encoding algorithm. The selection of techniques is not static but adaptively adjusted based on real-time performance metrics.
(53) A key feature of the system is its feedback loop mechanism. This mechanism continuously monitors the effectiveness of the applied processing techniques, updates a knowledge base with performance data, and influences future selections of processing techniques based on historical performance. This adaptive approach ensures that the system remains effective even as data characteristics change over time.
(54) The system incorporates an output module that creates new codewords for processed data, packages the processed data with metadata describing the applied techniques, and transmits the packaged data and metadata to a recipient system. This approach not only ensures efficient data processing but also provides the recipient with necessary information for proper decoding and interpretation of the data.
(55) The adaptive data processing system can operate in various modes, including a lossless mode where perfect reconstruction of the original data is possible, and potentially a lossy mode for scenarios where perfect reconstruction is not required. The system's flexibility allows it to be tailored to different data types and processing requirements.
(56) Security is a fundamental aspect of the system. The dynamic selection and application of processing techniques, combined with the creation of new codewords and metadata packaging, provide a level of security that goes beyond traditional encryption methods. The system's ability to adapt to changing data characteristics also makes it resilient against potential attacks that might exploit static processing methods.
(57) The system's performance can be analyzed using various metrics from information theory, such as entropy and Kullback-Leibler divergence. These metrics help in optimizing the system's efficiency and in quantifying the effectiveness of the applied processing techniques.
(58) At the core of the dyadic platform is the observation that both lossless compression and encryption share a common goal: transforming data reversibly and efficiently into an approximately uniformly random string. In compression, this uniformity indicates that the data cannot be further compressed, while in encryption, it ensures that no information can be extracted from the encrypted sequence. By leveraging this shared objective, the platform achieves both compression and encryption simultaneously, offering significant improvements in efficiency and security over traditional methods that treat these processes separately.
(59) The dyadic system operates on the principle of transforming input data into a dyadic distribution whose Huffman encoding is close to uniform. This is achieved through the use of a transformation matrix B, which maps the original data distribution to the desired dyadic distribution. The transformations applied to the data are then stored in a compressed secondary stream, which is interwoven with the main data stream.
(60) The dyadic platform is built upon solid theoretical foundations from information theory, cryptography, and data compression. These foundations provide the mathematical basis for the system's ability to simultaneously compress and encrypt data efficiently.
(61) The system leverages the concept of entropy from information theory. For a discrete probability distribution P, the entropy H(P) is defined as: H(P)=−Σ(p(x)*log.sub.2(p(x))) where p(x) is the probability of symbol x. Entropy represents the theoretical limit of lossless data compression. The dyadic distribution algorithm aims to transform the data distribution to approach this limit.
(62) An important aspect of the dyadic system is the transformation of data into a dyadic distribution. A distribution is dyadic if all probabilities are of the form ½.sup.k for some integer k. Dyadic distributions are optimal for Huffman coding, as they result in integer-length codewords.

The system utilizes Huffman coding, which is provably optimal for symbol-by-symbol encoding with known probabilities. The system constructs a Huffman tree T(C) for the encoding C, where the depth d(v) of a vertex v in T(C) relates to the probability of the symbol it represents. The transformation matrix B is important to the platform's operation. It is designed to satisfy: Σ(σ(ω′)*b_ωω′)=π(ω) for all ω∈Ω where σ is the original distribution, π is the Huffman-implied distribution, and Ω is the set of states. This ensures that applying B to data sampled from σ results in data distributed according to T.
(63) The dyadic algorithm models the input data as samples from a Markov chain. This allows for the use of mixing time t in security analysis. The mixing time is defined as: τ=min {t:Δ(t)≤1/(2e)} where Δ(t) is the maximum total variation distance between the chain's distribution at time t and its stationary distribution.
(64) The security of the dyadic system is analyzed using a modified version of Yao's next-bit test. For a bit string C(x) produced by the dyadic algorithm, it is proved that: |Pr[C(x)_j=0]−½|≤2*(e{circumflex over ( )}(−└j/(2M−m)┘/τ))/(1−e{circumflex over ( )}(−1/τ)) where M and m are the maximum and minimum codeword lengths, and τ is the mixing time of the Markov chain.
(65) The system's performance may be analyzed using the Kullback-Leibler (KL) divergence, which measures the difference between two probability distributions P and Q: KL(P∥Q)=π(P(x)*log(P(x)/Q(x))). This is used to bound the difference between the original and transformed distributions.
(66) The platform's compression efficiency is related to the cross-entropy H(σ,π) between the original distribution σ and the Huffman-implied distribution π. It is proved that: |H(σ,π)−H(π)|≤(M√2)/In(2) where M is the maximum codeword length. This bounds the extra bits needed to encode σ beyond its entropy rate.
(67) The security of the interleaved streams is analyzed using probability bounds on predicting bits in the combined stream. For the interleaved stream Z, it can be shown that: |Pr[Z_j=0]−½|≤max(2*(e{circumflex over ( )}(−└j′/(2M−m)┘/(τ∥B∥.sub.1)))/(1−e{circumflex over ( )}(−1/(τ∥B∥.sub.1))), b_(j−j′)) where j′ is the number of bits from the main stream, ∥B∥.sub.1 is the 1-norm of B, and b_k bounds the predictability of the transformation stream.
(68) Another key feature of the dyadic system is its ability to pass a modified version of Yao's “next-bit test”, a standard measure of cryptographic security. This means that nearby bits in the output stream cannot be predicted with substantial accuracy, even given all previous data. Importantly, the dyadic system achieves this level of security while requiring significantly fewer bits of entropy than standard encryption methods.
(69) The dyadic system can operate in various modes: a lossless mode where both the main data stream and the transformation data are transmitted, allowing perfect reconstruction of the original data, a modified lossless mode, and a lossy mode where only the transformed data is transmitted, providing even stronger encryption at the cost of perfect reconstruction.
(70) In its operation, dyadic platform first analyzes the input data to estimate its probability distribution. It then constructs a Huffman encoding based on this distribution, which defines another distribution x over the data space. The system partitions the data space into overrepresented states (where the original probability is greater than or equal to the Huffman-implied probability) and underrepresented states (where the original probability is less than the Huffman-implied probability).
(71) The transformation matrix B is then constructed to map the original distribution to the Huffman-implied distribution. This matrix has several important properties: 1. It is row-stochastic, meaning the sum of each row is 1. 2. When applied to data sampled from the original distribution, it produces the Huffman-implied distribution. 3. Underrepresented states only transform to themselves. 4. Overrepresented states only transform to themselves or to underrepresented states.
(72) The dyadic distribution algorithm applies these transformations to the input data, producing a main data stream that follows the Huffman-implied distribution (and is thus highly compressible)

and a secondary stream containing the transformation information. These streams may be interleaved to produce the final output.
(73) The security of this system stems from several factors. First, the transformation process introduces controlled randomness into the data. Second, the interleaving of the two streams makes it difficult to separate the transformed data from the transformation information. Finally, the system passes a modified next-bit test, ensuring that future bits cannot be predicted with significant accuracy even given all previous bits.
(74) Importantly, the dyadic distribution algorithm requires significantly less entropy (random bits) than traditional encryption methods. This is because the randomness is introduced in a controlled manner through the transformation process, rather than being applied to the entire data stream.
(75) The system may also include protections against various side-channel attacks, implemented by a security module. These include measures to prevent timing attacks, power analysis, cache attacks, and other potential vulnerabilities.
(76) In summary, the adaptive data processing system provides a novel approach to data processing that combines dynamic technique selection, continuous performance monitoring, and adaptive retraining. This approach ensures efficient, secure, and adaptable data processing, making it well-suited for handling the diverse and evolving data landscapes of modern computing environments.
(77) One or more different aspects may be described in the present application. Further, for one or more of the aspects described herein, numerous alternative arrangements may be described; it should be appreciated that these are presented for illustrative purposes only and are not limiting of the aspects contained herein or the claims presented herein in any way. One or more of the arrangements may be widely applicable to numerous aspects, as may be readily apparent from the disclosure. In general, arrangements are described in sufficient detail to enable those skilled in the art to practice one or more of the aspects, and it should be appreciated that other arrangements may be utilized and that structural, logical, software, electrical and other changes may be made without departing from the scope of the particular aspects. Particular features of one or more of the aspects described herein may be described with reference to one or more particular aspects or figures that form a part of the present disclosure, and in which are shown, by way of illustration, specific arrangements of one or more of the aspects. It should be appreciated, however, that such features are not limited to usage in the one or more particular aspects or figures with reference to which they are described. The present disclosure is neither a literal description of all arrangements of one or more of the aspects nor a listing of features of one or more of the aspects that must be present in all arrangements.
(78) Headings of sections provided in this patent application and the title of this patent application are for convenience only, and are not to be taken as limiting the disclosure in any way.
(79) Devices that are in communication with each other need not be in continuous communication with each other, unless expressly specified otherwise. In addition, devices that are in communication with each other may communicate directly or indirectly through one or more communication means or intermediaries, logical or physical.
(80) A description of an aspect with several components in communication with each other does not imply that all such components are required. To the contrary, a variety of optional components may be described to illustrate a wide variety of possible aspects and in order to more fully illustrate one or more aspects. Similarly, although process steps, method steps, algorithms or the like may be described in a sequential order, such processes, methods and algorithms may generally be configured to work in alternate orders, unless specifically stated to the contrary. In other words, any sequence or order of steps that may be described in this patent application does not, in and of itself, indicate a requirement that the steps be performed in that order. The steps of described processes may be performed in any order practical. Further, some steps may be performed simultaneously despite being described or implied as occurring non-simultaneously (e.g., because one step is described after the other step). Moreover, the illustration of a process by its depiction in a drawing

does not imply that the illustrated process is exclusive of other variations and modifications thereto, does not imply that the illustrated process or any of its steps are necessary to one or more of the aspects, and does not imply that the illustrated process is preferred. Also, steps are generally described once per aspect, but this does not mean they must occur once, or that they may only occur once each time a process, method, or algorithm is carried out or executed. Some steps may be omitted in some aspects or some occurrences, or some steps may be executed more than once in a given aspect or occurrence.

(81) When a single device or article is described herein, it will be readily apparent that more than one device or article may be used in place of a single device or article. Similarly, where more than one device or article is described herein, it will be readily apparent that a single device or article may be used in place of the more than one device or article.

(82) The functionality or the features of a device may be alternatively embodied by one or more other devices that are not explicitly described as having such functionality or features. Thus, other aspects need not include the device itself.

(83) Techniques and mechanisms described or referenced herein will sometimes be described in singular form for clarity. However, it should be appreciated that particular aspects may include multiple iterations of a technique or multiple instantiations of a mechanism unless noted otherwise. Process descriptions or blocks in figures should be understood as representing modules, segments, or portions of code which include one or more executable instructions for implementing specific logical functions or steps in the process. Alternate implementations are included within the scope of various aspects in which, for example, functions may be executed out of order from that shown or discussed, including substantially concurrently or in reverse order, depending on the functionality involved, as would be understood by those having ordinary skill in the art.

Definitions

(84) The term “bit” refers to the smallest unit of information that can be stored or transmitted. It is in the form of a binary digit (either 0 or 1). In terms of hardware, the bit is represented as an electrical signal that is either off (representing 0) or on (representing 1).

(85) The term “byte” refers to a series of bits exactly eight bits in length.

(86) The terms “compression” and “deflation” as used herein mean the representation of data in a more compact form than the original dataset. Compression and/or deflation may be either “lossless”, in which the data can be reconstructed in its original form without any loss of the original data, or “lossy” in which the data can be reconstructed in its original form, but with some loss of the original data.

(87) The terms “compression factor” and “deflation factor” as used herein mean the net reduction in size of the compressed data relative to the original data (e.g., if the new data is 70% of the size of the original, then the deflation/compression factor is 30% or 0.3.)

(88) The terms “compression ratio” and “deflation ratio”, and as used herein all mean the size of the original data relative to the size of the compressed data (e.g., if the new data is 70% of the size of the original, then the deflation/compression ratio is 70% or 0.7.)

(89) The term “data” means information in any computer-readable form.

(90) The term “sourcepacket” as used herein means a packet of data received for encoding or decoding. A sourcepacket may be a portion of a data set.

(91) The term “sourceblock” as used herein means a defined number of bits or bytes used as the block size for encoding or decoding. A sourcepacket may be divisible into a number of sourceblocks. As one non-limiting example, a 1 megabyte sourcepacket of data may be encoded using 512 byte sourceblocks. The number of bits in a sourceblock may be dynamically optimized by the system during operation. In one aspect, a sourceblock may be of the same length as the block size used by a particular file system, typically 512 bytes or 4,096 bytes.

(92) A “database” or “data storage subsystem” (these terms may be considered substantially synonymous), as used herein, is a system adapted for the long-term storage, indexing, and retrieval

of data, the retrieval typically being via some sort of querying interface or language. “Database” may be used to refer to relational database management systems known in the art, but should not be considered to be limited to such systems. Many alternative database or data storage system technologies have been, and indeed are being, introduced in the art, including but not limited to distributed non-relational data storage systems such as Hadoop, column-oriented databases, in-memory databases, and the like. While various aspects may preferentially employ one or another of the various data storage subsystems available in the art (or available in the future), the invention should not be construed to be so limited, as any data storage architecture may be used according to the aspects. Similarly, while in some cases one or more particular data storage needs are described as being satisfied by separate components (for example, an expanded private capital markets database and a configuration database), these descriptions refer to functional uses of data storage systems and do not refer to their physical architecture. For instance, any group of data storage systems of databases referred to herein may be included together in a single database management system operating on a single machine, or they may be included in a single database management system operating on a cluster of machines as is known in the art. Similarly, any single database (such as an expanded private capital markets database) may be implemented on a single machine, on a set of machines using clustering technology, on several machines connected by one or more messaging systems known in the art, or in a master/slave arrangement common in the art. These examples should make clear that no particular architectural approaches to database management is preferred according to the invention, and choice of data storage technology is at the discretion of each implementer, without departing from the scope of the invention as claimed.
(93) The term “effective compression” or “effective compression ratio” refers to the additional amount data that can be stored using the method herein described versus conventional data storage methods. Although the method herein described is not data compression, per se, expressing the additional capacity in terms of compression is a useful comparison.
(94) The term “data set” refers to a grouping of data for a particular purpose. One example of a data set might be a word processing file containing text and formatting information.
(95) The term “library” refers to a database containing sourceblocks each with a pattern of bits and reference code unique within that library. The term “codebook” is synonymous with the term library.
(96) The term “codeword” refers to the reference code form in which data is stored or transmitted in an aspect of the system. A codeword consists of a reference code to a sourceblock in the library plus an indication of that sourceblock's location in a particular data set.
(97) The term “dyadic distribution” refers to a probability distribution where all probabilities are of the form ½{circumflex over ( )}k for some integer k.
(98) The term “transformation matrix” refers to a mathematical construct used to map one probability distribution to another, typically represented as a two-dimensional array of numbers.
(99) The term “Huffman coding” refers to an entropy encoding algorithm used for lossless data compression, which assigns variable-length codes to input characters based on their frequencies of occurrence.
(100) The term “entropy encoding” refers to a form of lossless data compression that encodes information using fewer bits for more common symbols and more bits for less common symbols.
(101) The term “lossless mode” refers to an operating mode of the system where the original data can be perfectly reconstructed from the compressed data.
(102) The term “lossy mode” refers to an operating mode of the system where some data loss is accepted in exchange for greater compression ratios.
(103) The term “modified lossless mode” refers to an operating mode of the system where the main data stream and secondary data stream are processed or transmitted separately, allowing for perfect reconstruction when both streams are available.
(104) The term “feedback loop mechanism” refers to a system component that monitors the

performance of applied techniques, updates a knowledge base, and influences future processing decisions based on historical performance.

(105) The term "codeword" refers to a sequence of bits representing a specific data pattern or symbol in the context of data encoding or compression.

(106) Conceptual Architecture

(107) FIG. **1** is a diagram showing an embodiment **100** of the system in which all components of the system are operated locally. As incoming data **101** is received by data deconstruction engine **102**. Data deconstruction engine **102** breaks the incoming data into sourceblocks, which are then sent to library manager **103**. Using the information contained in sourceblock library lookup table **104** and sourceblock library storage **105**, library manager **103** returns reference codes to data deconstruction engine **102** for processing into codewords, which are stored in codeword storage **106**. When a data retrieval request **107** is received, data reconstruction engine **108** obtains the codewords associated with the data from codeword storage **106**, and sends them to library manager **103**. Library manager **103** returns the appropriate sourceblocks to data reconstruction engine **108**, which assembles them into the proper order and sends out the data in its original form **109**.

(108) FIG. **2** is a diagram showing an embodiment of one aspect **200** of the system, specifically data deconstruction engine **201**. Incoming data **202** is received by data analyzer **203**, which optimally analyzes the data based on machine learning algorithms and input **204** from a sourceblock size optimizer, which is disclosed below. Data analyzer may optionally have access to a sourceblock cache **205** of recently-processed sourceblocks, which can increase the speed of the system by avoiding processing in library manager **103**. Based on information from data analyzer **203**, the data is broken into sourceblocks by sourceblock creator **206**, which sends sourceblocks **207** to library manager **203** for additional processing. Data deconstruction engine **201** receives reference codes **208** from library manager **103**, corresponding to the sourceblocks in the library that match the sourceblocks sent by sourceblock creator **206**, and codeword creator **209** processes the reference codes into codewords comprising a reference code to a sourceblock and a location of that sourceblock within the data set. The original data may be discarded, and the codewords representing the data are sent out to storage **210**.

(109) FIG. **3** is a diagram showing an embodiment of another aspect of system **300**, specifically data reconstruction engine **301**. When a data retrieval request **302** is received by data request receiver **303** (in the form of a plurality of codewords corresponding to a desired final data set), it passes the information to data retriever **304**, which obtains the requested data **305** from storage. Data retriever **304** sends, for each codeword received, a reference codes from the codeword **306** to library manager **103** for retrieval of the specific sourceblock associated with the reference code. Data assembler **308** receives the sourceblock **307** from library manager **103** and, after receiving a plurality of sourceblocks corresponding to a plurality of codewords, assembles them into the proper order based on the location information contained in each codeword (recall each codeword comprises a sourceblock reference code and a location identifier that specifies where in the resulting data set the specific sourceblock should be restored to. The requested data is then sent to user **309** in its original form.

(110) FIG. **4** is a diagram showing an embodiment of another aspect of the system **400**, specifically library manager **401**. One function of library manager **401** is to generate reference codes from sourceblocks received from data deconstruction engine **301**. As sourceblocks are received **402** from data deconstruction engine **301**, sourceblock lookup engine **403** checks sourceblock library lookup table **404** to determine whether those sourceblocks already exist in sourceblock library storage **105**. If a particular sourceblock exists in sourceblock library storage **105**, reference code return engine **405** sends the appropriate reference code **406** to data deconstruction engine **301**. If the sourceblock does not exist in sourceblock library storage **105**, optimized reference code generator **407** generates a new, optimized reference code based on machine learning algorithms. Optimized reference code generator **407** then saves the reference code **408** to sourceblock library lookup table **104**; saves the

associated sourceblock **409** to sourceblock library storage **105**; and passes the reference code to reference code return engine **405** for sending **406** to data deconstruction engine **301**. Another function of library manager **401** is to optimize the size of sourceblocks in the system. Based on information **411** contained in sourceblock library lookup table **104**, sourceblock size optimizer **410** dynamically adjusts the size of sourceblocks in the system based on machine learning algorithms and outputs that information **412** to data analyzer **203**. Another function of library manager **401** is to return sourceblocks associated with reference codes received from data reconstruction engine **301**. As reference codes are received **414** from data reconstruction engine **301**, reference code lookup engine **413** checks sourceblock library lookup table **415** to identify the associated sourceblocks; passes that information to sourceblock retriever **416**, which obtains the sourceblocks **417** from sourceblock library storage **105**; and passes them 418 to data reconstruction engine **301**.
(111) FIG. **5** is a diagram showing another embodiment of system **500**, in which data is transferred between remote locations. As incoming data **501** is received by data deconstruction engine **502** at Location **1**, data deconstruction engine **301** breaks the incoming data into sourceblocks, which are then sent to library manager **503** at Location **1**. Using the information contained in sourceblock library lookup table **504** at Location **1** and sourceblock library storage **505** at Location **1**, library manager **503** returns reference codes to data deconstruction engine **301** for processing into codewords, which are transmitted **506** to data reconstruction engine **507** at Location **2**. In the case where the reference codes contained in a particular codeword have been newly generated by library manager **503** at Location **1**, the codeword is transmitted along with a copy of the associated sourceblock. As data reconstruction engine **507** at Location **2** receives the codewords, it passes them to library manager module **508** at Location **2**, which looks up the sourceblock in sourceblock library lookup table **509** at Location **2**, and retrieves the associated from sourceblock library storage **510**. Where a sourceblock has been transmitted along with a codeword, the sourceblock is stored in sourceblock library storage **510** and sourceblock library lookup table **504** is updated. Library manager **503** returns the appropriate sourceblocks to data reconstruction engine **507**, which assembles them into the proper order and sends the data in its original form **511**.
(112) FIG. **6** is a diagram showing an embodiment **600** in which a standardized version of a sourceblock library **603** and associated algorithms **604** would be encoded as firmware **602** on a dedicated processing chip **601** included as part of the hardware of a plurality of devices **600**. Contained on dedicated chip **601** would be a firmware area **602**, on which would be stored a copy of a standardized sourceblock library **603** and deconstruction/reconstruction algorithms **604** for processing the data. Processor **605** would have both inputs **606** and outputs **607** to other hardware on the device **600**. Processor **605** would store incoming data for processing on on-chip memory **608**, process the data using standardized sourceblock library **603** and deconstruction/reconstruction algorithms **604**, and send the processed data to other hardware on device **600**. Using this embodiment, the encoding and decoding of data would be handled by dedicated chip **601**, keeping the burden of data processing off device's **600** primary processors. Any device equipped with this embodiment would be able to store and transmit data in a highly optimized, bandwidth-efficient format with any other device equipped with this embodiment.
(113) FIG. **12** is a diagram showing an exemplary system architecture **1200**, according to a preferred embodiment of the invention. Incoming training data sets may be received at a customized library generator **1300** that processes training data to produce a customized word library **1201** comprising key-value pairs of data words (each comprising a string of bits) and their corresponding calculated binary Huffman codewords. The resultant word library **1201** may then be processed by a library optimizer **1400** to reduce size and improve efficiency, for example by pruning low-occurrence data entries or calculating approximate codewords that may be used to match more than one data word. A transmission encoder/decoder **1500** may be used to receive incoming data intended for storage or transmission, process the data using a word library **1201** to retrieve codewords for the words in the incoming data, and then append the codewords (rather than

the original data) to an outbound data stream. Each of these components is described in greater detail below, illustrating the particulars of their respective processing and other functions, referring to FIGS. **2-4**.

(114) System **1200** provides near-instantaneous source coding that is dictionary-based and learned in advance from sample training data, so that encoding and decoding may happen concurrently with data transmission. This results in computational latency that is near zero, but the data size reduction is comparable to classical compression. For example, if N bits are to be transmitted from sender to receiver, the compression ratio of classical compression is C, the ratio between the deflation factor of system **1200** and that of multi-pass source coding is p, the classical compression encoding rate is R.sub.C bit/s and the decoding rate is R.sub.D bit/s, and the transmission speed is S bit/s, the compress-send-decompress time will be

(115) $T_{old} = \frac{N}{R_C} + \frac{N}{CS} + \frac{N}{CR_D}$

while the transmit-while-coding time for system **1200** will be (assuming that encoding and decoding happen at least as quickly as network latency):

(116) $T_{new} = \frac{N_p}{CS}$

so that the total data transit time improvement factor is

(117) $\frac{T_{old}}{T_{new}} = \frac{\frac{CS}{R_C} + 1 + \frac{S}{R_D}}{p}$

which presents a savings whenever

(118) $\frac{CS}{R_C} + \frac{S}{R_D} > p - 1.$

This is a reasonable scenario given that typical values in real-world practice are C=0.32, R.sub.C=1.1.Math.10.sup.12, R.sub.D=4.2.Math.10.sup.12, S=10.sup.11, giving

(119) $\frac{CS}{R_C} + \frac{S}{R_D} = 0.053.$

such that system **1200** will outperform the total transit time of the best compression technology available as long as its deflation factor is no more than 5% worse than compression. Such customized dictionary-based encoding will also sometimes exceed the deflation ratio of classical compression, particularly when network speeds increase beyond 100 Gb/s.

(120) The delay between data creation and its readiness for use at a receiving end will be equal to only the source word length t (typically 5-15 bytes), divided by the deflation factor C/p and the network speed

(121) $S, i.e. \text{delay}_{\text{invention}} = \frac{tp}{CS}$

since encoding and decoding occur concurrently with data transmission. On the other hand, the latency associated with classical compression is

(122) $\text{delay}_{\text{priorart}} = \frac{N}{R_C} + \frac{N}{CS} + \frac{N}{CR_D}$

where N is the packet/file size. Even with the generous values chosen above as well as N=512K, t=10, and p=1.05, this results in delay.sub.invention≈3.3.Math.10.sup.−10 while delay.sub.priorart≈1.3.Math.10.sup.−7, a more than 400-fold reduction in latency.

(123) A key factor in the efficiency of Huffman coding used by system **1200** is that key-value pairs be chosen carefully to minimize expected coding length, so that the average deflation/compression ratio is minimized. It is possible to achieve the best possible expected code length among all instantaneous codes using Huffman codes if one has access to the exact probability distribution of source words of a given desired length from the random variable generating them. In practice this is impossible, as data is received in a wide variety of formats and the random processes underlying the source data are a mixture of human input, unpredictable (though in principle, deterministic) physical events, and noise. System **1200** addresses this by restriction of data types and density estimation; training data is provided that is representative of the type of data anticipated in “real-world” use of system **1200**, which is then used to model the distribution of binary strings in the data in order to build a Huffman code word library **1200**.

(124) FIG. **13** is a diagram showing a more detailed architecture for a customized library generator

1300. When an incoming training data set **1301** is received, it may be analyzed using a frequency creator **1302** to analyze for word frequency (that is, the frequency with which a given word occurs in the training data set). Word frequency may be analyzed by scanning all substrings of bits and directly calculating the frequency of each substring by iterating over the data set to produce an occurrence frequency, which may then be used to estimate the rate of word occurrence in non-training data. A first Huffman binary tree is created based on the frequency of occurrences of each word in the first dataset, and a Huffman codeword is assigned to each observed word in the first dataset according to the first Huffman binary tree. Machine learning may be utilized to improve results by processing a number of training data sets and using the results of each training set to refine the frequency estimations for non-training data, so that the estimation yield better results when used with real-world data (rather than, for example, being only based on a single training data set that may not be very similar to a received non-training data set). A second Huffman tree creator **1303** may be utilized to identify words that do not match any existing entries in a word library **1201** and pass them to a hybrid encoder/decoder **1304**, that then calculates a binary Huffman codeword for the mismatched word and adds the codeword and original data to the word library **1201** as a new key-value pair. In this manner, customized library generator **1300** may be used both to establish an initial word library **1201** from a first training set, as well as expand the word library **1201** using additional training data to improve operation.

(125) FIG. **14** is a diagram showing a more detailed architecture for a library optimizer **1400**. A pruner **1401** may be used to load a word library **1201** and reduce its size for efficient operation, for example by sorting the word library **1201** based on the known occurrence probability of each key-value pair and removing low-probability key-value pairs based on a loaded threshold parameter. This prunes low-value data from the word library to trim the size, eliminating large quantities of very-low-frequency key-value pairs such as single-occurrence words that are unlikely to be encountered again in a data set. Pruning eliminates the least-probable entries from word library **1201** up to a given threshold, which will have a negligible impact on the deflation factor since the removed entries are only the least-common ones, while the impact on word library size will be larger because samples drawn from asymptotically normal distributions (such as the log-probabilities of words generated by a probabilistic finite state machine, a model well-suited to a wide variety of real-world data) which occur in tails of the distribution are disproportionately large in counting measure. A delta encoder **1402** may be utilized to apply delta encoding to a plurality of words to store an approximate codeword as a value in the word library, for which each of the plurality of source words is a valid corresponding key. This may be used to reduce library size by replacing numerous key-value pairs with a single entry for the approximate codeword and then represent actual codewords using the approximate codeword plus a delta value representing the difference between the approximate codeword and the actual codeword. Approximate coding is optimized for low-weight sources such as Golomb coding, run-length coding, and similar techniques. The approximate source words may be chosen by locality-sensitive hashing, so as to approximate Hamming distance without incurring the intractability of nearest-neighbor-search in Hamming space. A parametric optimizer **1403** may load configuration parameters for operation to optimize the use of the word library **1201** during operation. Best-practice parameter/hyperparameter optimization strategies such as stochastic gradient descent, quasi-random grid search, and evolutionary search may be used to make optimal choices for all interdependent settings playing a role in the functionality of system **1200**. In cases where lossless compression is not required, the delta value may be discarded at the expense of introducing some limited errors into any decoded (reconstructed) data.

(126) FIG. **15** is a diagram showing a more detailed architecture for a transmission encoder/decoder **1500**. According to various arrangements, transmission encoder/decoder **1500** may be used to deconstruct data for storage or transmission, or to reconstruct data that has been received, using a word library **1201**. A library comparator **1501** may be used to receive data

comprising words or codewords, and compare against a word library **1201** by dividing the incoming stream into substrings of length t and using a fast hash to check word library **1201** for each substring. If a substring is found in word library **1201**, the corresponding key/value (that is, the corresponding source word or codeword, according to whether the substring used in comparison was itself a word or codeword) is returned and appended to an output stream. If a given substring is not found in word library **1201**, a mismatch handler **1502** and hybrid encoder/decoder **1503** may be used to handle the mismatch similarly to operation during the construction or expansion of word library **1201**. A mismatch handler **1502** may be utilized to identify words that do not match any existing entries in a word library **1201** and pass them to a hybrid encoder/decoder **1503**, that then calculates a binary Huffman codeword for the mismatched word and adds the codeword and original data to the word library **1201** as a new key-value pair. The newly-produced codeword may then be appended to the output stream. In arrangements where a mismatch indicator is included in a received data stream, this may be used to preemptively identify a substring that is not in word library **1201** (for example, if it was identified as a mismatch on the transmission end), and handled accordingly without the need for a library lookup.

(127) FIG. **19** is an exemplary system architecture of a data encoding system used for cyber security purposes. Much like in FIG. **1**, incoming data **101** to be deconstructed is sent to a data deconstruction engine **102**, which may attempt to deconstruct the data and turn it into a collection of codewords using a library manager **103**. Codeword storage **106** serves to store unique codewords from this process, and may be queried by a data reconstruction engine **108** which may reconstruct the original data from the codewords, using a library manager **103**. However, a cybersecurity gateway **1900** is present, communicating in-between a library manager **103** and a deconstruction engine **102**, and containing an anomaly detector **1910** and distributed denial of service (DDOS) detector **1920**. The anomaly detector examines incoming data to determine whether there is a disproportionate number of incoming reference codes that do not match reference codes in the existing library. A disproportionate number of non-matching reference codes may indicate that data is being received from an unknown source, of an unknown type, or contains unexpected (possibly malicious) data. If the disproportionate number of non-matching reference codes exceeds an established threshold or persists for a certain length of time, the anomaly detector **1910** raises a warning to a system administrator. Likewise, the DDOS detector **1920** examines incoming data to determine whether there is a disproportionate amount of repetitive data. A disproportionate amount of repetitive data may indicate that a DDOS attack is in progress. If the disproportionate amount of repetitive data exceeds an established threshold or persists for a certain length of time, the DDOS detector **1910** raises a warning to a system administrator. In this way, a data encoding system may detect and warn users of, or help mitigate, common cyber-attacks that result from a flow of unexpected and potentially harmful data, or attacks that result from a flow of too much irrelevant data meant to slow down a network or system, as in the case of a DDOS attack.

(128) FIG. **22** is an exemplary system architecture of a data encoding system used for data mining and analysis purposes. Much like in FIG. **1**, incoming data **101** to be deconstructed is sent to a data deconstruction engine **102**, which may attempt to deconstruct the data and turn it into a collection of codewords using a library manager **103**. Codeword storage **106** serves to store unique codewords from this process, and may be queried by a data reconstruction engine **108** which may reconstruct the original data from the codewords, using a library manager **103**. A data analysis engine **2210**, typically operating while the system is otherwise idle, sends requests for data to the data reconstruction engine **108**, which retrieves the codewords representing the requested data from codeword storage **106**, reconstructs them into the data represented by the codewords, and send the reconstructed data to the data analysis engine **2210** for analysis and extraction of useful data (i.e., data mining). Because the speed of reconstruction is significantly faster than decompression using traditional compression technologies (i.e., significantly less decompression latency), this approach makes data mining feasible. Very often, data stored using traditional compression is not mined

precisely because decompression lag makes it unfeasible, especially during shorter periods of system idleness. Increasing the speed of data reconstruction broadens the circumstances under which data mining of stored data is feasible.

(129) FIG. **24** is an exemplary system architecture of a data encoding system used for remote software and firmware updates. Software and firmware updates typically require smaller, but more frequent, file transfers. A server which hosts a software or firmware update **2410** may host an encoding-decoding system **2420**, allowing for data to be encoded into, and decoded from, sourceblocks or codewords, as disclosed in previous figures. Such a server may possess a software update, operating system update, firmware update, device driver update, or any other form of software update, which in some cases may be minor changes to a file, but nevertheless necessitate sending the new, completed file to the recipient. Such a server is connected over a network **2430**, which is further connected to a recipient computer **2440**, which may be connected to a server **2410** for receiving such an update to its system. In this instance, the recipient device **2440** also hosts the encoding and decoding system **2450**, along with a codebook or library of reference codes that the hosting server **2410** also shares. The updates are retrieved from storage at the hosting server **2410** in the form of codewords, transferred over the network **2430** in the form of codewords, and reconstructed on the receiving computer **2440**. In this way, a far smaller file size, and smaller total update size, may be sent over a network. The receiving computer **2440** may then install the updates on any number of target computing devices **2460***a-n*, using a local network or other high-bandwidth connection.

(130) FIG. **26** is an exemplary system architecture of a data encoding system used for large-scale software installation such as operating systems. Large-scale software installations typically require very large, but infrequent, file transfers. A server which hosts an installable software **2610** may host an encoding-decoding system **2620**, allowing for data to be encoded into, and decoded from, sourceblocks or codewords, as disclosed in previous figures. The files for the large scale software installation are hosted on the server **2610**, which is connected over a network **2630** to a recipient computer **2640**. In this instance, the encoding and decoding system **2650***a-n* is stored on or connected to one or more target devices **2660***a-n*, along with a codebook or library of reference codes that the hosting server **2610** shares. The software is retrieved from storage at the hosting server **2610** in the form of codewords, and transferred over the network **2630** in the form of codewords to the receiving computer **2640**. However, instead of being reconstructed at the receiving computer **2640**, the codewords are transmitted to one or more target computing devices, and reconstructed and installed directly on the target devices **2660***a-n*. In this way, a far smaller file size, and smaller total update size, may be sent over a network or transferred between computing devices, even where the network **2630** between the receiving computer **2640** and target devices **2660***a-n* is low bandwidth, or where there are many target devices **2660***a-n*.

(131) FIG. **28** is a block diagram of an exemplary system architecture **2800** of a codebook training system for a data encoding system, according to an embodiment. According to this embodiment, two separate machines may be used for encoding **2810** and decoding **2820**. Much like in FIG. **1**, incoming data **101** to be deconstructed is sent to a data deconstruction engine **102** residing on encoding machine **2810**, which may attempt to deconstruct the data and turn it into a collection of codewords using a library manager **103**. Codewords may be transmitted **2840** to a data reconstruction engine **108** residing on decoding machine **2820**, which may reconstruct the original data from the codewords, using a library manager **103**. However, according to this embodiment, a codebook training module **2830** is present on the decoding machine **2810**, communicating in-between a library manager **103** and a deconstruction engine **102**. According to other embodiments, codebook training module **2830** may reside instead on decoding machine **2820** if the machine has enough computing resources available; which machine the module **2830** is located on may depend on the system user's architecture and network structure. Codebook training module **2830** may send requests for data to the data reconstruction engine **2810**, which routes incoming data **101** to

codebook training module **2830**. Codebook training module **2830** may perform analyses on the requested data in order to gather information about the distribution of incoming data **101** as well as monitor the encoding/decoding model performance. Additionally, codebook training module **2830** may also request and receive device data **2860** to supervise network connected devices and their processes and, according to some embodiments, to allocate training resources when requested by devices running the encoding system. Devices may include, but are not limited to, encoding and decoding machines, training machines, sensors, mobile compute devices, and Internet-of-things ("IoT") devices. Based on the results of the analyses, the codebook training module **2830** may create a new training dataset from a subset of the requested data in order to counteract the effects of data drift on the encoding/decoding models, and then publish updated **2850** codebooks to both the encoding machine **2810** and decoding machine **2820**.

(132) FIG. **29** is a block diagram of an exemplary architecture for a codebook training module **2900**, according to an embodiment. According to the embodiment, a data collector **2910** is present which may send requests for incoming data **2905** to a data deconstruction engine **102** which may receive the request and route incoming data to codebook training module **2900** where it may be received by data collector **2910**. Data collector **2910** may be configured to request data periodically such as at schedule time intervals, or for example, it may be configured to request data after a certain amount of data has been processed through the encoding machine **2810** or decoding machine **2820**. The received data may be a plurality of sourceblocks, which are a series of binary digits, originating from a source packet otherwise referred to as a datagram. The received data may be compiled into a test dataset and temporarily stored in a cache **2970**. Once stored, the test dataset may be forwarded to a statistical analysis engine **2920** which may utilize one or more algorithms to determine the probability distribution of the test dataset. Best-practice probability distribution metrics such as Kullback-Leibler divergence, adaptive windowing, and Jensen-Shannon divergence may be used to compute and/or estimate the probability distribution of training and test datasets. These metrics may also be used to estimate the probability distribution from the current run-time data. In some implementations, the estimate of the training data may be compared against the estimate of the run-time data to verify if the difference in calculated distributions exceeds a predetermined difference threshold. If the difference in distributions does not exceed the difference threshold, that indicates the test dataset, and therefore the incoming data, has not experienced enough data drift to cause the encoding/decoding system performance to degrade significantly, which indicates that no updates are necessary to the existing codebooks. However, if the difference threshold has been surpassed, then the data drift is significant enough to cause the encoding/decoding system performance to degrade to the point where the existing models and accompanying codebooks need to be updated. A monitoring database **2930** may be used to store a variety of statistical data related to training datasets and model performance metrics in one place to facilitate quick and accurate system monitoring capabilities as well as assist in system debugging functions. For example, the original or current training dataset and the calculated probability distribution of this training dataset used to develop the current encoding and decoding algorithms may be stored in monitor database **2930**.

(133) Since data drifts involve statistical change in the data, the best approach to detect drift is by monitoring the incoming data's statistical properties, the model's predictions, and their correlation with other factors. After statistical analysis engine **2920** calculates the probability distribution of the test dataset it may retrieve from monitor database **2930** the calculated and stored probability distribution of the current training dataset. It may then compare the two probability distributions of the two different datasets in order to verify if the difference in calculated distributions exceeds a predetermined difference threshold. If the difference in distributions does not exceed the difference threshold, that indicates the test dataset, and therefore the incoming data, has not experienced enough data drift to cause the encoding/decoding system performance to degrade significantly, which indicates that no updates are necessary to the existing codebooks. However, if the difference

threshold has been surpassed, then the data drift is significant enough to cause the encoding/decoding system performance to degrade to the point where the existing models and accompanying codebooks need to be updated. According to an embodiment, an alert may be generated by statistical analysis engine **2920** if the difference threshold is surpassed or if otherwise unexpected behavior arises.

(134) In the event that an update is required, the test dataset stored in the cache **2970** and its associated calculated probability distribution may be sent to monitor database **2930** for long term storage. This test dataset may be used as a new training dataset to retrain the encoding and decoding algorithms **2940** used to create new sourceblocks based upon the changed probability distribution. The new sourceblocks may be sent out to a library manager **2915** where the sourceblocks can be assigned new codewords. Each new sourceblock and its associated codeword may then be added to a new codebook and stored in a storage device. The new and updated codebook may then be sent back **2925** to codebook training module **2900** and received by a codebook update engine **2950**. Codebook update engine **2950** may temporarily store the received updated codebook in the cache **2970** until other network devices and machines are ready, at which point codebook update engine **2950** will publish the updated codebooks **2945** to the necessary network devices.

(135) A network device manager **2960** may also be present which may request and receive network device data **2935** from a plurality of network connected devices and machines. When the disclosed encoding system and codebook training system **2800** are deployed in a production environment, upstream process changes may lead to data drift, or other unexpected behavior. For example, a sensor being replaced that changes the units of measurement from inches to centimeters, data quality issues such as a broken sensor always reading 0, and covariate shift which occurs when there is a change in the distribution of input variables from the training set. These sorts of behavior and issues may be determined from the received device data **2935** in order to identify potential causes of system error that is not related to data drift and therefore does not require an updated codebook. This can save network resources from being unnecessarily used on training new algorithms as well as alert system users to malfunctions and unexpected behavior devices connected to their networks. Network device manager **2960** may also utilize device data **2935** to determine available network resources and device downtime or periods of time when device usage is at its lowest. Codebook update engine **2950** may request network and device availability data from network device manager **2960** in order to determine the most optimal time to transmit updated codebooks (i.e., trained libraries) to encoder and decoder devices and machines.

(136) FIG. **30** is a block diagram of another embodiment of the codebook training system using a distributed architecture and a modified training module. According to an embodiment, there may be a server which maintains a master supervisory process over remote training devices hosting a master training module **3010** which communicates via a network **3020** to a plurality of connected network devices **3030***a-n*. The server may be located at the remote training end such as, but not limited to, cloud-based resources, a user-owned data center, etc. The master training module located on the server operates similarly to the codebook training module disclosed in FIG. **29** above, however, the server **3010** utilizes the master training module via the network device manager **2960** to farm out training resources to network devices **3030***a-n*. The server **3010** may allocate resources in a variety of ways, for example, round-robin, priority-based, or other manner, depending on the user needs, costs, and number of devices running the encoding/decoding system. Server **3010** may identify elastic resources which can be employed if available to scale up training when the load becomes too burdensome. On the network devices **3030***a-n* may be present a lightweight version of the training module **3040** that trades a little suboptimality in the codebook for training on limited machinery and/or makes training happen in low-priority threads to take advantage of idle time. In this way the training of new encoding/decoding algorithms may take place in a distributed manner which allows data gathering or generating devices to process and

train on data gathered locally, which may improve system latency and optimize available network resources.

(137) FIG. **36** is a system diagram illustrating a data deconstruction engine **201** containing a compression engine that may be used for pre-compression of high-entropy data streams before being compacted using codebook techniques, according to an embodiment. Incoming data **202** is received by data analyzer **203**, which optimally analyzes the data based on machine learning algorithms and input **204** from a sourceblock size optimizer, which is disclosed below. Data analyzer may optionally have access to a sourceblock cache **205** of recently-processed sourceblocks, which can increase the speed of the system by avoiding processing in library manager **103**. Based on information from data analyzer **203**, the data is broken into sourceblocks by sourceblock creator **206**, which sends sourceblocks **207** to library manager **203** for additional processing. Data deconstruction engine **201** receives reference codes **208** from library manager **103**, corresponding to the sourceblocks in the library that match the sourceblocks sent by sourceblock creator **206**, and codeword creator **209** processes the reference codes into codewords comprising a reference code to a sourceblock and a location of that sourceblock within the data set. The original data may be discarded, and the codewords representing the data are sent out to storage **210**.

(138) Before sending codewords to storage, a compression engine **3610** exists which may be forwarded image data, or other high-entropy data, from a data analyzer **203**, for pre-compression of the data before it is turned into sourceblocks by a sourceblock creator **206**. When fed in an image, the original image **3611** has a forward transform **3612** or other mathematical transform such as Discrete Cosine Transforms (“DCT”), which include the Fast Fourier Transform (“FFT”), applied to it, to transform the image into numeric data suitable for compression. Upon having a DCT applied to the image **3611**, **3612**, it is quantized **3613**, a process common in compression technologies in which integers in the numeric data of an image are divided by a quantization matrix, then rounded to the nearest integer, reducing their precision and subsequently the number of bits needed to store the integers. In this manner, this compression is considered lossy, as there is irreversible data loss due to the loss in precision of these integers. The selection of quantization matrix values may be undertaken with numerous methods, commonly including those that use machine learning to find a set of values that provide for suitable compression without an unacceptable amount of loss of data, or manually specifying DCT coefficients and generating the matrix from those values. Upon quantization **3613** application within a compression engine **3610**, an algorithm for entropy encoding **3614** may be selected and applied, such as Huffman encoding, Shannon encoding, or other encoding methods. It should be readily apparent that any high entropy signal data such as audio or video may also be compressed in this manner.

(139) FIG. **37** is a system diagram illustrating a data reconstruction engine **301** containing a decompression engine that may be used for decompressing high-entropy data streams after they have been de-compacted from their codebook-translated format, according to an embodiment. When a data retrieval request **302** is received by data request receiver **303** (in the form of a plurality of codewords corresponding to a desired final data set), it passes the information to data retriever **304**, which obtains the requested data **305** from storage. Data retriever **304** sends, for each codeword received, a reference codes from the codeword **306** to library manager **103** for retrieval of the specific sourceblock associated with the reference code. Data assembler **308** receives the sourceblock **307** from library manager **103** and, after receiving a plurality of sourceblocks corresponding to a plurality of codewords, assembles them into the proper order based on the location information contained in each codeword (recall each codeword comprises a sourceblock reference code and a location identifier that specifies where in the resulting data set the specific sourceblock should be restored to. The requested data is then sent to user **309** in its original form.

(140) Before sending the data to a user in its current form, a decompression engine **3710** exists as a way to decompress data that, after having been reassembled from sourceblocks by a data assembler

308, is still in a recognized compressed format, rather than a raw or original form. First, any entropy encoding **3711** such as Huffman encoding or Shannon encoding, is reversed, which are typically lossless compression methods and do not result in data degradation. Dequantization **3712** may then be performed by the engine, whereby the integers making up the data are multiplied by a DCT matrix, with precision loss due to rounding in a quantization step during compression. If dequantization is needed for decompression, this would result in lossy compression/decompression. An inverse transform **3713** may be applied to reverse the transformation from signal data to numeric data, restoring the existing signal data that may not be represented by numeric data, resulting in a restored image **3714** or other high entropy data. It should be readily apparent that any high entropy signal data such as audio or video may also be decompressed in this manner.
(141) FIG. **38** is a method diagram illustrating the operation of a compression engine within a data deconstruction engine, according to an embodiment. Data analysis of an incoming data stream may be used to detect high-entropy data, such as image data **3810**, but also potentially audio or video data, indicating data that may benefit from lossy pre-compression before codebook compaction **3820**, thereby reducing its data size even further and enabling more technologies such as progressive image loading to take place. In order to compress the data, a DCT may be applied **3830**, converting signal data of some sort into numeric data. In this way, raw signal data such as that received from physical sensors may be converted into numeric data ready for processing and further compression. The numeric data may be quantized, a process in which the integers representing the data are divided by a quantization matrix and then rounded down, reducing precision of the integers and essentially resulting in the loss of “noise” data, but also the compaction of the data by reducing the actual number of bits needed to store the data **3840**. In addition to quantization, an entropy encoding algorithm may be applied to the data **3850**, such as Huffman, Shannon, or arithmetic coding. When these steps, together or in some limited combination, are completed, the compressed data may be returned to a data analyzer that may then direct the newly compressed data for codebook compaction **3860**, which is a lossless process by comparison. The data may then be compacted using codebook methods **3870** described elsewhere.
(142) FIG. **39** is a method diagram illustrating the operation of a decompression engine within a data reconstruction engine, according to an embodiment. To reverse a lossy compression attained by a compression engine, a data reconstruction engine must operate in reverse order from a data deconstruction engine, first un-compacting the data from its codebook representation **3910** as described elsewhere, before proceeding with decompressing the potentially lossy precompression that was performed on the data prior to codebook compaction. For decompression of the remaining data, the decompression engine first reversing the entropy coding algorithm applied **3920**, before dequantization of the data if it was quantized **3930**, and reversing the DCT previously applied **3940** to transform the compressed data back into its original form, such as image, audio, video, or other signal data. The now decompressed high-entropy data is then able to be sent to a receiving user or system, decompressed and expanded from its pre-compressed and codebook format **3950**.
(143) FIG. **40** is a message flow diagram illustrating the use of pre-compression of high-entropy data streams before being compacted using codebook techniques, transmitted to a data reconstruction engine, expanded from their codebook format, and decompressed, before being transmitted to a receiver or end-user, according to an embodiment. Components that are transmitting or receiving data, or otherwise communicating with each other, include a data deconstruction engine **201**, a compression engine **3610**, a data reconstruction engine **301**, a decompression engine **3710**, a sender **4010** (such as a user trying to compact and compress data they wish to send), and a receiver **4011** (such as the destination for the data that the sender **4010** is attempting to send). A data sender **4010** first begins the process of preparing their data for transmission, with a data deconstruction engine **201**, **4020**. This may be done with minimal user interaction, for example a physical sensor system may already be configured to operate a data deconstruction engine **201** as part of its normal operations, and from a sender's **4010** perspective,

all they are doing is turning on or otherwise using the device normally, without knowledge or configuration of the data compression or compaction techniques therein. When the data deconstruction engine **201** detects high-entropy data such as image, audio, video, or similar data, it may use a compression engine **3610** to process that data **4030**, before receiving it back in a compressed state **4040**. That data, newly compressed by the compression engine **3610** and then otherwise compacted by the data deconstruction engine **201** into compact codebook translations that will be understood by a receiver **4011**, may then be sent to a data reconstruction engine **301** operated by a receiver **4011**, **4050**, and this data transmission may take the form of progressive resolution, progressive compression, or more commonly known progressive image loading, techniques. For instance the data stream may contain a highly compressed resolution of the original data, with a high degree of signal loss due to lossy compression, within or while also transmitting a more complete or losslessly compressed or compacted data stream.

(144) The data, when received by a data reconstruction engine **301**, is first decompacted and translated from its codebook form, before it can then be sent to a decompression engine **3710**, **4060** to be decompressed from its still compressed form. This is due to the fact that the original data was first pre-compressed before being codebook compacted, therefore the decompression must take place after the codebook compaction has been reversed. This decompressed data is forwarded back to the data reconstruction engine **301**, **4070** to finalize reconstructing and assembling the data, that is then sent or presented to the data receiver **4011**, **4080**. In this way, highly compacted and compressed data transmission may take place, including with the use of progressive image loading or similar techniques, depending on how the data is encoded in a particular implementation of data encoding.

(145) Description of Method Aspects

(146) Since the library consists of re-usable building sourceblocks, and the actual data is represented by reference codes to the library, the total storage space of a single set of data would be much smaller than conventional methods, wherein the data is stored in its entirety. The more data sets that are stored, the larger the library becomes, and the more data can be stored in reference code form.

(147) As an analogy, imagine each data set as a collection of printed books that are only occasionally accessed. The amount of physical shelf space required to store many collections would be quite large, and is analogous to conventional methods of storing every single bit of data in every data set. Consider, however, storing all common elements within and across books in a single library, and storing the books as references codes to those common elements in that library. As a single book is added to the library, it will contain many repetitions of words and phrases. Instead of storing the whole words and phrases, they are added to a library, and given a reference code, and stored as reference codes. At this scale, some space savings may be achieved, but the reference codes will be on the order of the same size as the words themselves. As more books are added to the library, larger phrases, quotations, and other words patterns will become common among the books. The larger the word patterns, the smaller the reference codes will be in relation to them as not all possible word patterns will be used. As entire collections of books are added to the library, sentences, paragraphs, pages, or even whole books will become repetitive. There may be many duplicates of books within a collection and across multiple collections, many references and quotations from one book to another, and much common phraseology within books on particular subjects. If each unique page of a book is stored only once in a common library and given a reference code, then a book of 1,000 pages or more could be stored on a few printed pages as a string of codes referencing the proper full-sized pages in the common library. The physical space taken up by the books would be dramatically reduced. The more collections that are added, the greater the likelihood that phrases, paragraphs, pages, or entire books will already be in the library, and the more information in each collection of books can be stored in reference form. Accessing entire collections of books is then limited not by physical shelf space, but by the ability to reprint

and recycle the books as needed for use.

(148) The projected increase in storage capacity using the method herein described is primarily dependent on two factors: 1) the ratio of the number of bits in a block to the number of bits in the reference code, and 2) the amount of repetition in data being stored by the system.

(149) With respect to the first factor, the number of bits used in the reference codes to the sourceblocks must be smaller than the number of bits in the sourceblocks themselves in order for any additional data storage capacity to be obtained. As a simple example, 16-bit sourceblocks would require 216, or 65536, unique reference codes to represent all possible patterns of bits. If all possible 65536 blocks patterns are utilized, then the reference code itself would also need to contain sixteen bits in order to refer to all possible 65,536 blocks patterns. In such case, there would be no storage savings. However, if only 16 of those block patterns are utilized, the reference code can be reduced to 4 bits in size, representing an effective compression of 4 times (16 bits/4 bits=4) versus conventional storage. Using a typical block size of 512 bytes, or 4,096 bits, the number of possible block patterns is 24,096, which for all practical purposes is unlimited. A typical hard drive contains one terabyte (TB) of physical storage capacity, which represents 1,953,125,000, or roughly 231, 512 byte blocks. Assuming that 1 TB of unique 512-byte sourceblocks were contained in the library, and that the reference code would thus need to be 31 bits long, the effective compression ratio for stored data would be on the order of 132 times (4,096/31≈132) that of conventional storage.

(150) With respect to the second factor, in most cases it could be assumed that there would be sufficient repetition within a data set such that, when the data set is broken down into sourceblocks, its size within the library would be smaller than the original data. However, it is conceivable that the initial copy of a data set could require somewhat more storage space than the data stored in a conventional manner, if all or nearly all sourceblocks in that set were unique. For example, assuming that the reference codes are 1/10.sup.th the size of a full-sized copy, the first copy stored as sourceblocks in the library would need to be 1.1 megabytes (MB), (1 MB for the complete set of full-sized sourceblocks in the library and 0.1 MB for the reference codes). However, since the sourceblocks stored in the library are universal, the more duplicate copies of something you save, the greater efficiency versus conventional storage methods. Conventionally, storing 10 copies of the same data requires 10 times the storage space of a single copy. For example, ten copies of a 1 MB file would take up 10 MB of storage space. However, using the method described herein, only a single full-sized copy is stored, and subsequent copies are stored as reference codes. Each additional copy takes up only a fraction of the space of the full-sized copy. For example, again assuming that the reference codes are 1/10.sup.th the size of the full-size copy, ten copies of a 1 MB file would take up only 2 MB of space (1 MB for the full-sized copy, and 0.1 MB each for ten sets of reference codes). The larger the library, the more likely that part or all of incoming data will duplicate sourceblocks already existing in the library.

(151) The size of the library could be reduced in a manner similar to storage of data. Where sourceblocks differ from each other only by a certain number of bits, instead of storing a new sourceblock that is very similar to one already existing in the library, the new sourceblock could be represented as a reference code to the existing sourceblock, plus information about which bits in the new block differ from the existing block. For example, in the case where 512 byte sourceblocks are being used, if the system receives a new sourceblock that differs by only one bit from a sourceblock already existing in the library, instead of storing a new 512 byte sourceblock, the new sourceblock could be stored as a reference code to the existing sourceblock, plus a reference to the bit that differs. Storing the new sourceblock as a reference code plus changes would require only a few bytes of physical storage space versus the 512 bytes that a full sourceblock would require. The algorithm could be optimized to store new sourceblocks in this reference code plus changes form unless the changes portion is large enough that it is more efficient to store a new, full sourceblock.

(152) It will be understood by one skilled in the art that transfer and synchronization of data would

be increased to the same extent as for storage. By transferring or synchronizing reference codes instead of full-sized data, the bandwidth requirements for both types of operations are dramatically reduced.

(153) In addition, the method described herein is inherently a form of encryption. When the data is converted from its full form to reference codes, none of the original data is contained in the reference codes. Without access to the library of sourceblocks, it would be impossible to re-construct any portion of the data from the reference codes. This inherent property of the method described herein could obviate the need for traditional encryption algorithms, thereby offsetting most or all of the computational cost of conversion of data back and forth to reference codes. In theory, the method described herein should not utilize any additional computing power beyond traditional storage using encryption algorithms. Alternatively, the method described herein could be in addition to other encryption algorithms to increase data security even further.

(154) In other embodiments, additional security features could be added, such as: creating a proprietary library of sourceblocks for proprietary networks, physical separation of the reference codes from the library of sourceblocks, storage of the library of sourceblocks on a removable device to enable easy physical separation of the library and reference codes from any network, and incorporation of proprietary sequences of how sourceblocks are read and the data reassembled.

(155) FIG. **7** is a diagram showing an example of how data might be converted into reference codes using an aspect of an embodiment **700**. As data is received **701**, it is read by the processor in sourceblocks of a size dynamically determined by the previously disclosed sourceblock size optimizer **410**. In this example, each sourceblock is 16 bits in length, and the library **702** initially contains three sourceblocks with reference codes 00, 01, and 10. The entry for reference code 11 is initially empty. As each 16 bit sourceblock is received, it is compared with the library. If that sourceblock is already contained in the library, it is assigned the corresponding reference code. So, for example, as the first line of data (0000 0011 0000 0000) is received, it is assigned the reference code (01) associated with that sourceblock in the library. If that sourceblock is not already contained in the library, as is the case with the third line of data (0000 1111 0000 0000) received in the example, that sourceblock is added to the library and assigned a reference code, in this case **11**. The data is thus converted **703** to a series of reference codes to sourceblocks in the library. The data is stored as a collection of codewords, each of which contains the reference code to a sourceblock and information about the location of the sourceblocks in the data set. Reconstructing the data is performed by reversing the process. Each stored reference code in a data collection is compared with the reference codes in the library, the corresponding sourceblock is read from the library, and the data is reconstructed into its original form.

(156) FIG. **8** is a method diagram showing the steps involved in using an embodiment **800** to store data. As data is received **801**, it would be deconstructed into sourceblocks **802**, and passed **803** to the library management module for processing. Reference codes would be received back **804** from the library management module, and could be combined with location information to create codewords **805**, which would then be stored **806** as representations of the original data.

(157) FIG. **9** is a method diagram showing the steps involved in using an embodiment **900** to retrieve data. When a request for data is received **901**, the associated codewords would be retrieved **902** from the library. The codewords would be passed **903** to the library management module, and the associated sourceblocks would be received back **904**. Upon receipt, the sourceblocks would be assembled **905** into the original data using the location data contained in the codewords, and the reconstructed data would be sent out **906** to the requestor.

(158) FIG. **10** is a method diagram showing the steps involved in using an embodiment **1000** to encode data. As sourceblocks are received **1001** from the deconstruction engine, they would be compared **1002** with the sourceblocks already contained in the library. If that sourceblock already exists in the library, the associated reference code would be returned **1005** to the deconstruction engine. If the sourceblock does not already exist in the library, a new reference code would be

created **1003** for the sourceblock. The new reference code and its associated sourceblock would be stored **1004** in the library, and the reference code would be returned to the deconstruction engine.
(159) FIG. **11** is a method diagram showing the steps involved in using an embodiment **1100** to decode data. As reference codes are received **1101** from the reconstruction engine, the associated sourceblocks are retrieved **1102** from the library, and returned **1103** to the reconstruction engine.
(160) FIG. **16** is a method diagram illustrating key system functionality utilizing an encoder and decoder pair, according to a preferred embodiment. In a first step **1601**, at least one incoming data set may be received at a customized library generator **1300** that then **1602** processes data to produce a customized word library **1201** comprising key-value pairs of data words (each comprising a string of bits) and their corresponding calculated binary Huffman codewords. A subsequent dataset may be received, and compared to the word library **1603** to determine the proper codewords to use in order to encode the dataset. Words in the dataset are checked against the word library and appropriate encodings are appended to a data stream **1604**. If a word is mismatched within the word library and the dataset, meaning that it is present in the dataset but not the word library, then a mismatched code is appended, followed by the unencoded original word. If a word has a match within the word library, then the appropriate codeword in the word library is appended to the data stream. Such a data stream may then be stored or transmitted **1605** to a destination as desired. For the purposes of decoding, an already-encoded data stream may be received and compared **1606**, and un-encoded words may be appended to a new data stream **1607** depending on word matches found between the encoded data stream and the word library that is present. A matching codeword that is found in a word library is replaced with the matching word and appended to a data stream, and a mismatch code found in a data stream is deleted and the following unencoded word is re-appended to a new data stream, the inverse of the process of encoding described earlier. Such a data stream may then be stored or transmitted **1608** as desired.
(161) FIG. **17** is a method diagram illustrating possible use of a hybrid encoder/decoder to improve the compression ratio, according to a preferred aspect. A second Huffman binary tree may be created **1701**, having a shorter maximum length of codewords than a first Huffman binary tree **1602**, allowing a word library to be filled with every combination of codeword possible in this shorter Huffman binary tree **1702**. A word library may be filled with these Huffman codewords and words from a dataset **1702**, such that a hybrid encoder/decoder **1304**, **1503** may receive any mismatched words from a dataset for which encoding has been attempted with a first Huffman binary tree **1703**, **1604** and parse previously mismatched words into new partial codewords (that is, codewords that are each a substring of an original mismatched codeword) using the second Huffman binary tree **1704**. In this way, an incomplete word library may be supplemented by a second word library. New codewords attained in this way may then be returned to a transmission encoder **1705**, **1500**. In the event that an encoded dataset is received for decoding, and there is a mismatch code indicating that additional coding is needed, a mismatch code may be removed and the unencoded word used to generate a new codeword as before **1706**, so that a transmission encoder **1500** may have the word and newly generated codeword added to its word library **1707**, to prevent further mismatching and errors in encoding and decoding.
(162) It will be recognized by a person skilled in the art that the methods described herein can be applied to data in any form. For example, the method described herein could be used to store genetic data, which has four data units: C, G, A, and T. Those four data units can be represented as 2 bit sequences: 00, 01, 10, and 11, which can be processed and stored using the method described herein. It will be recognized by a person skilled in the art that certain embodiments of the methods described herein may have uses other than data storage. For example, because the data is stored in reference code form, it cannot be reconstructed without the availability of the library of sourceblocks. This is effectively a form of encryption, which could be used for cyber security purposes. As another example, an embodiment of the method described herein could be used to store backup copies of data, provide for redundancy in the event of server failure, or provide

additional security against cyberattacks by distributing multiple partial copies of the library among computers are various locations, ensuring that at least two copies of each sourceblock exist in different locations within the network.

(163) FIG. **18** is a flow diagram illustrating the use of a data encoding system used to recursively encode data to further reduce data size. Data may be input **1805** into a data deconstruction engine **102** to be deconstructed into code references, using a library of code references based on the input **1810**. Such example data is shown in a converted, encoded format **1815**, highly compressed, reducing the example data from 96 bits of data, to 12 bits of data, before sending this newly encoded data through the process again **1820**, to be encoded by a second library **1825**, reducing it even further. The newly converted data **1830** is shown as only 6 bits in this example, thus a size of 6.25% of the original data packet. With recursive encoding, then, it is possible and implemented in the system to achieve increasing compression ratios, using multi-layered encoding, through recursively encoding data. Both initial encoding libraries **1810** and subsequent libraries **1825** may be achieved through machine learning techniques to find optimal encoding patterns to reduce size, with the libraries being distributed to recipients prior to transfer of the actual encoded data, such that only the compressed data **1830** must be transferred or stored, allowing for smaller data footprints and bandwidth requirements. This process can be reversed to reconstruct the data. While this example shows only two levels of encoding, recursive encoding may be repeated any number of times. The number of levels of recursive encoding will depend on many factors, a non-exhaustive list of which includes the type of data being encoded, the size of the original data, the intended usage of the data, the number of instances of data being stored, and available storage space for codebooks and libraries. Additionally, recursive encoding can be applied not only to data to be stored or transmitted, but also to the codebooks and/or libraries, themselves. For example, many installations of different libraries could take up a substantial amount of storage space. Recursively encoding those different libraries to a single, universal library would dramatically reduce the amount of storage space required, and each different library could be reconstructed as necessary to reconstruct incoming streams of data.

(164) FIG. **20** is a flow diagram of an exemplary method used to detect anomalies in received encoded data and producing a warning. A system may have trained encoding libraries **2010**, before data is received from some source such as a network connected device or a locally connected device including USB connected devices, to be decoded **2020**. Decoding in this context refers to the process of using the encoding libraries to take the received data and attempt to use encoded references to decode the data into its original source **2030**, potentially more than once if recursive encoding was used, but not necessarily more than once. An anomaly detector **1910** may be configured to detect a large amount of un-encoded data **2040** in the midst of encoded data, by locating data or references that do not appear in the encoding libraries, indicating at least an anomaly, and potentially data tampering or faulty encoding libraries. A flag or warning is set by the system **2050**, allowing a user to be warned at least of the presence of the anomaly and the characteristics of the anomaly. However, if a large amount of invalid references or unencoded data are not present in the encoded data that is attempting to be decoded, the data may be decoded and output as normal **2060**, indicating no anomaly has been detected.

(165) FIG. **21** is a flow diagram of a method used for Distributed Denial of Service (DDOS) attack denial. A system may have trained encoding libraries **2110**, before data is received from some source such as a network connected device or a locally connected device including USB connected devices, to be decoded **2120**. Decoding in this context refers to the process of using the encoding libraries to take the received data and attempt to use encoded references to decode the data into its original source **2130**, potentially more than once if recursive encoding was used, but not necessarily more than once. A DDOS detector **1920** may be configured to detect a large amount of repeating data **2140** in the encoded data, by locating data or references that repeat many times over (the number of which can be configured by a user or administrator as need be), indicating a

possible DDOS attack. A flag or warning is set by the system **2150**, allowing a user to be warned at least of the presence of a possible DDOS attack, including characteristics about the data and source that initiated the flag, allowing a user to then block incoming data from that source. However, if a large amount of repeat data in a short span of time is not detected, the data may be decoded and output as normal **2160**, indicating no DDOS attack has been detected.

(166) FIG. **23** is a flow diagram of an exemplary method used to enable high-speed data mining of repetitive data. A system may have trained encoding libraries **2310**, before data is received from some source such as a network connected device or a locally connected device including USB connected devices, to be analyzed **2320** and decoded **2330**. When determining data for analysis, users may select specific data to designate for decoding **2330**, before running any data mining or analytics functions or software on the decoded data **2340**. Rather than having traditional decryption and decompression operate over distributed drives, data can be regenerated immediately using the encoding libraries disclosed herein, as it is being searched. Using methods described in FIG. **9** and FIG. **11**, data can be stored, retrieved, and decoded swiftly for searching, even across multiple devices, because the encoding library may be on each device. For example, if a group of servers host codewords relevant for data mining purposes, a single computer can request these codewords, and the codewords can be sent to the recipient swiftly over the bandwidth of their connection, allowing the recipient to locally decode the data for immediate evaluation and searching, rather than running slow, traditional decompression algorithms on data stored across multiple devices or transfer larger sums of data across limited bandwidth.

(167) FIG. **25** is a flow diagram of an exemplary method used to encode and transfer software and firmware updates to a device for installation, for the purposes of reduced bandwidth consumption. A first system may have trained code libraries or “codebooks” present **2510**, allowing for a software update of some manner to be encoded **2520**. Such a software update may be a firmware update, operating system update, security patch, application patch or upgrade, or any other type of software update, patch, modification, or upgrade, affecting any computer system. A codebook for the patch must be distributed to a recipient **2530**, which may be done beforehand and either over a network or through a local or physical connection, but must be accomplished at some point in the process before the update may be installed on the recipient device **2560**. An update may then be distributed to a recipient device **2540**, allowing a recipient with a codebook distributed to them **2530** to decode the update **2550** before installation **2560**. In this way, an encoded and thus heavily compressed update may be sent to a recipient far quicker and with less bandwidth usage than traditional lossless compression methods for data, or when sending data in uncompressed formats. This especially may benefit large distributions of software and software updates, as with enterprises updating large numbers of devices at once.

(168) FIG. **27** is a flow diagram of an exemplary method used to encode new software and operating system installations for reduced bandwidth required for transference. A first system may have trained code libraries or “codebooks” present **2710**, allowing for a software installation of some manner to be encoded **2720**. Such a software installation may be a software update, operating system, security system, application, or any other type of software installation, execution, or acquisition, affecting a computer system. An encoding library or “codebook” for the installation must be distributed to a recipient **2730**, which may be done beforehand and either over a network or through a local or physical connection, but must be accomplished at some point in the process before the installation can begin on the recipient device **2760**. An installation may then be distributed to a recipient device **2740**, allowing a recipient with a codebook distributed to them **2730** to decode the installation **2750** before executing the installation **2760**. In this way, an encoded and thus heavily compressed software installation may be sent to a recipient far quicker and with less bandwidth usage than traditional lossless compression methods for data, or when sending data in uncompressed formats. This especially may benefit large distributions of software and software updates, as with enterprises updating large numbers of devices at once.

(169) FIG. **31** is a method diagram illustrating the steps **3100** involved in using an embodiment of the codebook training system to update a codebook. The process begins when requested data is received **3101** by a codebook training module. The requested data may comprise a plurality of sourceblocks. Next, the received data may be stored in a cache and formatted into a test dataset **3102**. The next step is to retrieve the previously computed probability distribution associated with the previous (most recent) training dataset from a storage device **3103**. Using one or more algorithms, measure and/or estimate and record the probability distribution of the test dataset **3104**. The step after that is to compare the measured/estimated probability distributions of the test dataset and the previous training dataset to compute the difference in distribution statistics between the two datasets **3105**. If the test dataset probability distribution exceeds a pre-determined difference threshold, then the test dataset will be used to retrain the encoding/decoding algorithms **3106** to reflect the new distribution of the incoming data to the encoder/decoder system. The retrained algorithms may then be used to create new data sourceblocks **3107** that better capture the nature of the data being received. These newly created data sourceblocks may then be used to create new codewords and update a codebook **3108** with each new data sourceblock and its associated new codeword. Last, the updated codebooks may be sent to encoding and decoding machines **3109** in order to ensure the encoding/decoding system function properly.
(170) Adaptive Data Processing System with Dynamic Technique Selection and Feedback-Driven Optimization
(171) FIG. **41** is a block diagram illustrating exemplary architecture of adaptive data processing system **4100**. System **4100** comprises dynamic processing subsystem **4110**, feedback loop mechanism **4120**, output subsystem **4130**, and system controller **4140**. Input data **4101** enters system **4100** and is processed to produce output **4102**.
(172) Input data **4101** may come from various sources. These can include external data sources such as sensors, databases, or file systems. Network streams transmitting data over a network can also serve as input. Users may directly input data into system **4100**. Additionally, data deconstruction engine **201** may pass data to system **4100** for enhanced processing. The versatility of input sources allows system **4100** to adapt to diverse data processing scenarios.
(173) Output **4102** from system **4100** has multiple potential uses. It may be transmitted over a network to a recipient system, leveraging the efficient compression and encryption applied by system **4100**. The processed data can be stored in its compressed and encrypted form for later use, optimizing storage resources. In some cases, output **4102** might be passed to other systems for additional processing or analysis. Data reconstruction engine **301** may receive output **4102** to reconstruct the original data when needed. The flexibility of output use cases demonstrates the adaptability of system **4100** to various data management and processing requirements.
(174) Dynamic processing subsystem **4110** receives input data **4101** and performs initial analysis and processing. Characteristic analyzer **4111** examines input data **4101** to determine its type, structure, and other relevant properties. Characteristic analyzer **4111** may employ machine learning techniques such as, for example, convolutional neural networks for image data classification or recurrent neural networks for time-series data analysis. This information is passed to probability distribution estimator **4112**, which calculates an estimated probability distribution of the input data. Probability distribution estimator **4112** might, in some implementations, use kernel density estimation or Bayesian inference techniques to estimate the distribution of input data. Distribution comparator **4113** then compares this estimated distribution with a previous training dataset distribution retrieved from monitoring database. One approach in an embodiment for the distribution comparator **4113** is to use statistical distance measures such as Kullback-Leibler divergence or Wasserstein distance to quantify the difference between distributions. Based on this comparison, transformation matrix creator **4114** generates a matrix to transform the input data. The transformation matrix creator **4114** may, in some cases, employ singular value decomposition or principal component analysis to generate efficient transformation matrices. Data transformer **4115**

applies this matrix to convert the input data into a dyadic distribution. Stream generator **4116** then produces a main data stream of transformed data and a secondary stream containing transformation information. Technique selector and applicator **4117** chooses and implements appropriate processing algorithms based on the data characteristics and current system performance. In certain implementations, the technique selector and applicator **4117** may use a decision tree algorithm or a reinforcement learning model to dynamically select optimal processing techniques. Huffman coder **4118** compresses the main data stream using Huffman coding. While Huffman coding is used in this example, other entropy encoding techniques such as arithmetic coding or range coding could also be employed by the system. Adaptive adjuster **4119** fine-tunes the selection and application of processing techniques based on real-time performance metrics. In an embodiment, characteristic analyzer **4111** receives data from data deconstruction engine **201**, analyzing it before sourceblock creation. Technique selector and applicator **4117** interfaces with compression engine **3610**, determining optimal compression strategies.
(175) Feedback loop mechanism **4120** monitors and optimizes system performance over time. Effectiveness monitor **4121** tracks the performance of applied processing techniques, collecting metrics such as compression ratio and processing time. The effectiveness monitor **4121** might utilize statistical process control techniques or anomaly detection algorithms to identify significant changes in system performance in various embodiments. This data is passed to knowledge base updater **4122**, which stores and organizes the performance information. Historical performance analyzer **4123** examines this accumulated data to identify trends and patterns, generating insights to improve future processing decisions. In some implementations, the historical performance analyzer **4123** could use time series forecasting methods like ARIMA or machine learning techniques such as long short-term memory (LSTM) networks to predict future performance trends. Effectiveness monitor **4121** collects performance data from compression engine **3610** and decompression engine **3710**, incorporating this information into its analysis.
(176) Output subsystem **4130** prepares the processed data for transmission or storage. Codeword creator **4131** generates new codewords for the processed data segments. Stream combiner **4132** merges the compressed main data stream with the secondary stream containing transformation information. Data packager **4133** adds metadata describing the applied processing techniques to the combined data stream. Security implementer **4134** applies encryption and other protective measures to the packaged data. The security implementer **4134** may employ a variety of encryption algorithms, such as AES for symmetric encryption or RSA for asymmetric encryption, depending on the security requirements of the data. Data transmitter **4135** then sends the secured, packaged data to its intended destination. In an embodiment, data packager **4133** prepares processed data and metadata for data reconstruction engine **301**, ensuring compatibility with existing reconstruction processes.
(177) System controller **4140** oversees and coordinates the operations of all other subsystems. Subsystem coordinator **4141** manages interactions between dynamic processing subsystem **4110**, feedback loop mechanism **4120**, and output subsystem **4130**, ensuring proper sequencing of operations. Subsystem coordinator **4141** may comprise a state machine implementation and a task scheduler to manage the workflow between subsystems. It may include inter-process communication protocols and synchronization mechanisms to prevent conflicts.
(178) External interface manager **4142** handles communication with existing systems such as data deconstruction engine **201**, compression engine **3610**, and library manager **103**. External interface manager **4142** may include a set of API adapters for various external systems, data format converters, and a message queue system for managing data input and output flows. Mode selector **4143** determines the operating mode (lossless, lossy, or modified lossless) based on current conditions and requirements. Mode selector **4143** may incorporate a decision engine with predefined rules and thresholds, as well as a configuration interface for updating these rules based on system requirements.

(179) Resource allocator **4144** optimizes the use of computational resources across subsystems. Resource allocator **4144** may include a resource monitoring subsystem, a predictive modeling component for anticipating resource needs, and/or a dynamic allocation algorithm for distributing resources based on current demands and priorities. Error handler **4145** detects and manages error conditions, implementing recovery procedures as needed. Error handler **4145** may comprise an error logging system, a pattern recognition module for identifying recurring errors, and a set of predefined recovery procedures for different error types, depending on the embodiment.

(180) Configuration manager **4146** maintains system-wide settings and applies updates based on performance data. Configuration Manager **4146** may include a centralized configuration database, a version control system for tracking changes, and a validation subsystem to ensure the consistency and correctness of configurations across all subsystems, based on the embodiment employed. Logging and monitoring service **4147** maintains system logs and provides real-time performance monitoring. Logging and monitoring service **4147** may incorporate a distributed logging architecture, log rotation and archiving mechanisms, real-time analytics engines for processing log data, and interfaces for integration with external monitoring and alerting systems. External interface manager **4142** facilitates communication between system **4100** and other components such as data deconstruction engine **201**, library manager **103**, and data reconstruction engine **301**.

(181) In operation, input data **4101** flows through dynamic processing subsystem **4110**, where it is analyzed, transformed, and compressed. Feedback loop mechanism **4120** continuously monitors this process, providing insights to improve performance over time. Output subsystem **4130** then packages and secures the processed data before transmission. System controller **4140** coordinates these operations, managing resources and interactions with external systems throughout the process. The resulting output **4102** is a compressed, secure data stream optimized for efficient storage or transmission.

(182) FIG. **42** is a method diagram illustrating the use of adaptive data processing system **4100**. Input data **4101** is received by the dynamic processing subsystem **4110**, where it undergoes initial processing and analysis **4201**. The characteristic analyzer **4111** examines the input data to determine its type, structure, and other relevant properties **4202**. A probability distribution of the input data is then estimated by the probability distribution estimator **4112** and compared to previous distributions stored in the monitoring database by the distribution comparator **4113**, allowing the system to detect any significant changes in data characteristics **4203**. Based on this analysis, a transformation matrix is created by the transformation matrix creator **4114** and applied to the data by the data transformer **4115**, converting it into a dyadic distribution for more efficient processing **4204**. The stream generator **4116** then produces main and secondary data streams, which are processed using techniques dynamically selected by the technique selector and applicator **4117** based on the analyzed characteristics and current system performance **4205**. During this process, the external interface manager **4142** facilitates communication between the adaptive data processing system **4100** and other components such as the data deconstruction engine **201**, library manager **103**, and data reconstruction engine **301**, ensuring seamless integration with the existing data processing infrastructure. The main data stream undergoes compression using Huffman coding **4118**, with the adaptive adjuster **4119** fine-tuning the process based on real-time performance metrics **4206**. Throughout this process, the feedback loop mechanism **4120** monitors the effectiveness of the applied techniques, updating the knowledge base with performance data to influence and optimize future processing decisions **4207**. Once processing is complete, the output subsystem **4130** takes over, packaging the processed data with metadata describing the applied techniques, implementing security measures, and preparing it for transmission **4208**. Finally, the data transmitter **4135** sends the fully processed and secured output **4102** to its intended destination, completing the adaptive data processing cycle **4209**.

(183) FIG. **43** is a method diagram illustrating the use of dynamic processing subsystem **4110**. The process begins as input data is thoroughly analyzed by the characteristic analyzer **4111** to determine

its type, structure, and other relevant properties, which may include data format, entropy levels, and potential patterns **4301**. In this phase, the characteristic analyzer **4111** may receive data directly from the data deconstruction engine **201**, analyzing it before sourceblock creation, while the technique selector and applicator **4117** interfaces with the compression engine **3610** to determine optimal compression strategies. Based on this analysis, a probability distribution of the input data is estimated by the probability distribution estimator **4112**, using techniques such as histogram creation or kernel density estimation **4302**. This estimated distribution is then compared to a previous training dataset distribution retrieved from the monitoring database by the distribution comparator **4113**, allowing the system to detect any significant changes or drift in data characteristics **4303**. Utilizing the results of this comparison and the properties of the input data, a transformation matrix is created by the transformation matrix creator **4114**, designed to map the input distribution to a target dyadic distribution **4304**. The data transformer **4115** then applies this matrix to transform the input data into a dyadic distribution, optimizing it for subsequent processing steps **4305**. From this transformed data, the stream generator **4116** produces a main data stream of transformed data and a secondary stream containing transformation information **4306**. The technique selector and applicator **4117** then dynamically chooses and implements appropriate processing techniques for these streams, based on the analyzed characteristics and current system performance **4307**. The main data stream undergoes compression using Huffman coding, performed by the Huffman coder **4118**, which assigns shorter codes to more frequent symbols for efficient representation **4308**. Throughout this process, the adaptive adjuster **4119** continually fine-tunes the selection and application of processing techniques based on real-time performance metrics, ensuring optimal efficiency and effectiveness of the dynamic processing subsystem **4309**.
(184) FIG. **44** is a method diagram illustrating the use of feedback loop mechanism **4120**. The process begins as real-time performance metrics, such as compression ratio, processing time, and error rates, are continuously collected by the effectiveness monitor **4121** from both the dynamic processing subsystem **4110** and output subsystem **4130** **4401**. Throughout this process, the effectiveness monitor **4121** may also collect performance data from external components such as the compression engine **3610** and decompression engine **3710**, providing a comprehensive view of the entire data processing pipeline and enabling optimizations that span across multiple system components. This raw performance data is then organized and structured by the knowledge base updater **4122**, transforming it into a format suitable for analysis and long-term storage **4402**. The structured performance data is subsequently stored in the knowledge base, a persistent storage system managed by the knowledge base updater **4122**, ensuring data integrity and consistency across updates **4403**. To analyze system performance over time, historical performance data is retrieved from the knowledge base by the historical performance analyzer **4123** **4404**. This analyzer applies sophisticated time series analysis techniques to the retrieved data, identifying trends, patterns, and potential anomalies in system performance **4405**. The historical performance analyzer **4123** then establishes correlations between performance metrics and specific processing techniques or data characteristics, providing deeper insights into the system's behavior **4406**. Based on these analyses, the historical performance analyzer **4123** generates actionable insights and recommendations for optimizing processing techniques, potentially using machine learning algorithms to predict future performance **4407**. These valuable insights are then fed back to the dynamic processing subsystem **4110**, particularly to the adaptive adjuster **4119**, enabling real-time optimization of system behavior **4408**. Finally, the technique selector and applicator **4117** adjusts its selection criteria based on the received insights, effectively influencing future processing decisions and completing the feedback loop **4409**. This continuous cycle of monitoring, analysis, and adjustment allows the system to learn from its own performance and constantly improve its efficiency and effectiveness.
(185) FIG. **45** is a method diagram illustrating the use of output subsystem **4130**. The process begins as new codewords for processed data segments are created by the codeword creator **4131**,

using techniques such as hash functions to generate unique identifiers for each data pattern **4501**. The stream combiner **4132** then merges the compressed main data stream and the secondary data stream containing transformation information, interleaving them according to a predefined protocol **4502**. Concurrently, metadata describing the applied processing techniques is generated by the data packager **4133**, which may include information about compression methods and transformation parameters **4503**. This metadata is then combined with the merged data stream by the data packager **4133**, ensuring that all necessary information for data reconstruction is included **4504**. To protect the processed data, the security implementer **4134** applies robust encryption algorithms, such as AES or RSA, to the packaged data **4505**. Additional security measures, including integrity checks and digital signatures, are added by the security implementer **4134** to prevent unauthorized alterations **4506**. The data transmitter **4135** then prepares the secured data for transmission, adapting it to the appropriate network protocols such as TCP/IP or HTTP **4507**. To ensure reliable transmission, error detection and correction codes are implemented by the data transmitter **4135**, allowing for data recovery in case of transmission errors **4508**. Finally, the fully processed, secured, and packaged data is transmitted to the recipient system by the data transmitter **4135**, completing the output process **4509**. Throughout this process, the output subsystem **4130** may interact with the data reconstruction engine **301**, ensuring that the packaged data and metadata are formatted in a way that facilitates efficient and accurate data reconstruction at the receiving end.

(186) FIG. **46** is a method diagram illustrating the use of system controller **4140**. The process begins as the subsystem coordinator **4141** continuously monitors and manages the current status of each subsystem, maintaining a state machine to ensure proper sequencing of operations **4601**. Concurrently, the external interface manager **4142** handles communication with external systems, such as the data deconstruction engine **201**, compression engine **3610**, and library manager **103**, managing data input and output flows **4602**. Based on the current system conditions, user preferences, and input data characteristics, the mode selector **4143** determines the appropriate operating mode (lossless, lossy, or modified lossless) and adjusts system behavior accordingly **4603**. To optimize performance, the resource allocator **4144** dynamically distributes computational resources across subsystems, implementing load balancing algorithms to ensure efficient processing **4604**. Throughout the operation, the error handler **4145** vigilantly detects and manages error conditions, implementing appropriate error recovery procedures and providing error summaries to system administrators **4605**. The configuration manager **4146** maintains and updates system-wide configuration settings, ensuring consistency across all subsystems and managing configuration backups and rollbacks as needed **4606**. All system activities are meticulously logged by the logging and monitoring service **4147**, which implements log rotation and archiving to manage log file sizes effectively **4607**. This service also provides real-time monitoring of system performance, generating periodic system health reports and integrating with external monitoring systems **4608**. Finally, based on the logged data and performance metrics, the subsystem coordinator **4141** makes necessary adjustments to system operations, optimizing the overall performance of the adaptive data processing system **4609**. This continuous cycle of monitoring, analysis, and adjustment ensures that the system controller **4140** effectively orchestrates all components of the adaptive data processing system, maintaining optimal performance and adaptability.

(187) In a non-limiting use case example of adaptive data processing system **4100**, a large-scale genomic research facility processes vast amounts of DNA sequencing data. The facility receives terabytes of raw sequencing data daily from various high-throughput sequencing machines. This data needs to be efficiently compressed for storage and securely transmitted to collaborating institutions worldwide.

(188) As the raw sequencing data enters the system, the dynamic processing subsystem **4110** analyzes its characteristics. The characteristic analyzer **4111** identifies the data as DNA sequencing information and detects patterns specific to the sequencing platform used. The probability

distribution estimator **4112** calculates the frequency distribution of nucleotide sequences, which is then compared to previous distributions by the distribution comparator **4113**.
(189) Based on this analysis, the transformation matrix creator **4114** generates a custom matrix optimized for DNA data, which the data transformer **4115** uses to convert the input into a dyadic distribution. The stream generator **4116** then creates a main data stream of transformed sequencing data and a secondary stream containing transformation parameters.
(190) The technique selector and applicator **4117** chooses a combination of specialized genomic compression algorithms and encryption methods suitable for sensitive genetic data. The Huffman coder **4118** further compresses the main data stream, achieving high compression ratios by exploiting the repetitive nature of genomic data.
(191) Throughout this process, the feedback loop mechanism **4120** monitors the system's performance. It notices that the compression efficiency for a particular type of sequencing data has decreased over time. The historical performance analyzer **4123** identifies this trend and suggests adjustments to the transformation matrix and compression algorithms.
(192) The output subsystem **4130** then packages the compressed genomic data with metadata describing the processing techniques used. The security implementer **4134** applies strong encryption to protect the sensitive genetic information before transmission.
(193) The system controller **4140** oversees this entire process, dynamically allocating more computational resources to handle peak data influxes from sequencing runs. It also manages the secure interfaces with the facility's data storage systems and external research networks.
(194) This adaptive approach allows the genomic research facility to efficiently process, store, and share massive amounts of sequencing data, automatically adjusting to changes in data characteristics or processing requirements over time. The system's ability to learn and optimize its performance ensures that it remains effective even as sequencing technologies and data formats evolve.
(195) In another non-limiting use case example of adaptive data processing system **4100**, a global financial institution processes and analyzes vast amounts of market data, transaction records, and customer information across multiple time zones. This data needs to be efficiently compressed, securely stored, and quickly accessible for real-time analysis and reporting.
(196) As market data streams into the system, it first passes through the data deconstruction engine **201**, which breaks it down into manageable sourceblocks. These sourceblocks are then fed into the adaptive data processing system **4100** for further processing.
(197) The dynamic processing subsystem **4110** analyzes the characteristics of the incoming data. The characteristic analyzer **4111** identifies various data types, including numerical time series data, textual transaction records, and structured customer information. It works in conjunction with the data deconstruction engine **201** to optimize the analysis process.
(198) Based on this analysis, the technique selector and applicator **4117** interfaces with the compression engine **3610** to determine the most effective compression strategies for each data type. For time series data, it might select a specialized financial data compression algorithm, while for textual data, it could choose a more general-purpose compression method.
(199) Library manager **103** is utilized throughout this process, providing optimized reference codes for common patterns in financial data. The dynamic processing subsystem **4110** continually updates the library manager **103** with new patterns it discovers, improving the system's efficiency over time.
(200) As the data is processed, the feedback loop mechanism **4120** monitors the performance of the applied techniques. It collects data not only from internal components but also from the external compression engine **3610** and decompression engine **3710**, ensuring a comprehensive view of the system's performance.
(201) The output subsystem **4130** packages the processed data, including metadata that describes the applied techniques. This packaged data is then stored in a distributed storage system, with the

data reconstruction engine **301** able to quickly retrieve and reconstruct the data as needed for real-time analysis.

(202) The system controller **4140** orchestrates this entire process, managing the interfaces between the adaptive data processing system **4100** and other components like the data deconstruction engine **201** and library manager **103**. It dynamically adjusts the operating mode based on current market conditions and analysis needs, switching between lossless mode for critical financial records and lossy mode for less sensitive market data streams.

(203) This adaptive and integrated approach allows the financial institution to efficiently process, store, and analyze massive amounts of diverse financial data. The system's ability to work seamlessly with existing data processing infrastructure while continuously optimizing its performance ensures that the institution can respond quickly to market changes and regulatory requirements, maintaining its competitive edge in the fast-paced financial world.

(204) In another non-limiting use case example of adaptive data processing system **4100**, a multinational aerospace company uses the system to manage and process complex engineering data from various stages of aircraft design, manufacturing, and maintenance.

(205) The company receives diverse data types including 3D CAD models, simulation results, sensor data from aircraft testing, and maintenance logs from operational aircraft. This data needs to be efficiently processed, stored, and made accessible across multiple global design centers and manufacturing facilities.

(206) As engineering data enters the system, the dynamic processing subsystem **4110** analyzes its characteristics. The characteristic analyzer **4111** identifies different data types, such as large 3D model files, numerical simulation data, and textual maintenance logs. For 3D models, it detects specific CAD file formats and typical structural patterns.

(207) The probability distribution estimator **4112** calculates distribution patterns for each data type, which the distribution comparator **4113** then compares with historical data. This comparison might reveal, for instance, that recent aircraft designs have more complex geometries, affecting the data distribution of 3D models.

(208) Based on this analysis, the transformation matrix creator **4114** generates customized matrices for each data type. The data transformer **4115** then converts the input into optimal dyadic distributions, separately handling the 3D model data, simulation results, and textual information.

(209) The technique selector and applicator **4117** chooses specialized compression algorithms for each data type. For 3D models, it might select advanced geometry compression techniques, while for sensor data, it could choose algorithms optimized for time-series data.

(210) Throughout the process, the feedback loop mechanism **4120** monitors performance. The effectiveness monitor **4121** might notice that compression efficiency for certain simulation data has improved, prompting the historical performance analyzer **4123** to investigate and potentially recommend this technique for similar data types.

(211) The output subsystem **4130** packages the processed data, with the data packager **4133** including detailed metadata about the applied techniques. This is crucial for ensuring that design teams in different locations can correctly interpret and use the data.

(212) The security implementer **4134** applies stringent encryption to protect sensitive design information before it's transmitted between global locations or stored in the company's distributed data centers.

(213) The system controller **4140** oversees the entire operation, with the mode selector **4143** dynamically switching between lossless mode for critical design data and lossy mode for less sensitive information like preliminary simulation results. The resource allocator **4144** ensures that sufficient computational power is available during peak times, such as during major design reviews or aircraft testing phases.

(214) This adaptive approach allows the aerospace company to efficiently manage its complex, varied, and sensitive engineering data across its global operations. The system's ability to recognize

and optimally process different data types, combined with its continuous self-improvement, ensures that the company can handle growing data volumes and complexities as aircraft designs become more sophisticated. This efficiency in data management translates to faster design iterations, improved collaboration between global teams, and ultimately, more innovative and reliable aircraft designs.
(215) It should be understood by one skilled in the art that the applications of system **4100** are not limited to the use case examples. For example, the adaptive data processing system could be employed in smart city infrastructures to efficiently manage and process data from numerous IoT sensors, traffic cameras, and public transportation systems. In the field of climate science, it could handle vast datasets from satellites, weather stations, and ocean buoys, optimizing storage and enabling faster analysis of climate patterns. The system could revolutionize telemedicine by facilitating the secure transmission and storage of high-resolution medical imaging data and real-time patient monitoring information. In the entertainment industry, it could streamline the production and distribution of high-definition video content, efficiently managing the enormous data volumes involved in modern CGI-heavy film production. For autonomous vehicle development, the system could process and compress the massive amounts of sensor and camera data generated during test drives, enabling more efficient data sharing among development teams. In each of these applications, the system's ability to adapt to different data types, optimize processing techniques, and continuously improve its performance would provide significant advantages in data management, storage efficiency, and processing speed.
(216) Hardware Architecture
(217) Generally, the techniques disclosed herein may be implemented on hardware or a combination of software and hardware. For example, they may be implemented in an operating system kernel, in a separate user process, in a library package bound into network applications, on a specially constructed machine, on an application-specific integrated circuit (ASIC), or on a network interface card.
(218) Software/hardware hybrid implementations of at least some of the aspects disclosed herein may be implemented on a programmable network-resident machine (which should be understood to include intermittently connected network-aware machines) selectively activated or reconfigured by a computer program stored in memory. Such network devices may have multiple network interfaces that may be configured or designed to utilize different types of network communication protocols. A general architecture for some of these machines may be described herein in order to illustrate one or more exemplary means by which a given unit of functionality may be implemented. According to specific aspects, at least some of the features or functionalities of the various aspects disclosed herein may be implemented on one or more general-purpose computers associated with one or more networks, such as for example an end-user computer system, a client computer, a network server or other server system, a mobile computing device (e.g., tablet computing device, mobile phone, smartphone, laptop, or other appropriate computing device), a consumer electronic device, a music player, or any other suitable electronic device, router, switch, or other suitable device, or any combination thereof. In at least some aspects, at least some of the features or functionalities of the various aspects disclosed herein may be implemented in one or more virtualized computing environments (e.g., network computing clouds, virtual machines hosted on one or more physical computing machines, or other appropriate virtual environments).
(219) Referring now to FIG. **32**, there is shown a block diagram depicting an exemplary computing device **10** suitable for implementing at least a portion of the features or functionalities disclosed herein. Computing device **10** may be, for example, any one of the computing machines listed in the previous paragraph, or indeed any other electronic device capable of executing software- or hardware-based instructions according to one or more programs stored in memory. Computing device **10** may be configured to communicate with a plurality of other computing devices, such as clients or servers, over communications networks such as a wide area network a metropolitan area

network, a local area network, a wireless network, the Internet, or any other network, using known protocols for such communication, whether wireless or wired.

(220) In one aspect, computing device **10** includes one or more central processing units (CPU) **12**, one or more interfaces **15**, and one or more busses **14** (such as a peripheral component interconnect (PCI) bus). When acting under the control of appropriate software or firmware, CPU **12** may be responsible for implementing specific functions associated with the functions of a specifically configured computing device or machine. For example, in at least one aspect, a computing device **10** may be configured or designed to function as a server system utilizing CPU **12**, local memory **11** and/or remote memory **16**, and interface(s) **15**. In at least one aspect, CPU **12** may be caused to perform one or more of the different types of functions and/or operations under the control of software modules or components, which for example, may include an operating system and any appropriate applications software, drivers, and the like.

(221) CPU **12** may include one or more processors **13** such as, for example, a processor from one of the Intel, ARM, Qualcomm, and AMD families of microprocessors. In some aspects, processors **13** may include specially designed hardware such as application-specific integrated circuits (ASICs), electrically erasable programmable read-only memories (EEPROMs), field-programmable gate arrays (FPGAs), and so forth, for controlling operations of computing device **10**. In a particular aspect, a local memory **11** (such as non-volatile random access memory (RAM) and/or read-only memory (ROM), including for example one or more levels of cached memory) may also form part of CPU **12**. However, there are many different ways in which memory may be coupled to system **10**. Memory **11** may be used for a variety of purposes such as, for example, caching and/or storing data, programming instructions, and the like. It should be further appreciated that CPU **12** may be one of a variety of system-on-a-chip (SOC) type hardware that may include additional hardware such as memory or graphics processing chips, such as a QUALCOMM SNAPDRAGON™ or SAMSUNG EXYNOS™ CPU as are becoming increasingly common in the art, such as for use in mobile devices or integrated devices.

(222) As used herein, the term “processor” is not limited merely to those integrated circuits referred to in the art as a processor, a mobile processor, or a microprocessor, but broadly refers to a microcontroller, a microcomputer, a programmable logic controller, an application-specific integrated circuit, and any other programmable circuit.

(223) In one aspect, interfaces **15** are provided as network interface cards (NICs). Generally, NICs control the sending and receiving of data packets over a computer network; other types of interfaces **15** may for example support other peripherals used with computing device **10**. Among the interfaces that may be provided are Ethernet interfaces, frame relay interfaces, cable interfaces, DSL interfaces, token ring interfaces, graphics interfaces, and the like. In addition, various types of interfaces may be provided such as, for example, universal serial bus (USB), Serial, Ethernet, FIREWIRE™, THUNDERBOLT™, PCI, parallel, radio frequency (RF), BLUETOOTH™, near-field communications (e.g., using near-field magnetics), 802.11 (Wi-Fi), frame relay, TCP/IP, ISDN, fast Ethernet interfaces, Gigabit Ethernet interfaces, Serial ATA (SATA) or external SATA (ESATA) interfaces, high-definition multimedia interface (HDMI), digital visual interface (DVI), analog or digital audio interfaces, asynchronous transfer mode (ATM) interfaces, high-speed serial interface (HSSI) interfaces, Point of Sale (POS) interfaces, fiber data distributed interfaces (FDDIs), and the like. Generally, such interfaces **15** may include physical ports appropriate for communication with appropriate media. In some cases, they may also include an independent processor (such as a dedicated audio or video processor, as is common in the art for high-fidelity A/V hardware interfaces) and, in some instances, volatile and/or non-volatile memory (e.g., RAM).

(224) Although the system shown in FIG. **32** illustrates one specific architecture for a computing device **10** for implementing one or more of the aspects described herein, it is by no means the only device architecture on which at least a portion of the features and techniques described herein may be implemented. For example, architectures having one or any number of processors **13** may be

used, and such processors **13** may be present in a single device or distributed among any number of devices. In one aspect, a single processor **13** handles communications as well as routing computations, while in other aspects a separate dedicated communications processor may be provided. In various aspects, different types of features or functionalities may be implemented in a system according to the aspect that includes a client device (such as a tablet device or smartphone running client software) and server systems (such as a server system described in more detail below).

(225) Regardless of network device configuration, the system of an aspect may employ one or more memories or memory modules (such as, for example, remote memory block **16** and local memory **11**) configured to store data, program instructions for the general-purpose network operations, or other information relating to the functionality of the aspects described herein (or any combinations of the above). Program instructions may control execution of or comprise an operating system and/or one or more applications, for example. Memory **16** or memories **11**, **16** may also be configured to store data structures, configuration data, encryption data, historical system operations information, or any other specific or generic non-program information described herein.

(226) Because such information and program instructions may be employed to implement one or more systems or methods described herein, at least some network device aspects may include nontransitory machine-readable storage media, which, for example, may be configured or designed to store program instructions, state information, and the like for performing various operations described herein. Examples of such nontransitory machine-readable storage media include, but are not limited to, magnetic media such as hard disks, floppy disks, and magnetic tape; optical media such as CD-ROM disks; magneto-optical media such as optical disks, and hardware devices that are specially configured to store and perform program instructions, such as read-only memory devices (ROM), flash memory (as is common in mobile devices and integrated systems), solid state drives (SSD) and “hybrid SSD” storage drives that may combine physical components of solid state and hard disk drives in a single hardware device (as are becoming increasingly common in the art with regard to personal computers), memristor memory, random access memory (RAM), and the like. It should be appreciated that such storage means may be integral and non-removable (such as RAM hardware modules that may be soldered onto a motherboard or otherwise integrated into an electronic device), or they may be removable such as swappable flash memory modules (such as “thumb drives” or other removable media designed for rapidly exchanging physical storage devices), “hot-swappable” hard disk drives or solid state drives, removable optical storage discs, or other such removable media, and that such integral and removable storage media may be utilized interchangeably. Examples of program instructions include both object code, such as may be produced by a compiler, machine code, such as may be produced by an assembler or a linker, byte code, such as may be generated by for example a JAVA™ compiler and may be executed using a Java virtual machine or equivalent, or files containing higher level code that may be executed by the computer using an interpreter (for example, scripts written in Python, Perl, Ruby, Groovy, or any other scripting language).

(227) In some aspects, systems may be implemented on a standalone computing system. Referring now to FIG. **33**, there is shown a block diagram depicting a typical exemplary architecture of one or more aspects or components thereof on a standalone computing system. Computing device **20** includes processors **21** that may run software that carry out one or more functions or applications of aspects, such as for example a client application **24**. Processors **21** may carry out computing instructions under control of an operating system **22** such as, for example, a version of MICROSOFT WINDOWS™ operating system, APPLE macOS™ or iOS™ operating systems, some variety of the Linux operating system, ANDROID™ operating system, or the like. In many cases, one or more shared services **23** may be operable in system **20**, and may be useful for providing common services to client applications **24**. Services **23** may for example be

WINDOWS™ services, user-space common services in a Linux environment, or any other type of common service architecture used with operating system **21**. Input devices **28** may be of any type suitable for receiving user input, including for example a keyboard, touchscreen, microphone (for example, for voice input), mouse, touchpad, trackball, or any combination thereof. Output devices **27** may be of any type suitable for providing output to one or more users, whether remote or local to system **20**, and may include for example one or more screens for visual output, speakers, printers, or any combination thereof. Memory **25** may be random-access memory having any structure and architecture known in the art, for use by processors **21**, for example to run software. Storage devices **26** may be any magnetic, optical, mechanical, memristor, or electrical storage device for storage of data in digital form (such as those described above, referring to FIG. **32**). Examples of storage devices **26** include flash memory, magnetic hard drive, CD-ROM, and/or the like.

(228) In some aspects, systems may be implemented on a distributed computing network, such as one having any number of clients and/or servers. Referring now to FIG. **34**, there is shown a block diagram depicting an exemplary architecture **30** for implementing at least a portion of a system according to one aspect on a distributed computing network. According to the aspect, any number of clients **33** may be provided. Each client **33** may run software for implementing client-side portions of a system; clients may comprise a system **20** such as that illustrated in FIG. **33**. In addition, any number of servers **32** may be provided for handling requests received from one or more clients **33**. Clients **33** and servers **32** may communicate with one another via one or more electronic networks **31**, which may be in various aspects any of the Internet, a wide area network, a mobile telephony network (such as CDMA or GSM cellular networks), a wireless network (such as Wi-Fi, WiMAX, LTE, and so forth), or a local area network (or indeed any network topology known in the art; the aspect does not prefer any one network topology over any other). Networks **31** may be implemented using any known network protocols, including for example wired and/or wireless protocols.

(229) In addition, in some aspects, servers **32** may call external services **37** when needed to obtain additional information, or to refer to additional data concerning a particular call. Communications with external services **37** may take place, for example, via one or more networks **31**. In various aspects, external services **37** may comprise web-enabled services or functionality related to or installed on the hardware device itself. For example, in one aspect where client applications **24** are implemented on a smartphone or other electronic device, client applications **24** may obtain information stored in a server system **32** in the cloud or on an external service **37** deployed on one or more of a particular enterprise's or user's premises.

(230) In some aspects, clients **33** or servers **32** (or both) may make use of one or more specialized services or appliances that may be deployed locally or remotely across one or more networks **31**. For example, one or more databases **34** may be used or referred to by one or more aspects. It should be understood by one having ordinary skill in the art that databases **34** may be arranged in a wide variety of architectures and using a wide variety of data access and manipulation means. For example, in various aspects one or more databases **34** may comprise a relational database system using a structured query language (SQL), while others may comprise an alternative data storage technology such as those referred to in the art as "NoSQL" (for example, HADOOP CASSANDRA™, GOOGLE BIGTABLE™, and so forth). In some aspects, variant database architectures such as column-oriented databases, in-memory databases, clustered databases, distributed databases, or even flat file data repositories may be used according to the aspect. It will be appreciated by one having ordinary skill in the art that any combination of known or future database technologies may be used as appropriate, unless a specific database technology or a specific arrangement of components is specified for a particular aspect described herein. Moreover, it should be appreciated that the term "database" as used herein may refer to a physical database machine, a cluster of machines acting as a single database system, or a logical database within an

overall database management system. Unless a specific meaning is specified for a given use of the term “database”, it should be construed to mean any of these senses of the word, all of which are understood as a plain meaning of the term “database” by those having ordinary skill in the art.

(231) Similarly, some aspects may make use of one or more security systems **36** and configuration systems **35**. Security and configuration management are common information technology (IT) and web functions, and some amount of each are generally associated with any IT or web systems. It should be understood by one having ordinary skill in the art that any configuration or security subsystems known in the art now or in the future may be used in conjunction with aspects without limitation, unless a specific security **36** or configuration system **35** or approach is specifically required by the description of any specific aspect.

(232) FIG. **35** shows an exemplary overview of a computer system **40** as may be used in any of the various locations throughout the system. It is exemplary of any computer that may execute code to process data. Various modifications and changes may be made to computer system **40** without departing from the broader scope of the system and method disclosed herein. Central processor unit (CPU) **41** is connected to bus **42**, to which bus is also connected memory **43**, nonvolatile memory **44**, display **47**, input/output (I/O) unit **48**, and network interface card (NIC) **53**. I/O unit **48** may, typically, be connected to keyboard **49**, pointing device **50**, hard disk **52**, and real-time clock **51**. NIC **53** connects to network **54**, which may be the Internet or a local network, which local network may or may not have connections to the Internet. Also shown as part of system **40** is power supply unit **45** connected, in this example, to a main alternating current (AC) supply **46**. Not shown are batteries that could be present, and many other devices and modifications that are well known but are not applicable to the specific novel functions of the current system and method disclosed herein. It should be appreciated that some or all components illustrated may be combined, such as in various integrated applications, for example Qualcomm or Samsung system-on-a-chip (SOC) devices, or whenever it may be appropriate to combine multiple capabilities or functions into a single hardware device (for instance, in mobile devices such as smartphones, video game consoles, in-vehicle computer systems such as navigation or multimedia systems in automobiles, or other integrated hardware devices).

(233) In various aspects, functionality for implementing systems or methods of various aspects may be distributed among any number of client and/or server components. For example, various software modules may be implemented for performing various functions in connection with the system of any particular aspect, and such modules may be variously implemented to run on server and/or client components.

(234) The skilled person will be aware of a range of possible modifications of the various aspects described above. Accordingly, the present invention is defined by the claims and their equivalents.

Claims

1. A system for adaptive data processing, comprising: a computing device comprising a processor and memory; a dynamic processing subsystem comprising a plurality of programming instructions which, when operating on the processor, causes the processor to: receive input data; transform the input data into a dyadic distribution using a transformation matrix based on statistical properties of the input data; generate a main data stream of transformed data and a secondary data stream of transformation information; dynamically select and apply a combination of processing techniques, wherein the processing techniques are selected from a group consisting of transformation algorithms, encoding algorithms, compression algorithms, and encryption algorithms; compress the main data stream using Huffman coding; adaptively adjust the selection and application of processing techniques based on real-time performance metrics; a feedback loop mechanism configured to: monitor the effectiveness of the applied processing techniques; update a knowledge base with performance data; and influence future selections of processing techniques based on

historical performance; and an output subsystem configured to: create new codewords for processed data; combine the compressed main data stream and the secondary data stream into an output stream; package the processed data with metadata describing the applied techniques; implement security measures to protect the output stream; and transmit the packaged data and metadata to a recipient system; wherein the security measures include providing cryptographically secure random numbers for use in data transformation and implementing protections against side-channel attacks.
2. The system of claim 1, wherein when the input data is image data, the dynamic processing subsystem is configured to: apply a mathematical transform to the image data; and an entropy encoding algorithm to the transformed data.
3. The system of claim 1, wherein the dynamic processing subsystem is further configured to operate in multiple modes selected from the group consisting of: a lossless mode where both the main data stream and the secondary data stream are included in the output stream; a lossy mode where only the main data stream is included in the output stream; and a modified lossless mode wherein the main data stream is included in a first output stream and the secondary data stream is included in a second output stream.
4. The system of claim 3, wherein in the lossy mode, the dynamic processing subsystem is further configured to estimate the quality of the compressed and encrypted data compared to the original input data.
5. The system of claim 1, wherein transforming the input data into a dyadic distribution comprises: constructing a Huffman encoding based on the estimated probability distribution of the input data; partitioning the data space into overrepresented states and underrepresented states based on the Huffman encoding; and applying transformations to the input data using the transformation matrix to reshape the data distribution.
6. The system of claim 1, wherein the security measures further comprise applying a modified next-bit test.
7. A method for adaptive data processing, comprising the steps of: receiving input data; transforming the input data into a dyadic distribution using a transformation matrix based on statistical properties of the input data; generating a main data stream of transformed data and a secondary data stream of transformation information; dynamically selecting and applying a combination of processing techniques, wherein the processing techniques are selected from a group consisting of transformation algorithms, encoding algorithms, compression algorithms, and encryption algorithms; compressing the main data stream using Huffman coding; adaptively adjusting the selection and application of processing techniques based on real-time performance metrics; monitoring the effectiveness of the applied processing techniques; updating a knowledge base with performance data; influencing future selections of processing techniques based on historical performance; creating new codewords for processed data; combining the compressed main data stream and the secondary data stream into an output stream; packaging the processed data with metadata describing the applied techniques; implementing security measures to protect the output stream; and transmitting the packaged data and metadata to a recipient system; wherein the security measures include providing cryptographically secure random numbers for use in data transformation and implementing protections against side-channel attacks.
8. The method of claim 7, wherein when the input data is image data, the method further comprises: applying a mathematical transform to the image data; and applying an entropy encoding algorithm to the transformed data.
9. The method of claim 7, further comprising operating in multiple modes selected from the group consisting of: a lossless mode where both the main data stream and the secondary data stream are included in the output stream; a lossy mode where only the main data stream is included in the output stream; and a modified lossless mode wherein the main data stream is included in a first output stream and the secondary data stream is included in a second output stream.

10. The method of claim 9, wherein in the lossy mode, the method further comprises estimating the quality of the compressed and encrypted data compared to the original input data.

11. The method of claim 7, wherein transforming the input data into a dyadic distribution comprises: constructing a Huffman encoding based on the estimated probability distribution of the input data; partitioning the data space into overrepresented states and underrepresented states based on the Huffman encoding; and applying transformations to the input data using the transformation matrix to reshape the data distribution.

12. The method of claim 7, wherein the security measures further comprise applying a modified next-bit test.